Package ‘caugi’

May 8, 2026

Title Causal Graph Interface

Version 1.2.0

Description Create, query, and modify causal graphs. 'caugi' (Causal Graph Interface) is a causality-first, high performance graph package that provides a simple interface to build, structure, and examine causal relationships.

License MIT + file LICENSE

Language en-US

URL `https://caugi.org/`, `https://github.com/frederikfabriciusbjerre/caugi`

BugReports `https://github.com/frederikfabriciusbjerre/caugi/issues`

Depends R (>= 4.2)

Imports data.table, grid, S7, stats, methods

Suggests bnlearn, dagitty, devtools, ggm, graph, gRbase, igraph, jsonlite, knitr, MASS, Matrix, rextendr, rmarkdown, testthat, waldo, withr

VignetteBuilder knitr

Config/rextendr/version 0.4.2

Encoding UTF-8

RoxygenNote 7.3.3

SystemRequirements Cargo (Rust's package manager), rustc >= 1.80.0, xz

Config/Needs/website rmarkdown

NeedsCompilation yes

Author Frederik Fabricius-Bjerre [aut, cre, cph],
Johan Larsson [aut] (ORCID: <`https://orcid.org/0000-0002-4029-5945`>),
Michael Sachs [aut] (ORCID: <`https://orcid.org/0000-0002-1279-8676`>),
Bjarke Hautop Kristensen [aut]

Maintainer Frederik Fabricius-Bjerre <`frederik@fabriciusbjerre.dk`>

Repository CRAN

Date/Publication 2026-05-05 14:56:15 UTC

Contents

`add-caugi_plot-caugi_plot` *Compose Plots Horizontally*

Description

Arrange two plots side-by-side with configurable spacing. The `+` and `|` operators are equivalent and can be used interchangeably. Compositions can be nested to create complex multi-plot layouts.

Arguments

`e1`	A `caugi_plot` object (left plot)
`e2`	A `caugi_plot` object (right plot)

Details

The spacing between plots is controlled by the global option `caugi_options()$plot$spacing`, which defaults to `grid::unit(1, "lines")`. Compositions can be nested arbitrarily:

- `p1 + p2` - two plots side-by-side
- `(p1 + p2) + p3` - three plots in a row
- `(p1 + p2) / p3` - two plots on top, one below

Value

A `caugi_plot` object containing the composed layout

See Also

`caugi_options()` for configuring spacing and default styles

Other plotting: `caugi_layout()`, `caugi_layout_bipartite()`, `caugi_layout_circle()`, `caugi_layout_fruchterman_`
`caugi_layout_kamada_kawai()`, `caugi_layout_sugiyama()`, `caugi_layout_tiered()`, `caugi_plot()`, `divide-caugi_plot-caugi_plot`, `plot()`

Examples

```
cg1 <- caugi(A %-->% B, B %-->% C)
cg2 <- caugi(X %-->% Y, Y %-->% Z)

p1 <- plot(cg1, main = "Graph 1")
p2 <- plot(cg2, main = "Graph 2")
```

```
# Horizontal composition
p1 + p2
p1 | p2 # equivalent

# Adjust spacing
caugi_options(plot = list(spacing = grid::unit(2, "lines")))
p1 + p2
```

`adjustment_set` *Compute an adjustment set*

Description

Computes an adjustment set for `X -> Y` in a DAG.

Usage

```
adjustment_set(
  cg,
  X = NULL,
  Y = NULL,
  X_index = NULL,
  Y_index = NULL,
  type = c("optimal", "parents", "backdoor")
)
```

Arguments

`cg`	A `caugi` object.
`X`, `Y`	Node names.
`X_index`, `Y_index`	Optional numeric 1-based indices.
`type`	One of `"parents"`, `"backdoor"`, `"optimal"`. The `optimal` option computes the O-set.

Details

Types supported:

- `"parents"`: $\bigcup \mathrm{Pa}(X)$ minus $X \cup Y$
- `"backdoor"`: Pearl backdoor formula
- `"optimal"`: O-set (only for single `x` and single `y`)

Value

A character vector of node names representing the adjustment set.

See Also

Other adjustment: `all_adjustment_sets_admg()`, `all_backdoor_sets()`, `d_separated()`, `is_valid_adjustment_admg`
`is_valid_backdoor()`, `minimal_separator()`

Examples

```
cg <- caugi(
  C %-->% X,
  X %-->% F,
  X %-->% D,
  A %-->% X,
  A %-->% K,
  K %-->% Y,
  D %-->% Y,
  D %-->% G,
  Y %-->% H,
  class = "DAG"
)

adjustment_set(cg, "X", "Y", type = "parents") # C, A
adjustment_set(cg, "X", "Y", type = "backdoor") # C, A
adjustment_set(cg, "X", "Y", type = "optimal") # K
```

`aid` *Adjustment Identification Distance*

Description

Compute the Adjustment Identification Distance (AID) between two graphs using the `gadjid` Rust package.

Usage

```
aid(truth, guess, type = c("oset", "ancestor", "parent"), normalized = TRUE)
```

Arguments

`truth`	A caugi object.
`guess`	A caugi object.
`type`	A character string specifying the type of AID to compute. Options are `"oset"` (default), `"ancestor"`, and `"parent"`.
`normalized`	Logical; if `TRUE`, returns the normalized AID. If `FALSE`, returns the count.

Value

A numeric representing the AID between the two graphs, if `normalized = TRUE`, or an integer count if `normalized = FALSE`.

See Also

Other metrics: `hd()`, `shd()`

Examples

```
set.seed(1)
truth <- generate_graph(n = 100, m = 200, class = "DAG")
guess <- generate_graph(n = 100, m = 200, class = "DAG")
aid(truth, guess) # 0.0187
```

---

all_adjustment_sets_admg

Get all valid adjustment sets in an ADMG

Description

Enumerates all valid adjustment sets for estimating the causal effect of `X` on `Y` in an ADMG, up to a specified maximum size.

Usage

```
all_adjustment_sets_admg(
  cg,
  X = NULL,
  Y = NULL,
  X_index = NULL,
  Y_index = NULL,
  minimal = TRUE,
  max_size = 3L
)
```

Arguments

`cg`	A `caugi` object of class ADMG.
`X, Y`	Node names (can be vectors for multiple treatments/outcomes).
`X_index, Y_index`	Optional 1-based indices.
`minimal`	Logical; if `TRUE` (default), only minimal sets are returned.
`max_size`	Integer; maximum size of sets to consider (default 3).

Value

A list of character vectors, each a valid adjustment set (possibly empty list if none exist).

See Also

Other adjustment: `adjustment_set()`, `all_backdoor_sets()`, `d_separated()`, `is_valid_adjustment_admg()`, `is_valid_backdoor()`, `minimal_separator()`

Examples

```
cg <- caugi(
  L %-->% X,
  X %-->% Y,
  L %-->% Y,
  M %-->% Y,
  class = "ADMG"
)

all_adjustment_sets_admg(cg, X = "X", Y = "Y", minimal = TRUE)
# Returns {L} as minimal adjustment set
```

`all_backdoor_sets` *Get all backdoor sets up to a certain size.*

Description

This function returns the backdoor sets up to size `max_size`, which per default is set to 10.

Usage

```
all_backdoor_sets(
  cg,
  X = NULL,
  Y = NULL,
  X_index = NULL,
  Y_index = NULL,
  minimal = TRUE,
  max_size = 3L
)
```

Arguments

`cg`	A `caugi`.
`X, Y`	Single node name.
`X_index, Y_index`	Optional 1-based indices (exclusive with name args).
`minimal`	Logical; if `TRUE` (default), only minimal sets are returned.
`max_size`	Integer; maximum size of sets to consider (default 3).

Value

A list of character vectors, each an adjustment set (possibly empty).

See Also

Other adjustment: `adjustment_set()`, `all_adjustment_sets_admg()`, `d_separated()`, `is_valid_adjustment_admg()`, `is_valid_backdoor()`, `minimal_separator()`

Examples

```
cg <- caugi(
  C %-->% X,
  X %-->% F,
  X %-->% D,
  A %-->% X,
  A %-->% K,
  K %-->% Y,
  D %-->% Y,
  D %-->% G,
  Y %-->% H,
  class = "DAG"
)

all_backdoor_sets(cg, X = "X", Y = "Y", max_size = 3L, minimal = FALSE)
#> [[1]]
#> [1] "A"
#>
#> [[2]]
#> [1] "K"
#>
#> [[3]]
#> [1] "C" "A"
#>
#> [[4]]
#> [1] "C" "K"
#>
#> [[5]]
#> [1] "A" "K"
#>
#> [[6]]
#> [1] "C" "A" "K"

all_backdoor_sets(cg, X = "X", Y = "Y", max_size = 3L, minimal = TRUE)
#> [[1]]
#> [1] "A"
#>
#> [[2]]
#> [1] "K"
```

`ancestors` *Get ancestors of nodes in a* `caugi`

Description

Get ancestors of nodes in a `caugi`

Usage

```
ancestors(
  cg,
  nodes = NULL,
  index = NULL,
  open = caugi_options("use_open_graph_definition")
)
```

Arguments

`cg`	A `caugi` object.
`nodes`	A character vector of node names.
`index`	A vector of node indexes.
`open`	Boolean. Determines how the graph is interpreted when retrieving ancestors. Default is taken from `caugi_options("use_open_graph_definition")`, which by default is TRUE.

Value

Either a character vector of node names (if a single node is requested) or a list of character vectors (if multiple nodes are requested).

See Also

Other queries: `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg <- caugi(
  A %-->% B,
  B %-->% C,
  class = "DAG"
)
ancestors(cg, "A") # NULL
ancestors(cg, "A", open = FALSE) # A
```

```
ancestors(cg, index = 2) # "A"
ancestors(cg, "B") # "A"
ancestors(cg, c("B", "C"))
#> $B
#> [1] "A"
#>
#> $C
#> [1] "A" "B"
```

`anteriors` *Get anteriors of nodes in a* `caugi`

Description

Get the anterior set of nodes in a graph. The anterior set (Richardson and Spirtes, 2002) includes all nodes reachable by following paths where every edge is either undirected or directed toward the target node.

For DAGs, the anterior set equals the ancestor set (since there are no undirected edges). For PDAGs, it includes both ancestors and nodes reachable via undirected edges.

Usage

```
anteriors(
  cg,
  nodes = NULL,
  index = NULL,
  open = caugi_options("use_open_graph_definition")
)
```

Arguments

`cg`	A `caugi` object of class DAG or PDAG.
`nodes`	A character vector of node names.
`index`	A vector of node indexes.
`open`	Boolean. Determines how the graph is interpreted when retrieving anteriors. Default is taken from `caugi_options("use_open_graph_definition")`, which by default is TRUE.

Value

Either a character vector of node names (if a single node is requested) or a list of character vectors (if multiple nodes are requested).

References

Richardson, T. and Spirtes, P. (2002). Ancestral graph Markov models. *The Annals of Statistics*, 30(4):962-1030.

See Also

Other queries: `ancestors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
# PDAG example with directed and undirected edges
cg <- caugi(
  A %-->% B %---% C,
  B %-->% D,
  class = "PDAG"
)
anteriors(cg, "A") # NULL (no anteriors)
anteriors(cg, "A", open = FALSE) # A
anteriors(cg, "C") # A, B
anteriors(cg, "D") # A, B, C

# For DAGs, anteriors equals ancestors
cg_dag <- caugi(
  A %-->% B %-->% C,
  class = "DAG"
)
anteriors(cg_dag, "C") # A, B
```

`as_adjacency` *Convert a caugi to an adjacency matrix*

Description

Does not take other edge types than the one found in a PDAG.

Usage

```
as_adjacency(x)
```

Arguments

`x`	A `caugi` object.

Value

An integer 0/1 adjacency matrix with row/col names.

See Also

Other conversions: `as_bnlearn()`, `as_caugi()`, `as_dagitty()`, `as_igraph()`

Examples

```
cg <- caugi(
  A %-->% B,
  class = "DAG"
)
adj <- as_adjacency(cg)
```

`as_bnlearn` *Convert a caugi to a bnlearn network*

Description

Convert a caugi to a bnlearn network

Usage

```
as_bnlearn(x)
```

Arguments

`x`	A `caugi` object.

Value

A `bnlearn` DAG.

See Also

Other conversions: `as_adjacency()`, `as_caugi()`, `as_dagitty()`, `as_igraph()`

Examples

```
cg <- caugi(
  A %-->% B,
  class = "DAG"
)
g_bn <- as_bnlearn(cg)
```

`as_caugi` *Convert to a* `caugi`

Description

Convert an object to a `caugi`. The object can be a `graphNEL`, `matrix`, `tidygraph`, `daggity`, `bn`, or `igraph`.

Usage

```
as_caugi(
  x,
  class = c("DAG", "PDAG", "MPDAG", "ADMG", "PAG", "UNKNOWN"),
  simple = TRUE,
  collapse = FALSE,
  collapse_to = "---",
  ...
)
```

Arguments

`x`	An object to convert to a `caugi`.
`class`	"DAG", "PDAG", "MPDAG", "ADMG", "PAG", or "UNKNOWN". "PAG" is only supported for integer coded matrices. "ADMG" is for Acyclic Directed Mixed Graphs (with `-->` and `<->` edges).
`simple`	logical. If `TRUE` (default) the graph will be simple (no multiple edges or self-loops).
`collapse`	logical. If `TRUE` collapse mutual directed edges to undirected edges. Default is `FALSE`.
`collapse_to`	Character string to use as the edge glyph when collapsing. Should be a registered symmetrical edge glyph. Default is `"---"`.
`...`	Additional arguments passed to specific methods.

Details

For matrices, `as_caugi` assumes that the rows are the `from` nodes and the columns are the `to` nodes. Thus, for a graph, G: A –> B, we would have that `G["A", "B"] == 1` and `G["B", "A"] == 0`. For PAGs, the integer codes are as follows (as used in `pcalg`):

- 0: no edge
- 1: circle (e.g., `A o-o B` or `A o-- B`)
- 2: arrowhead (e.g., `A --> B` or `A o-> B`)
- 3: tail (e.g., `A o-- B` or `A --- B`)

Value

A `caugi` object.

See Also

Other conversions: `as_adjacency()`, `as_bnlearn()`, `as_dagitty()`, `as_igraph()`

Examples

```
# igraph
ig <- igraph::graph_from_literal(A - +B, B - +C)
cg_ig <- as_caugi(ig, class = "DAG")

# graphNEL
gn <- graph::graphNEL(nodes = c("A", "B", "C"), edgemode = "directed")
gn <- graph::addEdge("A", "B", gn)
gn <- graph::addEdge("B", "C", gn)
cg_gn <- as_caugi(gn, class = "DAG")

# adjacency matrix
m <- matrix(0L, 3, 3, dimnames = list(LETTERS[1:3], LETTERS[1:3]))
m["A", "B"] <- 1L
m["B", "C"] <- 1L
cg_adj <- as_caugi(m, class = "DAG")

# bnlearn
bn <- bnlearn::model2network("[A][B|A][C|B]")
cg_bn <- as_caugi(bn, class = "DAG")

# dagitty
dg <- dagitty::dagitty("dag {
 A -> B
 B -> C
 }")
cg_dg <- as_caugi(dg, class = "DAG")

cg <- caugi(A %-->% B %-->% C, class = "DAG")

# check that all nodes are equal in all graph objects
for (cg_converted in list(cg_ig, cg_gn, cg_adj, cg_bn, cg_dg)) {
  stopifnot(identical(nodes(cg), nodes(cg_converted)))
  stopifnot(identical(edges(cg), edges(cg_converted)))
}

# collapse mutual edges
ig2 <- igraph::graph_from_literal(A - +B, B - +A, C - +D)
cg2 <- as_caugi(ig2, class = "PDAG", collapse = TRUE, collapse_to = "---")

# coded integer matrix for PAGs (pcalg style)
nm <- c("A", "B", "C", "D")
M <- matrix(0L, 4, 4, dimnames = list(nm, nm))
```

```
# A --> B
M["A", "B"] <- 2L # mark at B end
M["B", "A"] <- 3L # mark at A end

# A --- C
M["A", "C"] <- 3L
M["C", "A"] <- 3L

# B o-> C
M["B", "C"] <- 2L
M["C", "B"] <- 1L

# C o-o D
M["C", "D"] <- 1L
M["D", "C"] <- 1L

cg <- as_caugi(M, class = "PAG")
```

`as_dagitty` *Convert a caugi to a dagitty graph*

Description

Convert a caugi to a dagitty graph

Usage

```
as_dagitty(x)
```

Arguments

`x`	A `caugi` object.

Value

A `dagitty` object.

See Also

Other conversions: `as_adjacency()`, `as_bnlearn()`, `as_caugi()`, `as_igraph()`

Examples

```
cg <- caugi(
  A %-->% B,
  class = "DAG"
)
g_dg <- as_dagitty(cg)
```

`as_igraph` *Convert a caugi to an igraph object*

Description

Convert a caugi to an igraph object

Usage

```
as_igraph(x, ...)
```

Arguments

`x`	A `caugi` object.
`...`	Additional arguments passed to `igraph::graph_from_data_frame()`.

Value

An `igraph` object representing the same graph structure.

See Also

Other conversions: `as_adjacency()`, `as_bnlearn()`, `as_caugi()`, `as_dagitty()`

Examples

```
cg <- caugi(
  A %-->% B,
  class = "DAG"
)
ig <- as_igraph(cg)
```

`build` *Build the `caugi` graph*

Description

Forces lazy compilation of the `caugi` graph without running a specific query. Useful to pre-initialize the graph.

Usage

```
build(cg)
```

Arguments

`cg`	A `caugi` object.

Value

The input `caugi` object (invisibly), with its graph built.

See Also

Other verbs: `caugi_verbs`

Examples

```
cg <- caugi(A %-->% B, class = "DAG")
build(cg) # initialize graph without querying
```

`caugi` *Create a* `caugi` *from edge expressions.*

Description

Create a `caugi` from a series of edge expressions using infix operators. Nodes can be specified as symbols, strings, or numbers.

The following edge operators are supported by default:

- `%-->%` for directed edges (A –> B)
- `%---%` for undirected edges (A — B)
- `%<->%` for bidirected edges (A <-> B)
- `%o->%` for partially directed edges (A o-> B)
- `%--o%` for partially undirected edges (A –o B)
- `%o-o%` for partial edges (A o-o B)

You can register additional edge types using `register_caugi_edge()`.

Usage

```
caugi(
  ...,
  from = NULL,
  edge = NULL,
  to = NULL,
  nodes = NULL,
  edges_df = NULL,
  simple = TRUE,
  build = NULL,
  class = c("AUTO", "DAG", "UG", "PDAG", "MPDAG", "ADMG", "AG", "UNKNOWN"),
```

```
  state = NULL,
  .session = NULL
)
```

Arguments

`...`	Edge expressions using the supported infix operators, or nodes given by symbols or strings. Multiple edges can be combined using `+`: `A --> B + C`, indicating an edge from `A` to both `B` and `C`. Nodes can also be grouped using `c(...)` or parentheses.
`from`	Character vector of source node names. Optional; mutually exclusive with `...`.
`edge`	Character vector of edge types. Optional; mutually exclusive with `...`.
`to`	Character vector of target node names. Optional; mutually exclusive with `...`.
`nodes`	Character vector of node names to declare as isolated nodes. An optional, but recommended, option is to provide all node names in the graph, including those that appear in edges. If `nodes` is provided, the order of nodes in the graph will follow the order in `nodes`.
`edges_df`	Optional data.frame or data.table with columns `from`, `edge`, and `to` to specify edges. Mutually exclusive with `...` and `from`, `edge`, `to`. Can be used to create graphs using `edges(cg)` from another `caugi` object, `cg`.
`simple`	Logical; if `TRUE` (default), the graph is a simple graph, and the function will throw an error if the input contains parallel edges or self-loops.
`build`	DEPRECATED. The graph is always built lazily, so this argument is ignored. Can use `build()` to force lazy compilation if desired.
`class`	Character; one of `"AUTO"`, `"DAG"`, `"UG"`, `"PDAG"`, `"MPDAG"`, `"ADMG"`, `"AG"`, or `"UNKNOWN"`. `"AUTO"` will automatically pick the appropriate class based on the first match in the order of `"DAG"`, `"UG"`, `"MPDAG"`, `"PDAG"`, `"ADMG"`, and `"AG"`. It will default to `"UNKNOWN"` if no match is found.
`state`	DEPRECATED. Replaced by `.session`.
`.session`	For internal use. Build a graph by supplying a pre-constructed session pointer from Rust.

Value

A `caugi` S7 object containing the nodes, edges, and a pointer to the underlying Rust graph structure.

Examples

```
# create a simple DAG (using NSE)
cg <- caugi(
  A %-->% B + C,
  B %-->% D,
  class = "DAG"
)

# create a PDAG with undirected edges (using NSE)
cg2 <- caugi(
```

```
  A %-->% B + C,
  B %---% D,
  E, # no neighbors for this node
  class = "PDAG"
)

# create a DAG (using SE)
cg3 <- caugi(
  from = c("A", "A", "B"),
  edge = c("-->", "-->", "-->"),
  to = c("B", "C", "D"),
  nodes = c("A", "B", "C", "D", "E"),
  class = "DAG"
)

# create a non-simple graph
cg4 <- caugi(
  A %-->% B,
  B %-->% A,
  class = "UNKNOWN",
  simple = FALSE
)

cg4@simple # FALSE
cg4@graph_class # "UNKNOWN"
```

`caugi_default_options` *Default options for caugi*

Description

Returns the default options for the caugi package. Useful for resetting options to their original state.

Usage

```
caugi_default_options()
```

Value

A list of default options for caugi.

See Also

`caugi_options()` for setting and getting options

Examples

```
# Get defaults
caugi_default_options()

# Reset to defaults
caugi_options(caugi_default_options())
```

`caugi_deserialize` *Deserialize caugi Graph from JSON String*

Description

Converts a JSON string in the native caugi format back to a caugi graph. This is a lower-level function; consider using `read_caugi()` for reading from files.

Usage

```
caugi_deserialize(json, lazy)
```

Arguments

`json`	Character string containing the JSON representation.
`lazy`	DEPRECATED, no longer necessary. The graph is always built lazily, so this argument is ignored.

Value

A `caugi` object.

See Also

Other export: `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-dot`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

Examples

```
cg <- caugi(A %-->% B, class = "DAG")
json <- caugi_serialize(cg)
cg2 <- caugi_deserialize(json)
```

`caugi_dot` *S7 Class for DOT Export*

Description

An S7 object that wraps a DOT format string for displaying caugi graphs. When printed interactively, displays the DOT string cleanly.

Usage

```
caugi_dot(content)
```

Arguments

`content`	A character string containing the DOT format graph.

See Also

Other export: `caugi_deserialize()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-dot`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

`caugi_export` *S7 Base Class for Caugi Exports*

Description

A base class for all caugi export formats. Provides common structure and behavior for different export formats (DOT, GraphML, etc.).

Usage

```
caugi_export(content = character(0), format = character(0))
```

Arguments

`content`	A character string containing the exported graph.
`format`	A character string indicating the export format.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-dot`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

`caugi_graphml` *S7 Class for GraphML Export*

Description

An S7 object that wraps a GraphML format string for caugi graphs.

Usage

```
caugi_graphml(content)
```

Arguments

`content`	A character string containing the GraphML format graph.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-dot`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

`caugi_layout` *Compute Graph Layout*

Description

Computes node coordinates for graph visualization using specified layout algorithm. If the graph has not been built yet, it will be built automatically before computing the layout.

Usage

```
caugi_layout(
  x,
  method = c("auto", "sugiyama", "fruchterman-reingold", "kamada-kawai", "bipartite",
    "tiered", "circle"),
  packing_ratio = 1.618034,
  ...
)
```

Arguments

`x`	A `caugi` object. Must contain only directed edges for Sugiyama layout.
`method`	Character string specifying the layout method. Options: • `"auto"`: Automatically choose the best layout (default). Selection order: 1. If `tiers` is provided, uses `"tiered"` 2. If `partition` is provided, uses `"bipartite"` 3. If graph has only directed edges, uses `"sugiyama"` 4. Otherwise, uses `"fruchterman-reingold"` • `"sugiyama"`: Hierarchical layout for DAGs (requires only directed edges) • `"fruchterman-reingold"`: Fast spring-electrical layout (works with all edge types) • `"kamada-kawai"`: High-quality stress minimization (works with all edge types) • `"bipartite"`: Bipartite layout (requires `partition` parameter) • `"tiered"`: Multi-tier layout (requires `tiers` parameter) • `"circle"`: Place nodes evenly along a circle
`packing_ratio`	Aspect ratio for packing disconnected components (width/height). Default is the golden ratio (1.618) which works well with widescreen displays. Use `1.0` for square grid, `2.0` for wider layouts, `0.5` for taller layouts, `Inf` for single row, or `0.0` for single column.
`...`	Additional arguments passed to the specific layout function. For bipartite layouts, use `partition` (logical vector) and `orientation` (`"columns"` or `"rows"`). For tiered layouts, use `tiers` (list, named vector, or data.frame) and `orientation` (`"rows"` or `"columns"`).

Value

A `data.frame` with columns `name`, `x`, and `y` containing node names and their coordinates.

Layout Algorithms

Sugiyama (Hierarchical Layout)

Optimized for directed acyclic graphs (DAGs). Places nodes in layers to emphasize hierarchical structure and causal flow from top to bottom. Edges are routed to minimize crossings. Best for visualizing clear cause-effect relationships. Only works with directed edges.

Fruchterman-Reingold (Spring-Electrical)

Fast force-directed layout using a spring-electrical model. Treats edges as springs and nodes as electrically charged particles. Produces organic, symmetric layouts with uniform edge lengths. Good for general-purpose visualization and works with all edge types. Results are deterministic.

Kamada-Kawai (Stress Minimization)

High-quality force-directed layout that minimizes "stress" by making Euclidean distances proportional to graph-theoretic distances. Better preserves the global structure and path lengths compared to Fruchterman-Reingold. Ideal for publication-quality visualizations where accurate distance representation matters. Works with all edge types and produces deterministic results.

Circle

Places nodes evenly along the perimeter of a circle. The first node is at the top and subsequent nodes proceed counter-clockwise. Useful for small graphs, cycle visualization, and as a deterministic fallback. Works with all edge types and ignores edge structure.

Source

Fruchterman, T. M. J., & Reingold, E. M. (1991). Graph drawing by force-directed placement. Software: Practice and Experience, 21(11), 1129-1164. doi:10.1002/spe.4380211102

Kamada, T., & Kawai, S. (1989). An algorithm for drawing general undirected graphs. Information Processing Letters, 31(1), 7-15. doi:10.1016/00200190(89)901026

Sugiyama, K., Tagawa, S., & Toda, M. (1981). Methods for visual understanding of hierarchical system structures. IEEE Transactions on Systems, Man, and Cybernetics, 11(2), 109-125. doi:10.1109/TSMC.1981.4308636

See Also

Other plotting: `add-caugi_plot-caugi_plot`, `caugi_layout_bipartite()`, `caugi_layout_circle()`, `caugi_layout_fruchterman_reingold()`, `caugi_layout_kamada_kawai()`, `caugi_layout_sugiyama()`, `caugi_layout_tiered()`, `caugi_plot()`, `divide-caugi_plot-caugi_plot`, `plot()`

Examples

```
cg <- caugi(
  A %-->% B + C,
  B %-->% D,
  C %-->% D,
  class = "DAG"
)

# Default: auto-selects best layout
layout <- caugi_layout(cg)

# Auto-selects tiered when tiers provided
cg_tiered <- caugi(X1 %-->% M1, X2 %-->% M2, M1 %-->% Y, M2 %-->% Y)
tiers <- list(c("X1", "X2"), c("M1", "M2"), "Y")
layout_auto <- caugi_layout(cg_tiered, tiers = tiers) # Uses "tiered"

# Explicitly use hierarchical layout
layout_sug <- caugi_layout(cg, method = "sugiyama")

# Use force-directed for organic appearance
layout_fr <- caugi_layout(cg, method = "fruchterman-reingold")

# Use stress minimization for publication quality
layout_kk <- caugi_layout(cg, method = "kamada-kawai")

# Place nodes on a circle
layout_circle <- caugi_layout(cg, method = "circle")
```

```
# Bipartite layout with auto-detected partition
cg_bp <- caugi(A %-->% X, A %-->% Y, B %-->% X, B %-->% Y)
layout_bp_rows <- caugi_layout(
  cg_bp,
  method = "bipartite",
  orientation = "rows"
)

# Explicit partition
partition <- c(TRUE, TRUE, FALSE, FALSE)
layout_bp_cols <- caugi_layout(
  cg_bp,
  method = "bipartite",
  partition = partition,
  orientation = "columns"
)

# Tiered layout with three tiers
cg_tiered <- caugi(
  X1 %-->% M1 + M2,
  X2 %-->% M1 + M2,
  M1 %-->% Y,
  M2 %-->% Y
)
tiers <- list(c("X1", "X2"), c("M1", "M2"), "Y")
layout_tiered <- caugi_layout(
  cg_tiered,
  method = "tiered",
  tiers = tiers,
  orientation = "rows"
)
```

`caugi_layout_bipartite` *Bipartite Graph Layout*

Description

Computes node coordinates for bipartite graphs, placing nodes in two parallel lines (rows or columns) based on a partition. If the graph has not been built yet, it will be built automatically before computing the layout.

Usage

```
caugi_layout_bipartite(x, partition = NULL, orientation = c("columns", "rows"))
```

Arguments

`x`	A `caugi` object.
`partition`	Optional logical vector indicating node partitions. Nodes with `TRUE` are placed in one partition and nodes with `FALSE` in the other. Length must equal the number of nodes. Both partitions must be non-empty. If `NULL` (default), attempts to detect bipartite structure automatically by assigning nodes without incoming edges to one partition and others to the second partition.
`orientation`	Character string specifying the layout orientation: • `"columns"`: Two vertical columns. First partition on right (x=1), second partition on left (x=0). • `"rows"`: Two horizontal rows. First partition on top (y=1), second partition on bottom (y=0).

Value

A `data.frame` with columns `name`, `x`, and `y` containing node names and their coordinates.

See Also

Other plotting: `add-caugi_plot-caugi_plot`, `caugi_layout()`, `caugi_layout_circle()`, `caugi_layout_fruchterman` `caugi_layout_kamada_kawai()`, `caugi_layout_sugiyama()`, `caugi_layout_tiered()`, `caugi_plot()`, `divide-caugi_plot-caugi_plot`, `plot()`

Examples

```
# Create a bipartite graph (causes -> effects)
cg <- caugi(A %-->% X, A %-->% Y, B %-->% X, B %-->% Y)
partition <- c(TRUE, TRUE, FALSE, FALSE) # A, B = causes, X, Y = effects

# Two horizontal rows (causes on top)
layout_rows <- caugi_layout_bipartite(cg, partition, orientation = "rows")

# Two vertical columns (causes on right)
layout_cols <- caugi_layout_bipartite(cg, partition, orientation = "columns")
```

`caugi_layout_circle` *Circle Layout*

Description

Computes node coordinates by placing nodes evenly along the perimeter of a circle. The first node is placed at the top of the circle, and subsequent nodes proceed counter-clockwise. Edge structure is ignored. Works with all edge types and produces deterministic results.

Usage

```
caugi_layout_circle(x, ...)
```

Arguments

`x`	A `caugi` object.
`...`	Ignored. For future extensibility.

Value

A `data.frame` with columns `name`, `x`, and `y` containing node names and their coordinates.

See Also

Other plotting: `add-caugi_plot-caugi_plot`, `caugi_layout()`, `caugi_layout_bipartite()`, `caugi_layout_fruchterman_reingold()`, `caugi_layout_kamada_kawai()`, `caugi_layout_sugiyama()`, `caugi_layout_tiered()`, `caugi_plot()`, `divide-caugi_plot-caugi_plot`, `plot()`

Examples

```
cg <- caugi(
  A %-->% B,
  B %-->% C,
  C %-->% D,
  D %-->% A
)
layout <- caugi_layout_circle(cg)
```

`caugi_layout_fruchterman_reingold`

Fruchterman-Reingold Force-Directed Layout

Description

Computes node coordinates using the Fruchterman-Reingold force-directed layout algorithm. Fast spring-electrical model that treats edges as springs and nodes as electrically charged particles. Produces organic, symmetric layouts with uniform edge lengths. Works with all edge types and produces deterministic results.

Usage

```
caugi_layout_fruchterman_reingold(x, packing_ratio = 1.618034, ...)
```

Arguments

`x`	A caugi object.
`packing_ratio`	Aspect ratio for packing disconnected components (width/height). Default is the golden ratio (1.618) which works well with widescreen displays. Use `1.0` for square grid, `2.0` for wider layouts, `0.5` for taller layouts, `Inf` for single row, or `0.0` for single column.
`...`	Ignored. For future extensibility.

Value

A `data.frame` with columns `name`, `x`, and `y` containing node names and their coordinates.

Source

Fruchterman, T. M. J., & Reingold, E. M. (1991). Graph drawing by force-directed placement. Software: Practice and Experience, 21(11), 1129-1164. doi:10.1002/spe.4380211102

See Also

Other plotting: `add-caugi_plot-caugi_plot`, `caugi_layout()`, `caugi_layout_bipartite()`, `caugi_layout_circle()`, `caugi_layout_kamada_kawai()`, `caugi_layout_sugiyama()`, `caugi_layout_tiered()`, `caugi_plot()`, `divide-caugi_plot-caugi_plot`, `plot()`

Examples

```
cg <- caugi(
  A %-->% B,
  B %<->% C,
  C %-->% D
)
layout <- caugi_layout_fruchterman_reingold(cg)
```

`caugi_layout_kamada_kawai`

Kamada-Kawai Stress Minimization Layout

Description

Computes node coordinates using the Kamada-Kawai stress minimization algorithm. High-quality force-directed layout that minimizes "stress" by making Euclidean distances proportional to graph-theoretic distances. Better preserves global structure and path lengths compared to Fruchterman-Reingold. Ideal for publication-quality visualizations. Works with all edge types and produces deterministic results.

Usage

```
caugi_layout_kamada_kawai(x, packing_ratio = 1.618034, ...)
```

Arguments

`x`	A `caugi` object.
`packing_ratio`	Aspect ratio for packing disconnected components (width/height). Default is the golden ratio (1.618) which works well with widescreen displays. Use `1.0` for square grid, `2.0` for wider layouts, `0.5` for taller layouts, `Inf` for single row, or `0.0` for single column.
`...`	Ignored. For future extensibility.

Value

A `data.frame` with columns `name`, `x`, and `y` containing node names and their coordinates.

Source

Kamada, T., & Kawai, S. (1989). An algorithm for drawing general undirected graphs. Information Processing Letters, 31(1), 7-15. doi:10.1016/00200190(89)901026

See Also

Other plotting: `add-caugi_plot-caugi_plot`, `caugi_layout()`, `caugi_layout_bipartite()`, `caugi_layout_circle()`, `caugi_layout_fruchterman_reingold()`, `caugi_layout_sugiyama()`, `caugi_layout_tiered()`, `caugi_plot()`, `divide-caugi_plot-caugi_plot`, `plot()`

Examples

```
cg <- caugi(
  A %-->% B,
  B %<->% C,
  C %-->% D
)
layout <- caugi_layout_kamada_kawai(cg)
```

`caugi_layout_sugiyama` *Sugiyama Hierarchical Layout*

Description

Computes node coordinates using the Sugiyama hierarchical layout algorithm. Optimized for directed acyclic graphs (DAGs), placing nodes in layers to emphasize hierarchical structure and causal flow from top to bottom.

Usage

```
caugi_layout_sugiyama(x, packing_ratio = 1.618034, ...)
```

Arguments

`x`	A `caugi` object. Must contain only directed edges.
`packing_ratio`	Aspect ratio for packing disconnected components (width/height). Default is the golden ratio (1.618) which works well with widescreen displays. Use `1.0` for square grid, `2.0` for wider layouts, `0.5` for taller layouts, `Inf` for single row, or `0.0` for single column.
`...`	Ignored. For future extensibility.

Value

A `data.frame` with columns `name`, `x`, and `y` containing node names and their coordinates.

Source

Sugiyama, K., Tagawa, S., & Toda, M. (1981). Methods for visual understanding of hierarchical system structures. IEEE Transactions on Systems, Man, and Cybernetics, 11(2), 109-125. doi:10.1109/TSMC.1981.4308636

See Also

Other plotting: `add-caugi_plot-caugi_plot`, `caugi_layout()`, `caugi_layout_bipartite()`, `caugi_layout_circle()`, `caugi_layout_fruchterman_reingold()`, `caugi_layout_kamada_kawai()`, `caugi_layout_tiered()`, `caugi_plot()`, `divide-caugi_plot-caugi_plot`, `plot()`

Examples

```
cg <- caugi(A %-->% B + C, B %-->% D, C %-->% D, class = "DAG")
layout <- caugi_layout_sugiyama(cg)
```

`caugi_layout_tiered` *Tiered Graph Layout*

Description

Computes node coordinates for graphs with multiple tiers (layers), placing nodes in parallel rows or columns based on tier assignments. If the graph has not been built yet, it will be built automatically before computing the layout.

Usage

```
caugi_layout_tiered(x, tiers, orientation = c("columns", "rows"))
```

Arguments

`x`	A `caugi` object.
`tiers`	Tier assignments specifying which tier each node belongs to. Can be provided in multiple formats: • **Named list**: List where each element is a character vector of node names belonging to that tier. Element names are ignored; tier order is determined by list order (first element = tier 0, etc.). • **Named numeric vector**: Vector where names are node names and values are tier indices (starting from 0 or 1). • **Data.frame**: Must contain columns `name` (node names) and `tier` (tier indices).

	All nodes must be assigned to a tier, all tiers must be non-empty, and tier indices must be consecutive starting from 0 or 1.
`orientation`	Character string specifying the layout orientation: • `"columns"`: Vertical tiers. First tier at left (x=0), subsequent tiers to the right, last tier at right (x=1). • `"rows"`: Horizontal tiers. First tier at top (y=1), subsequent tiers below, last tier at bottom (y=0).

Value

A `data.frame` with columns `name`, `x`, `y`, and `tier` containing node names, their coordinates, and tier assignments (0-indexed). The returned data.frame also has an `orientation` attribute storing the orientation used. When passed to `plot()`, tier information is automatically extracted, so you don't need to specify `tiers` again.

See Also

Other plotting: `add-caugi_plot-caugi_plot`, `caugi_layout()`, `caugi_layout_bipartite()`, `caugi_layout_circle()`, `caugi_layout_fruchterman_reingold()`, `caugi_layout_kamada_kawai()`, `caugi_layout_sugiyama()`, `caugi_plot()`, `divide-caugi_plot-caugi_plot`, `plot()`

Examples

```
# Create a three-tier causal graph (exposures -> mediators -> outcome)
cg <- caugi(
  X1 %-->% M1 + M2,
  X2 %-->% M1 + M2,
  M1 %-->% Y,
  M2 %-->% Y
)

# Option 1: Named list (tier names are just labels)
tiers <- list(
  exposures = c("X1", "X2"),
  mediators = c("M1", "M2"),
  outcome = "Y"
)
layout_rows <- caugi_layout_tiered(cg, tiers, orientation = "rows")

# Option 2: Named numeric vector (0-indexed or 1-indexed both work)
tiers <- c(X1 = 1, X2 = 1, M1 = 2, M2 = 2, Y = 3)
layout_cols <- caugi_layout_tiered(cg, tiers, orientation = "columns")

# Option 3: Data.frame
tiers <- data.frame(
  name = c("X1", "X2", "M1", "M2", "Y"),
  tier = c(1, 1, 2, 2, 3)
)
layout <- caugi_layout_tiered(cg, tiers, orientation = "rows")

# The layout includes tier information, so plot() works without passing tiers
```

```
plot(cg, layout = layout)
```

---

caugi_mermaid *S7 Class for Mermaid Export*

Description

An S7 object that wraps a Mermaid format string for displaying caugi graphs. When printed interactively, displays the Mermaid string cleanly.

Usage

```
caugi_mermaid(content)
```

Arguments

`content`	A character string containing the Mermaid format graph.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-dot`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

---

caugi_options *Get or set global options for caugi*

Description

Configure global defaults for caugi, including plot composition spacing and default visual styles for nodes, edges, labels, and titles.

Usage

```
caugi_options(...)
```

Arguments

`...`	Named values to update options with, or unnamed option names to retrieve. Multiple unnamed arguments drill down through nested options. To query all options, call without arguments. Attempting to access a non-existent option will raise an error.

Details

The `use_open_graph_definition` option (`TRUE`/`FALSE`, default `TRUE`) controls how graph queries interpret reachability relations. When `TRUE` (open definition), queries such as `ancestors()`, `descendants()`, `posteriors()`,and `anteriors()` exclude the queried node itself. When `FALSE` (closed definition), the queried node is included in the results.

The plot options are nested under the `plot` key:

- `spacing`: A `grid::unit()` controlling space between composed plots (default: `grid::unit(1, "lines")`)
- `node_style`: List of default node appearance parameters:
  - `fill`: Fill color (default: `"lightgrey"`)
  - `padding`: Padding around labels in mm (default: `2`)
  - `size`: Size multiplier (default: `1`)
- `edge_style`: List of default edge appearance parameters:
  - `arrow_size`: Arrow size in mm (default: `3`)
  - `circle_size`: Radius of endpoint circles for partial edges in mm (default: `1.5`)
  - `fill`: Arrow/line color (default: `"black"`)
- `label_style`: List of label text parameters (see `grid::gpar()`)
- `title_style`: List of title text parameters:
  - `col`: Text color (default: `"black"`)
  - `fontface`: Font face (default: `"bold"`)
  - `fontsize`: Font size in pts (default: `14.4`)

Options set via `caugi_options()` serve as global defaults that can be overridden by arguments to `plot()`.

Value

When setting, returns (invisibly) the previous values for the updated options. When getting (no arguments or unnamed character vector), returns the requested options.

See Also

`plot()` for per-plot style arguments, `grid::gpar()` for available graphical parameters

Examples

```
# Query all options
caugi_options()

# Use closed graph definition
caugi_options(use_open_graph_definition = FALSE)

# Query specific option
caugi_options("plot")

# Query nested option
```

```
caugi_options("plot", "tier_style")
caugi_options("plot", "node_style", "fill")

# Set plot spacing
caugi_options(plot = list(spacing = grid::unit(2, "lines")))

# Set default node style
caugi_options(plot = list(
  node_style = list(fill = "lightblue", padding = 3)
))

# Set multiple options at once
caugi_options(plot = list(
  spacing = grid::unit(1.5, "lines"),
  node_style = list(fill = "lightblue", padding = 3),
  edge_style = list(arrow_size = 4, fill = "darkgray"),
  title_style = list(col = "blue", fontsize = 16)
))

# Reset to defaults
caugi_options(caugi_default_options())
```

`caugi_plot` *S7 Class for caugi Plot*

Description

An S7 object that wraps a grid gTree for displaying caugi graphs. Similar to ggplot objects, these are created by the plot method but not drawn until explicitly printed or plotted. This allows for returning plot objects from functions and controlling when/where they are displayed.

Usage

```
caugi_plot(grob = NULL)
```

Arguments

`grob`	A grid gTree representing the graph plot.

See Also

Other plotting: `add-caugi_plot-caugi_plot`, `caugi_layout()`, `caugi_layout_bipartite()`, `caugi_layout_circle()`, `caugi_layout_fruchterman_reingold()`, `caugi_layout_kamada_kawai()`, `caugi_layout_sugiyama()`, `caugi_layout_tiered()`, `divide-caugi_plot-caugi_plot`, `plot()`

`caugi_serialize` *Serialize caugi Graph to JSON String*

Description

Converts a caugi graph to a JSON string in the native caugi format. This is a lower-level function; consider using `write_caugi()` for writing to files.

Usage

```
caugi_serialize(x, comment = NULL, tags = NULL)
```

Arguments

`x`	A `caugi` object or an object coercible to `caugi`.
`comment`	Optional character string with a comment about the graph.
`tags`	Optional character vector of tags for categorizing the graph.

Value

A character string containing the JSON representation.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `export-classes`, `format-caugi`, `format-dot`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

Examples

```
cg <- caugi(A %-->% B, class = "DAG")
json <- caugi_serialize(cg)
cat(json)
```

`caugi_verbs` *Manipulate nodes and edges of a* `caugi`

Description

Add, remove, or and set nodes or edges to / from a `caugi` object. Edges can be specified using expressions with the infix operators. Alternatively, the edges to be added are specified using the `from`, `edge`, and `to` arguments.

Usage

```
add_edges(cg, ..., from = NULL, edge = NULL, to = NULL, inplace = FALSE)

remove_edges(cg, ..., from = NULL, edge = NULL, to = NULL, inplace = FALSE)

set_edges(cg, ..., from = NULL, edge = NULL, to = NULL, inplace = FALSE)

add_nodes(cg, ..., name = NULL, inplace = FALSE)

remove_nodes(cg, ..., name = NULL, inplace = FALSE)
```

Arguments

`cg`	A `caugi` object.
`...`	Expressions specifying edges to add using the infix operators, or nodes to add using unquoted names, vectors via `c()`, or + composition.
`from`	Character vector of source node names. Default is `NULL`.
`edge`	Character vector of edge types. Default is `NULL`.
`to`	Character vector of target node names. Default is `NULL`.
`inplace`	DEPRECATED This parameter is deprecated and will be ignored. Graphs are always modified via copy-on-write.
`name`	Character vector of node names. Default is `NULL`.

Details

Caugi graph verbs

Value

The updated `caugi`.

Functions

- `add_edges()`: Add edges.
- `remove_edges()`: Remove edges.
- `set_edges()`: Set edge type for given pair(s).
- `add_nodes()`: Add nodes.
- `remove_nodes()`: Remove nodes.

See Also

Other verbs: `build()`

Examples

```
# initialize empty graph and build slowly
cg <- caugi(class = "PDAG")

cg <- cg |>
  add_nodes(c("A", "B", "C", "D", "E")) |> # A, B, C, D, E
  add_edges(A %-->% B %-->% C) |> # A --> B --> C, D, E
  set_edges(B %---% C) # A --> B --- C, D, E

cg <- remove_edges(cg, B %---% C) |> # A --> B, C, D, E
  remove_nodes(c("C", "D", "E")) # A --> B

# Graphs are now built lazily when needed
parents(cg, "B") # triggers compilation
```

`children` *Get children of nodes in a* `caugi`

Description

Get children of nodes in a graph (nodes with directed edges pointing OUT from the target nodes). This is equivalent to `neighbors(cg, nodes, mode = "out")`.

Usage

```
children(cg, nodes = NULL, index = NULL)
```

Arguments

`cg`	A `caugi` object.
`nodes`	A character vector of node names.
`index`	A vector of node indexes.

Value

Either a character vector of node names (if a single node is requested) or a list of character vectors (if multiple nodes are requested).

See Also

Other queries: `ancestors()`, `anteriors()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg <- caugi(
  A %-->% B,
  B %-->% C,
  class = "DAG"
)
children(cg, "A") # "B"
children(cg, index = 2) # "C"
children(cg, "B") # "C"
children(cg, c("B", "C"))
#> $B
#> [1] "C"
#>
#> $C
#> NULL
```

`condition_marginalize` *Marginalize and/or condition on variables in an ancestral graph (AG)*

Description

Marginalize variables out of an AG, and/or condition on variables. Depending on the structure, it could produce a graph with directed, bidirected, and undirected edges.

Usage

```
condition_marginalize(cg, cond_vars = NULL, marg_vars = NULL)
```

Arguments

`cg`	A `caugi` ancestral graph of class `"AG"`.
`cond_vars`	Character vector of nodes to condition on.
`marg_vars`	Character vector of nodes to marginalize over.

Value

A `caugi` object of class `"AG"`.

References

Definition 4.2.1 in Thomas Richardson. Peter Spirtes. "Ancestral graph Markov models." Ann. Statist. 30 (4) 962 - 1030, August 2002. doi:10.1214/aos/1031689015

See Also

Other operations: `dag_from_pdag()`, `exogenize()`, `latent_project()`, `meek_closure()`, `moralize()`, `mutate_caugi()`, `normalize_latent_structure()`, `skeleton()`

Examples

```
mg <- caugi(
  U %-->% X + Y,
  A %-->% X,
  B %-->% Y,
  class = "DAG"
)

condition_marginalize(mg, marg_vars = "U") # ADMG
condition_marginalize(mg, cond_vars = "U") # DAG
```

`dag_from_pdag` *Extend a PDAG to a DAG using the Dor-Tarsi Algorithm*

Description

Given a Partially Directed Acyclic Graph (PDAG), this function attempts to extend it to a Directed Acyclic Graph (DAG) by orienting the undirected edges while preserving acyclicity and all existing directed edges. The procedure implements the Dor-Tarsi algorithm.

If the PDAG cannot be consistently extended to a DAG, the function will raise an error.

Usage

```
dag_from_pdag(PDAG)
```

Arguments

`PDAG`	A `caugi` object of class `"PDAG"`.

Value

A `caugi` object of class `"DAG"` representing a DAG extension of the input PDAG.

References

Dor, D., & Tarsi, M. (1992). "A simple algorithm to construct a consistent extension of a partially directed acyclic graph".

See Also

Other operations: `condition_marginalize()`, `exogenize()`, `latent_project()`, `meek_closure()`, `moralize()`, `mutate_caugi()`, `normalize_latent_structure()`, `skeleton()`

Examples

```
PDAG <- caugi(
  A %---% B,
  B %---% C,
  class = "PDAG"
)
DAG <- dag_from_pdag(PDAG)
edges(DAG)
```

`descendants` *Get descendants of nodes in a* `caugi`

Description

Get descendants of nodes in a `caugi`

Usage

```
descendants(
  cg,
  nodes = NULL,
  index = NULL,
  open = caugi_options("use_open_graph_definition")
)
```

Arguments

`cg`	A `caugi` object.
`nodes`	A character vector of node names.
`index`	A vector of node indexes.
`open`	Boolean. Determines how the graph is interpreted when retrieving descendants. Default is taken from `caugi_options("use_open_graph_definition")`, which by default is TRUE.

Value

Either a character vector of node names (if a single node is requested) or a list of character vectors (if multiple nodes are requested).

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg <- caugi(
  A %-->% B,
  B %-->% C,
  class = "DAG"
)
descendants(cg, "A") # "B" "C"
descendants(cg, "A", open = FALSE) # "A" "B" "C"
descendants(cg, index = 2) # "C"
descendants(cg, "B") # "C"
descendants(cg, c("B", "C"))
#> $B
#> [1] "C"
#>
#> $C
#> NULL
```

`districts` *Get districts (c-components) of an ADMG or AG*

Description

Get districts (c-components) for all nodes, or for selected nodes in an ADMG/AG. A district is a maximal set of nodes connected via bidirected edges. If both `nodes` and `index` are `NULL`, returns all districts in the graph.

Usage

```
districts(cg, nodes = NULL, index = NULL, all = NULL)
```

Arguments

`cg`	A `caugi` object of class ADMG or AG.
`nodes`	Optional character vector of node names. If supplied, returns district(s) containing these nodes.
`index`	Optional numeric vector of 1-based node indices. If supplied, returns district(s) containing these indices.
`all`	DEPRECATED (If `TRUE`, return all districts explicitly. Cannot be combined with `nodes` or `index`.)

Value

If all districts are requested: a list of character vectors, one per district. If `nodes`/`index` are supplied: either a character vector (single target) or a named list of character vectors (multiple targets).

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg <- caugi(
  A %-->% B,
  A %<->% C,
  D %<->% E,
  class = "ADMG"
)
districts(cg)
# Returns list with districts: {A, C}, {B}, {D, E}
districts(cg, nodes = "A") # Returns c("A", "C")
districts(cg, index = c(1, 4))
```

`divide-caugi_plot-caugi_plot`

Compose Plots Vertically

Description

Stack two plots vertically with configurable spacing. Compositions can be nested to create complex multi-plot layouts.

Arguments

`e1`	A `caugi_plot` object (top plot)
`e2`	A `caugi_plot` object (bottom plot)

Details

The spacing between plots is controlled by the global option `caugi_options()$plot$spacing`, which defaults to `grid::unit(1, "lines")`. Compositions can be nested arbitrarily:

- `p1 / p2` - two plots stacked vertically
- `p1 / p2 / p3` - three plots in a column
- `(p1 + p2) / p3` - two plots on top, one below

Value

A `caugi_plot` object containing the composed layout

See Also

`caugi_options()` for configuring spacing and default styles

Other plotting: `add-caugi_plot-caugi_plot`, `caugi_layout()`, `caugi_layout_bipartite()`, `caugi_layout_circle()`, `caugi_layout_fruchterman_reingold()`, `caugi_layout_kamada_kawai()`, `caugi_layout_sugiyama()`, `caugi_layout_tiered()`, `caugi_plot()`, `plot()`

Examples

```
cg1 <- caugi(A %-->% B, B %-->% C)
cg2 <- caugi(X %-->% Y, Y %-->% Z)

p1 <- plot(cg1, main = "Graph 1")
p2 <- plot(cg2, main = "Graph 2")

# Vertical composition
p1 / p2

# Mixed composition
(p1 + p2) / p1
```

`d_separated` *Are X and Y d-separated given Z?*

Description

Checks whether every node in `X` is d-separated from every node in `Y` given `Z` in a DAG.

Usage

```
d_separated(
  cg,
  X = NULL,
  Y = NULL,
  Z = NULL,
  X_index = NULL,
  Y_index = NULL,
  Z_index = NULL
)
```

Arguments

`cg`	A `caugi` object.
`X`, `Y`, `Z`	Character vectors of node names, or `NULL`. Use `*_index` to pass 1-based indices. If `Z` is `NULL` or missing, no nodes are conditioned on.
`X_index`, `Y_index`, `Z_index`	Optional numeric 1-based indices (exclusive with `X`,`Y`,`Z` respectively).

Value

`TRUE` if d-separated, `FALSE` otherwise.

See Also

Other adjustment: `adjustment_set()`, `all_adjustment_sets_admg()`, `all_backdoor_sets()`, `is_valid_adjustment_admg()`, `is_valid_backdoor()`, `minimal_separator()`

Examples

```
cg <- caugi(
  C %-->% X,
  X %-->% F,
  X %-->% D,
  A %-->% X,
  A %-->% K,
  K %-->% Y,
  D %-->% Y,
  D %-->% G,
  Y %-->% H,
  class = "DAG"
)

d_separated(cg, "X", "Y", Z = c("A", "D")) # TRUE
d_separated(cg, "X", "Y", Z = NULL) # FALSE
```

`edges` *Get edges of a* `caugi`.

Description

Get edges of a `caugi`.

Usage

```
edges(cg, ...)

E(cg, ...)
```

Arguments

`cg`	A `caugi` object.
`...`	Additional arguments (currently unused).

Value

A `data.table` with columns `from`, `edge`, and `to`.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg <- caugi(
  A %-->% B,
  B %-->% C,
  D,
  class = "DAG"
)
edges(cg) # returns the data.table with columns from, edge, to
```

`edge_types` *Get the edge types of a* `caugi`*.*

Description

Get the edge types of a `caugi`.

Usage

```
edge_types(cg)
```

Arguments

`cg`	A `caugi` object.

Value

A character vector of edge types.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg <- caugi(
  A %-->% B,
  B %--o% C,
  C %<->% D,
  D %---% E,
  A %o-o% E,
  class = "UNKNOWN"
)
edge_types(cg) # returns c("-->", "o-o", "--o", "<->", "---")
```

`exogenize` *Exogenize a graph*

Description

Exogenize a graph by removing all ingoing edges to the set of nodes specified (i.e., make the nodes exogenous), as well as joining the parents of the nodes specified to the children of the nodes specified.

Usage

```
exogenize(cg, nodes)
```

Arguments

`cg`	A `caugi` object of class `"DAG"`.
`nodes`	A character vector of node names to exogenize. Must be a subset of the nodes in the graph.

Value

A `caugi` object representing the exogenized graph.

See Also

Other operations: `condition_marginalize()`, `dag_from_pdag()`, `latent_project()`, `meek_closure()`, `moralize()`, `mutate_caugi()`, `normalize_latent_structure()`, `skeleton()`

Examples

```
cg <- caugi(A %-->% B, class = "DAG")
exogenize(cg, nodes = "B") # A, B
```

`exogenous` *Get all exogenous nodes in a* `caugi`

Description

Get all exogenous nodes (nodes with no parents) in a `caugi`.

Usage

```
exogenous(cg, undirected_as_parents = FALSE)
```

Arguments

`cg`	A `caugi` object.
`undirected_as_parents`	Logical; if `TRUE`, undirected edges are treated as (possible) parents, if `FALSE` (default), undirected edges are ignored.

Value

Either a character vector of node names (if a single node is requested) or a list of character vectors (if multiple nodes are requested).

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg <- caugi(
  A %-->% B,
  B %-->% C,
  class = "DAG"
)
exogenous(cg) # "A"
```

`export-classes` *Export Format Classes*

Description

S7 classes for representing caugi graphs in various export formats. These classes provide a common interface for serializing graphs to different text formats like DOT, GraphML, JSON, etc.

Base Class

`caugi_export` is the base class for all export formats. It provides:

- `content` property: Character string containing the serialized graph
- `format` property: Character string indicating the format type
- Common methods: `print()`, `as.character()`, `knit_print()`

Subclasses

- `caugi_dot`: DOT format for Graphviz visualization
- `caugi_mermaid`: Mermaid format for web-based visualization

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `format-caugi`, `format-dot`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

`format-caugi` *Caugi Native Format Serialization*

Description

Functions for converting caugi graphs to and from the native caugi JSON format. This format provides efficient, reproducible serialization for saving and sharing caugi graphs.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-dot`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

`format-dot` *DOT Format Export and Import*

Description

Functions for converting caugi graphs to and from Graphviz DOT format. The DOT format is a plain text graph description language used by Graphviz tools for visualization.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

`format-graphml` *GraphML Format Export and Import*

Description

Functions for converting caugi graphs to and from GraphML format. GraphML is an XML-based file format for graphs supported by many graph tools and libraries.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-dot`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

`format-mermaid` *Mermaid Format Export*

Description

Functions for converting caugi graphs to Mermaid flowchart format. Mermaid is a JavaScript-based diagramming tool that renders in web browsers and is natively supported by Quarto, GitHub, and many other platforms.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-dot`, `format-graphml`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

`generate_graph` *Generate a* `caugi` *using Erdős-Rényi.*

Description

Sample a random DAG or CPDAG using Erdős-Rényi for random graph generation.

Usage

```
generate_graph(n, m = NULL, p = NULL, class = c("DAG", "CPDAG"), seed = NULL)
```

Arguments

`n`	Integer >= 0. Number of nodes in the graph.
`m`	Integer in `0, n*(n-1)/2`. Number of edges in the graph. Exactly one of `m` or `p` must be supplied.
`p`	Numeric in `[0,1]`. Probability of edge creation. Exactly one of `m` or `p` must be supplied.
`class`	"DAG" or "CPDAG".
`seed`	Optional integer; random seed for reproducibility.

Value

The sampled `caugi` object.

See Also

Other simulation functions: `simulate_data()`

Examples

```
# generate a random DAG with 5 nodes and 4 edges
dag <- generate_graph(n = 5, m = 4, class = "DAG")

# generate a random CPDAG with 5 nodes and edge probability 0.3
cpdag <- generate_graph(n = 5, p = 0.3, class = "CPDAG")
```

`hd` *Hamming Distance*

Description

Compute the Hamming Distance between two graphs.

Usage

```
hd(cg1, cg2, normalized = FALSE)
```

Arguments

`cg1`	A `caugi` object.
`cg2`	A `caugi` object.
`normalized`	Logical; if `TRUE`, returns the normalized Hamming Distance.

Value

An integer representing the Hamming Distance between the two graphs, if `normalized` = `FALSE`, or a numeric between 0 and 1 if `normalized` = `TRUE`.

See Also

Other metrics: `aid()`, `shd()`

Examples

```
cg1 <- caugi(A %-->% B %-->% C, D %-->% C, class = "DAG")
cg2 <- caugi(A %-->% B %-->% C, D %---% C, class = "PDAG")
hd(cg1, cg2) # 0
```

`is_acyclic` *Is the* `caugi` *acyclic?*

Description

Checks if the given `caugi` graph is acyclic.

Usage

```
is_acyclic(cg, force_check = FALSE)
```

Arguments

`cg`	A `caugi` object.
`force_check`	Logical; if `TRUE`, the function will test if the graph is acyclic, if `FALSE` (default), it will look at the graph class and match it, if possible.

Details

Logically, it should not be possible to have a graph class of "DAG", "PDAG", or "MPDAG" that has cycles, but in case the user modified the graph after creation in some unforeseen way that could have introduced cycles, this function allows to force a check of acyclicity, if needed.

Value

A logical value indicating whether the graph is acyclic.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg_acyclic <- caugi(
  A %-->% B,
  B %-->% C,
  class = "DAG"
)
is_acyclic(cg_acyclic) # TRUE
cg_cyclic <- caugi(
  A %-->% B,
  B %-->% C,
  C %-->% A,
  class = "UNKNOWN"
)
is_acyclic(cg_cyclic) # FALSE
```

`is_admg` *Is the* `caugi` *graph an ADMG?*

Description

Checks if the given `caugi` graph is an Acyclic Directed Mixed Graph (ADMG).

An ADMG contains only directed (`-->`) and bidirected (`<->`) edges, and the directed part must be acyclic.

Usage

```
is_admg(cg, force_check = FALSE)
```

Arguments

`cg`	A `caugi` object.
`force_check`	Logical; if `TRUE`, the function will test if the graph is an ADMG, if `FALSE` (default), it will look at the graph class and match it, if possible.

Value

A logical value indicating whether the graph is an ADMG.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg_admg <- caugi(
  A %-->% B,
  A %<->% C,
  class = "ADMG"
)
is_admg(cg_admg) # TRUE

cg_dag <- caugi(
  A %-->% B,
  class = "DAG"
)
is_admg(cg_dag) # TRUE (DAGs are valid ADMGs)
```

`is_ag` *Is the* `caugi` *graph an AG?*

Description

Checks if the given `caugi` graph is an Ancestral Graph (AG).

An AG contains directed (`-->`), bidirected (`<->`), and undirected (`---`) edges, and must satisfy ancestral graph constraints (no directed cycles, anterior constraint, and undirected constraint).

Usage

```
is_ag(cg, force_check = FALSE)
```

Arguments

`cg`	A `caugi` object.
`force_check`	Logical; if `TRUE`, the function will test if the graph is an AG, if `FALSE` (default), it will look at the graph class and match it, if possible.

Value

A logical value indicating whether the graph is an AG.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg_ag <- caugi(
  A %-->% B,
  C %<->% D,
  E %---% F,
  class = "AG"
)
is_ag(cg_ag) # TRUE

cg_ug <- caugi(
  A %---% B,
  class = "UG"
)
is_ag(cg_ug) # TRUE (UGs are valid AGs)
```

`is_caugi` *Is it a* `caugi` *graph?*

Description

Checks if the given object is a `caugi`. Mostly used internally to validate inputs.

Usage

```
is_caugi(x, throw_error = FALSE)
```

Arguments

`x`	An object to check.
`throw_error`	Logical; if `TRUE`, throws an error if `x` is not a `caugi`.

Value

A logical value indicating whether the object is a `caugi`.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg <- caugi(
  A %-->% B,
  class = "DAG"
)

is_caugi(cg) # TRUE
```

`is_cpdag` *Is the* `caugi` *graph a CPDAG?*

Description

Checks if the given `caugi` graph is a Complete Partially Directed Acyclic Graph (CPDAG).

Usage

```
is_cpdag(cg)
```

Arguments

`cg`	A `caugi` object.

Value

A logical value indicating whether the graph is a CPDAG.

References

C. Meek (1995). Causal inference and causal explanation with background knowledge. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI-95)*, pp. 403–411. Morgan Kaufmann.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg_cpdag <- caugi(
  A %---% B,
  A %-->% C,
  B %-->% C,
  class = "PDAG"
)
is_cpdag(cg_cpdag) # TRUE

cg_not_cpdag <- caugi(
  A %---% B,
  A %---% C,
  B %-->% C,
  class = "PDAG"
)
is_cpdag(cg_not_cpdag) # FALSE
```

`is_dag` *Is the* `caugi` *graph a DAG?*

Description

Checks if the given `caugi` graph is a Directed Acyclic Graph (DAG).

Usage

```
is_dag(cg, force_check = FALSE)
```

Arguments

`cg`	A `caugi` object.
`force_check`	Logical; if `TRUE`, the function will test if the graph is a DAG, if `FALSE` (default), it will look at the graph class and match it, if possible.

Value

A logical value indicating whether the graph is a DAG.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg_dag_class <- caugi(
  A %-->% B,
  class = "DAG"
)
is_dag(cg_dag_class) # TRUE
cg_dag_but_pdag_class <- caugi(
  A %-->% B,
  class = "PDAG"
)
is_dag(cg_dag_but_pdag_class) # TRUE
cg_cyclic <- caugi(
  A %-->% B,
  B %-->% C,
  C %-->% A,
  class = "UNKNOWN",
  simple = FALSE
)
is_dag(cg_cyclic) # FALSE

cg_undirected <- caugi(
  A %---% B,
  class = "UNKNOWN"
)
is_dag(cg_undirected) # FALSE
```

`is_empty_caugi` *Is the* `caugi` *graph empty?*

Description

Checks if the given `caugi` graph is empty (has no nodes).

Usage

```
is_empty_caugi(cg)
```

Arguments

`cg`	A `caugi` object.

Value

A logical value indicating whether the graph is empty.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg_empty <- caugi(class = "DAG")
is_empty_caugi(cg_empty) # TRUE
cg_non_empty <- caugi(
  A %-->% B,
  class = "DAG"
)
is_empty_caugi(cg_non_empty) # FALSE

cg_no_edges_but_has_nodes <- caugi(
  A, B,
  class = "DAG"
)
is_empty_caugi(cg_no_edges_but_has_nodes) # FALSE
```

`is_mag` *Is the* `caugi` *graph a MAG?*

Description

Checks if the given `caugi` graph is a Maximal Ancestral Graph (MAG).

A MAG is an ancestral graph where no additional edge can be added without violating the ancestral graph constraints or changing the encoded independence model.

Usage

```
is_mag(cg, force_check = FALSE)
```

Arguments

`cg`	A `caugi` object.
`force_check`	Logical; if `TRUE`, the function will test if the graph is a MAG, if `FALSE` (default), it will look at the graph class and match it, if possible.

Value

A logical value indicating whether the graph is a MAG.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg_ag <- caugi(
  A %-->% B,
  B %-->% C,
  class = "AG"
)
is_mag(cg_ag) # TRUE (0 and 2 are m-separated by {B})
```

`is_mpdag` *Is the* `caugi` *graph an MPDAG?*

Description

Checks if the given `caugi` graph is a Maximally oriented Partially Directed Acyclic Graph (MPDAG), i.e. a PDAG where no additional edge orientations are implied by Meek's rules (R1–R4).

Usage

```
is_mpdag(cg)
```

Arguments

`cg`	A `caugi` object.

Details

If the graph is not PDAG-compatible, the function returns `FALSE`.

Value

A logical value indicating whether the graph is an MPDAG.

References

C. Meek (1995). Causal inference and causal explanation with background knowledge. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI-95)*, pp. 403–411. Morgan Kaufmann.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg_not_mpdag <- caugi(
  A %---% B,
  A %-->% C,
  C %-->% B,
  class = "PDAG"
)
is_mpdag(cg_not_mpdag) # FALSE
```

`is_pdag` *Is the* `caugi` *graph a PDAG?*

Description

Checks if the given `caugi` graph is a Partially Directed Acyclic Graph (PDAG).

Usage

```
is_pdag(cg, force_check = FALSE)
```

Arguments

`cg`	A `caugi` object.
`force_check`	Logical; if `TRUE`, the function will test if the graph is a PDAG, if `FALSE` (default), it will look at the graph class and match it, if possible.

Value

A logical value indicating whether the graph is a PDAG.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg_dag_class <- caugi(
  A %-->% B,
  class = "DAG"
)
is_pdag(cg_dag_class) # TRUE
cg_dag_but_pdag_class <- caugi(
  A %-->% B,
  class = "PDAG"
)
is_pdag(cg_dag_but_pdag_class) # TRUE
cg_cyclic <- caugi(
  A %-->% B,
  B %-->% C,
  C %-->% A,
  D %---% A,
  class = "UNKNOWN",
  simple = FALSE
)
is_pdag(cg_cyclic) # FALSE

cg_undirected <- caugi(
  A %---% B,
  class = "UNKNOWN"
)
is_pdag(cg_undirected) # TRUE

cg_pag <- caugi(
  A %o->% B,
  class = "UNKNOWN"
)
is_pdag(cg_pag) # FALSE
```

`is_simple` *Is the* `caugi` *graph simple?*

Description

Checks if the given `caugi` graph is simple (no self-loops and no parallel edges).

Usage

```
is_simple(cg, force_check = FALSE)
```

Arguments

`cg`	A `caugi` object.
`force_check`	Logical; if `TRUE`, force a check against the compiled graph representation. If `FALSE` (default), return the declared `simple` property.

Value

A logical value indicating whether the graph is simple.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg_simple <- caugi(
  A %-->% B,
  class = "DAG"
)
is_simple(cg_simple) # TRUE

cg_nonsimple <- caugi(
  A %-->% B,
  A %<->% B,
  class = "UNKNOWN",
  simple = FALSE
)
is_simple(cg_nonsimple) # FALSE
```

`is_ug` *Is the* `caugi` *graph an UG?*

Description

Checks if the given `caugi` graph is an undirected graph (UG).

Usage

```
is_ug(cg, force_check = FALSE)
```

Arguments

`cg`	A `caugi` object.
`force_check`	Logical; if `TRUE`, the function will test if the graph is an UG, if `FALSE` (default), it will look at the graph class and match it, if possible.

Value

A logical value indicating whether the graph is an UG.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg_ug_class <- caugi(
  A %---% B,
  class = "UG"
)
is_ug(cg_ug_class) # TRUE
cg_not_ug <- caugi(
  A %-->% B,
  class = "DAG"
)
is_ug(cg_not_ug) # FALSE
```

`is_valid_adjustment_admg`

Is a set a valid adjustment set in an ADMG?

Description

Checks whether `Z` is a valid adjustment set for estimating the causal effect of `X` on `Y` in an ADMG using the generalized adjustment criterion.

Usage

```
is_valid_adjustment_admg(
  cg,
  X = NULL,
  Y = NULL,
  Z = NULL,
  X_index = NULL,
```

```
  Y_index = NULL,
  Z_index = NULL
)
```

Arguments

`cg`	A `caugi` object of class ADMG.
`X, Y`	Node names (can be vectors for multiple treatments/outcomes).
`Z`	Conditioning set (character vector of node names).
`X_index, Y_index, Z_index`	Optional 1-based indices.

Value

Logical value indicating if the adjustment set is valid.

See Also

Other adjustment: `adjustment_set()`, `all_adjustment_sets_admg()`, `all_backdoor_sets()`, `d_separated()`, `is_valid_backdoor()`, `minimal_separator()`

Examples

```
# Classic confounding
cg <- caugi(
  L %-->% X,
  X %-->% Y,
  L %-->% Y,
  class = "ADMG"
)

is_valid_adjustment_admg(cg, X = "X", Y = "Y", Z = NULL) # FALSE
is_valid_adjustment_admg(cg, X = "X", Y = "Y", Z = "L") # TRUE
```

`is_valid_backdoor` *Is a backdoor set valid?*

Description

Checks whether `Z` is a valid backdoor adjustment set for `X --> Y`.

Usage

```
is_valid_backdoor(
  cg,
  X = NULL,
  Y = NULL,
  Z = NULL,
  X_index = NULL,
  Y_index = NULL,
  Z_index = NULL
)
```

Arguments

`cg`	A `caugi` object.
`X, Y`	Single node names.
`Z`	Optional node set for conditioning
`X_index, Y_index, Z_index`	Optional 1-based indices.

Value

Logical value indicating if backdoor is valid or not.

See Also

Other adjustment: `adjustment_set()`, `all_adjustment_sets_admg()`, `all_backdoor_sets()`, `d_separated()`, `is_valid_adjustment_admg()`, `minimal_separator()`

Examples

```
cg <- caugi(
  C %-->% X,
  X %-->% F,
  X %-->% D,
  A %-->% X,
  A %-->% K,
  K %-->% Y,
  D %-->% Y,
  D %-->% G,
  Y %-->% H,
  class = "DAG"
)

is_valid_backdoor(cg, X = "X", Y = "Y", Z = NULL) # FALSE
is_valid_backdoor(cg, X = "X", Y = "Y", Z = "K") # TRUE
is_valid_backdoor(cg, X = "X", Y = "Y", Z = c("A", "C")) # TRUE
```

`knit_print.caugi_export` *Knit Print Method for caugi_export*

Description

Renders caugi export objects as code blocks in Quarto/R Markdown documents. This method is automatically invoked when an export object is the last expression in a code chunk.

Arguments

`x`	A `caugi_export` object.
`...`	Additional arguments (currently unused).

Details

This method enables seamless rendering of caugi graphs in Quarto and R Markdown. The code block type is determined by the export format. Simply use an export function (e.g., `to_dot(cg)`) as the last expression in a chunk with `output: asis`:

```
#| output: asis
to_dot(cg)
```

Value

A `knit_asis` object for rendering by knitr.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-dot`, `format-graphml`, `format-mermaid`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

`latent_project` *Project latent variables from a DAG to an ADMG*

Description

Projects out latent (unobserved) variables from a DAG to produce an Acyclic Directed Mixed Graph (ADMG) over the observed variables.

Usage

```
latent_project(cg, latents)
```

Arguments

`cg`	A `caugi` object of class `"DAG"`.
`latents`	Character vector of latent variable names to project out.

Value

A `caugi` object of class `"ADMG"` containing only the observed variables.

See Also

Other operations: `condition_marginalize()`, `dag_from_pdag()`, `exogenize()`, `meek_closure()`, `moralize()`, `mutate_caugi()`, `normalize_latent_structure()`, `skeleton()`

Examples

```
# DAG with latent confounder U
dag <- caugi(
  U %-->% X,
  U %-->% Y,
  X %-->% Y,
  class = "DAG"
)

# Project out the latent variable
admg <- latent_project(dag, latents = "U")
# Result: X -> Y, X <-> Y (children of U become bidirected-connected)
edges(admg)

# DAG with directed path through latent
dag2 <- caugi(
  X %-->% L,
  L %-->% Y,
  class = "DAG"
)

# Project out the latent variable
admg2 <- latent_project(dag2, latents = "L")
# Result: X -> Y (directed path X -> L -> Y becomes X -> Y)
edges(admg2)
```

`length` *Length of a* `caugi`

Description

Returns the number of nodes in the graph.

Arguments

`x`	A `caugi` object.

Value

An integer representing the number of nodes.

See Also

Other caugi methods: `print()`

Examples

```
cg <- caugi(
  A %-->% B,
  class = "DAG"
)
length(cg) # 2

cg2 <- caugi(
  A %-->% B + C,
  nodes = LETTERS[1:5],
  class = "DAG"
)
length(cg2) # 5
```

`markov_blanket` *Get Markov blanket of nodes in a* `caugi`

Description

Get Markov blanket of nodes in a `caugi`

Usage

```
markov_blanket(cg, nodes = NULL, index = NULL)
```

Arguments

`cg`	A `caugi` object.
`nodes`	A character vector of node names.
`index`	A vector of node indexes.

Value

Either a character vector of node names (if a single node is requested) or a list of character vectors (if multiple nodes are requested).

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg <- caugi(
  A %-->% B,
  B %-->% C,
  class = "DAG"
)
markov_blanket(cg, "A") # "B"
markov_blanket(cg, index = 2) # "A" "C"
markov_blanket(cg, "B") # "A" "C"
markov_blanket(cg, c("B", "C"))
#> $B
#> [1] "A" "C"
#>
#> $C
#> [1] "B"
```

`meek_closure` *Apply Meek closure to a PDAG*

Description

Applies Meek's orientation rules (R1–R4) repeatedly to a PDAG until no more orientations are implied.

Usage

```
meek_closure(cg)
```

Arguments

`cg`	A `caugi` object. Must be PDAG-compatible.

Value

A `caugi` object of class `"PDAG"` that is closed under Meek's rules.

References

C. Meek (1995). Causal inference and causal explanation with background knowledge. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI-95)*, pp. 403–411. Morgan Kaufmann.

See Also

Other operations: `condition_marginalize()`, `dag_from_pdag()`, `exogenize()`, `latent_project()`, `moralize()`, `mutate_caugi()`, `normalize_latent_structure()`, `skeleton()`

Examples

```
pdag <- caugi(
  A %---% B,
  A %-->% C,
  C %-->% B,
  class = "PDAG"
)
mpdag <- meek_closure(pdag)
edges(mpdag)
```

`minimal_separator` *Compute a minimal separator*

Description

Computes a minimal separator Z for sets X and Y, optionally with mandatory inclusions and restrictions on the separator. Supports DAGs (where the result is a d-separator), ADMGs, and ancestral graphs (where the result is an m-separator).

Usage

```
minimal_separator(
  cg,
  X = NULL,
  Y = NULL,
  I = character(0),
  R = NULL,
  X_index = NULL,
  Y_index = NULL,
  I_index = NULL,
  R_index = NULL
)

minimal_d_separator(
  cg,
  X = NULL,
  Y = NULL,
  I = character(0),
  R = NULL,
  X_index = NULL,
  Y_index = NULL,
```

```
  I_index = NULL,
  R_index = NULL
)
```

Arguments

`cg`	A `caugi` object (DAG, ADMG, or AG).
`X, Y`	Character vectors of node names. Use `*_index` to pass 1-based indices.
`I`	Nodes that must be included in the separator.
`R`	Nodes allowed in the separator. If `NULL`, uses all nodes excluding X and Y.
`X_index, Y_index, I_index, R_index`	Optional numeric 1-based indices (exclusive with corresponding name parameters).

Details

A separator Z for X and Y is a set of nodes such that conditioning on Z d- or m-separates X from Y in the graph. This function returns a minimal separator: no proper subset of Z (excluding I) still separates X and Y.

The algorithm runs in linear time O(n + m) and is unified across graph classes via the Bayes-ball reachability of van der Zander & Liśkiewicz (2020). For DAGs the algorithm specializes to FINDMINSEPINDAG; for ADMGs and AGs (and their subclasses CPDAG, RCG, MAG) it uses the general FINDMINSEP over mixed graphs.

Value

A character vector of node names representing the minimal separator, or `NULL` if no valid separator exists within the restriction R.

Source

van der Zander, B. & Liśkiewicz, M. (2020). Finding Minimal d-separators in Linear Time and Applications. Proceedings of The 35th Uncertainty in Artificial Intelligence Conference, in Proceedings of Machine Learning Research 115:637-647 Available from `https://proceedings.mlr.press/v115/van-der-zander20a.html`.

References

van der Zander, B. & Liśkiewicz, M. (2020). Finding Minimal d-separators in Linear Time and Applications. In *Proceedings of the 35th Conference on Uncertainty in Artificial Intelligence (UAI 2020)*, PMLR 115:637–647. `https://proceedings.mlr.press/v115/van-der-zander20a.html`.

See Also

Other adjustment: `adjustment_set()`, `all_adjustment_sets_admg()`, `all_backdoor_sets()`, `d_separated()`, `is_valid_adjustment_admg()`, `is_valid_backdoor()`

Examples

```
cg <- caugi(
  A %-->% X,
  X %-->% M,
  M %-->% Y,
  A %-->% Y,
  class = "DAG"
)

# Find any minimal separator between X and Y
minimal_separator(cg, "X", "Y")

# Force M to be in the separator
minimal_separator(cg, "X", "Y", I = "M")

# Restrict separator to only {A, M}
minimal_separator(cg, "X", "Y", R = c("A", "M"))

# Works on ADMGs (returns a minimal m-separator)
admg <- caugi(
  X %-->% Y,
  L %-->% X,
  L %-->% Y,
  class = "ADMG"
)
minimal_separator(admg, "X", "Y")
```

`moralize` *Moralize a DAG*

Description

Moralizing a DAG involves connecting all parents of each node and then converting all directed edges into undirected edges.

Usage

```
moralize(cg)
```

Arguments

`cg`	A `caugi` object (DAG).

Details

This changes the graph from a Directed Acyclic Graph (DAG) to an Undirected Graph (UG), also known as a Markov Graph.

Value

A `caugi` object representing the moralized graph (UG).

See Also

Other operations: `condition_marginalize()`, `dag_from_pdag()`, `exogenize()`, `latent_project()`, `meek_closure()`, `mutate_caugi()`, `normalize_latent_structure()`, `skeleton()`

Examples

```
cg <- caugi(A %-->% C, B %-->% C, class = "DAG")
moralize(cg) # A -- B, A -- C, B -- C
```

`mutate_caugi` *Mutate* `caugi` *class*

Description

Mutate the `caugi` class from one graph class to another, if possible. For example, convert a `DAG` to a `PDAG`, or a fully directed `caugi` of class `UNKNOWN` to a `DAG`. Throws an error if not possible.

Usage

```
mutate_caugi(cg, class)
```

Arguments

`cg`	A caugi object.
`class`	A character string specifying the new class.

Details

This function returns a copy of the object, and the original remains unchanged.

Value

A `caugi` object of the specified class.

See Also

Other operations: `condition_marginalize()`, `dag_from_pdag()`, `exogenize()`, `latent_project()`, `meek_closure()`, `moralize()`, `normalize_latent_structure()`, `skeleton()`

Examples

```
cg <- caugi(A %-->% B, class = "UNKNOWN")
cg_dag <- mutate_caugi(cg, "DAG")
```

`m_separated` *M-separation test for AGs and ADMGs*

Description

Test whether two sets of nodes are m-separated given a conditioning set in an ancestral graph (AG) or an ADMG.

M-separation generalizes d-separation to AGs/ADMGs and applies to DAGs.

Usage

```
m_separated(
  cg,
  X = NULL,
  Y = NULL,
  Z = NULL,
  X_index = NULL,
  Y_index = NULL,
  Z_index = NULL
)
```

Arguments

`cg`	A `caugi` object of class AG, ADMG, or DAG.
`X, Y, Z`	Character vectors of node names, or `NULL`. Use `*_index` to pass 1-based indices. If `Z` is `NULL` or missing, no nodes are conditioned on.
`X_index, Y_index, Z_index`	Optional numeric 1-based indices (exclusive with `X`,`Y`,`Z` respectively).

Value

A logical value; `TRUE` if `X` and `Y` are m-separated given `Z`.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
# Classic confounding example
cg <- caugi(
  L %-->% X,
  X %-->% Y,
```

```
  L %-->% Y,
  class = "ADMG"
)
m_separated(cg, X = "X", Y = "Y") # FALSE (connected via L)
m_separated(cg, X = "X", Y = "Y", Z = "L") # TRUE (L blocks the path)
```

`neighbors` *Get neighbors of nodes in a* `caugi`

Description

Get neighbors of a node in the graph, optionally filtered by edge direction or type. This function works for all graph classes including `UNKNOWN`.

Usage

```
neighbors(cg, nodes = NULL, index = NULL, mode = "all")

neighbours(cg, nodes = NULL, index = NULL, mode = "all")
```

Arguments

`cg`	A `caugi` object.
`nodes`	A character vector of node names.
`index`	A vector of node indexes.
`mode`	Character; specifies which types of neighbors to return: `"all"` All neighbors (default) `"in"` Parents: nodes with directed edges pointing INTO the target node (equivalent to `parents()`) `"out"` Children: nodes with directed edges pointing OUT from the target node (equivalent to `children()`) `"undirected"` Nodes connected via undirected (`---`) edges `"bidirected"` Nodes connected via bidirected (`<->`) edges (equivalent to `spouses()` for ADMGs) `"partial"` Nodes connected via partial edges (edges with circle endpoints: `o-o`, `o->`, `--o`) Not all modes are valid for all graph classes: • DAG: `"in"`, `"out"`, `"all"` only • PDAG: `"in"`, `"out"`, `"undirected"`, `"all"` • UG: `"undirected"`, `"all"` only • ADMG: `"in"`, `"out"`, `"bidirected"`, `"all"` • UNKNOWN: all modes allowed

Value

Either a character vector of node names (if a single node is requested) or a list of character vectors (if multiple nodes are requested).

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg <- caugi(
  A %-->% B,
  B %-->% C,
  class = "DAG"
)
neighbors(cg, "A") # "B"
neighbors(cg, index = 2) # "A" "C"
neighbors(cg, "B") # "A" "C"
neighbors(cg, c("B", "C"))
#> $B
#> [1] "A" "C"
#>
#> $C
#> [1] "B"

# Using mode to filter by edge direction
neighbors(cg, "B", mode = "in") # "A" (parents)
neighbors(cg, "B", mode = "out") # "C" (children)

# Works for UNKNOWN graphs too
cg_unknown <- caugi(
  A %-->% B,
  B %---% C,
  C %o->% D,
  class = "UNKNOWN"
)
neighbors(cg_unknown, "B", mode = "in") # "A"
neighbors(cg_unknown, "B", mode = "undirected") # "C"
neighbors(cg_unknown, "C", mode = "partial") # "D"
```

---

nodes *Get nodes or edges of a* `caugi`

Description

Get nodes or edges of a `caugi`

Usage

```
nodes(cg, ...)

vertices(cg, ...)

V(cg, ...)
```

Arguments

`cg`	A `caugi` object.
`...`	Additional arguments (currently unused).

Value

A `data.table` with a `name` column.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg <- caugi(
  A %-->% B,
  B %-->% C,
  D,
  class = "DAG"
)
nodes(cg) # returns the data.table with nodes A, B, C, D
```

---

normalize_latent_structure

Normalize latent structure in a DAG

Description

Normalizes a DAG with latent variables while preserving the induced marginal model over the observed variables. This is done by:

(1) exogenizing all latent nodes (making them parentless), (2) removing exogenous latent nodes with at most one child, and (3) removing exogenous latent nodes whose child sets are strict subsets of another latent node's child set.

This corresponds to Lemmas 1–3 in Evans (2016).

Usage

```
normalize_latent_structure(cg, latents)
```

Arguments

`cg`	A `caugi` object of class "DAG".
`latents`	Character vector of latent node names.

Value

A `caugi` object of class "DAG".

References

Evans, R. J. (2016). Graphs for margins of Bayesian networks. Scandinavian Journal of Statistics, 43(3), 625–648. doi:10.1111/sjos.12194

See Also

Other operations: `condition_marginalize()`, `dag_from_pdag()`, `exogenize()`, `latent_project()`, `meek_closure()`, `moralize()`, `mutate_caugi()`, `skeleton()`

Examples

```
dag <- caugi(
  A %-->% U,
  U %-->% X + Y,
  class = "DAG"
)

normalize_latent_structure(dag, latents = "U")

# More complex example with two latents and nested child sets
dag2 <- caugi(
  A %-->% U,
  U %-->% X + Y + Z,
  U2 %-->% Y + Z,
  class = "DAG"
)
normalize_latent_structure(dag2, c("U", "U2"))
```

`parents` *Get parents of nodes in a* `caugi`

Description

Get parents of nodes in a graph (nodes with directed edges pointing INTO the target node). This is equivalent to `neighbors(cg, nodes, mode = "in")`.

Note that not both nodes and index can be given.

Usage

```
parents(cg, nodes = NULL, index = NULL)
```

Arguments

`cg`	A `caugi` object.
`nodes`	A character vector of node names.
`index`	A vector of node indexes.

Value

Either a character vector of node names (if a single node is requested) or a list of character vectors (if multiple nodes are requested).

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg <- caugi(
  A %-->% B,
  B %-->% C,
  class = "DAG"
)
parents(cg, "A") # NULL
parents(cg, index = 2) # "A"
parents(cg, "B") # "A"
parents(cg, c("B", "C"))
#> $B
#> [1] "A"
#>
#> $C
#> [1] "B"
```

`plot` *Create a caugi Graph Plot Object*

Description

Creates a grid graphics object (gTree) representing a `caugi` graph. If the graph has not been built yet, it will be built automatically before plotting. This implementation uses idiomatic grid graphics with viewports for proper coordinate handling.

Arguments

`x`	A `caugi` object. Must contain only directed edges for Sugiyama layout.
`layout`	Specifies the graph layout method. Can be: • A character string: `"auto"` (default), `"sugiyama"`, `"fruchterman-reingold"`, `"kamada-kawai"`, `"bipartite"`. See `caugi_layout()` for details. • A layout function: e.g., `caugi_layout_sugiyama`, `caugi_layout_bipartite`, etc. The function will be called with `x` and any additional arguments passed via `...`. • A pre-computed layout data.frame with columns `name`, `x`, and `y`.
`...`	Additional arguments passed to `caugi_layout()`. For bipartite layouts, include `partition` (logical vector) and `orientation` (`"rows"` or `"columns"`).
`node_style`	List of node styling parameters. Supports: • Appearance (passed to `gpar()`): `fill`, `col`, `lwd`, `lty`, `alpha` • Geometry: `padding` (text padding inside nodes in mm, default 2), `size` (node size multiplier, default 1) • Local overrides via `by_node`: a named list of nodes with their own style lists, e.g. `by_node = list(A = list(fill = "red"), B = list(col = "blue"))`

`edge_style`	List of edge styling parameters. Can specify global options or per-type options via `directed`, `undirected`, `bidirected`, `partial`. Supports:
	• Appearance (passed to `gpar()`): `col`, `lwd`, `lty`, `alpha`, `fill`.
	• Geometry: `arrow_size` (arrow length in mm, default 3), `circle_size` (radius of endpoint circles for partial edges in mm, default 1.5)
	• Local overrides via `by_edge`: a named list with:
	  – Node-wide styles: applied to all edges touching a node, e.g. `A = list(col = "red", lwd = 2)`
	  – Specific edges: nested named lists for particular edges, e.g. `A = list(B = list(col = "blue", lwd = 4))`
	Multiple levels can be combined: **Style precedence** (highest to lowest): specific edge settings > node-wide settings > edge type settings > global settings.

`label_style`	List of label styling parameters. Supports: • Appearance (passed to `gpar()`): `col`, `fontsize`, `fontface`, `fontfamily`, `cex`

`tier_style`	List of tier box styling parameters. Tier boxes are shown when `boxes = TRUE` is set within this list. Supports: • Appearance (passed to `gpar()`): `fill`, `col` (border color), `lwd`, `lty`, `alpha` • Geometry: `padding` (padding around tier nodes as proportion of plot range, default 0.05) • Labels: `labels` (logical or character vector). If `TRUE`, uses tier names from `tiers` argument. If a character vector, uses custom labels (one per tier). If `FALSE` or `NULL` (default), no labels are shown. • Label styling: `label_style` (list with `col`, `fontsize`, `fontface`, etc.) • Values can be scalars (applied to all tiers) or vectors (auto-expanded to each tier in order) • Local overrides via `by_tier`: a named list (using tier names from `tiers` argument) or indexed list for per-tier customization, e.g. `by_tier = list(exposures = list(fill = "lightblue"), outcome = list(fill = "yellow"))` or `by_tier = list("1" = list(fill = "lightblue"))`
`main`	Optional character string for plot title. If `NULL` (default), no title is displayed.
`title_style`	List of title styling parameters. Supports: • Appearance (passed to `gpar()`): `col`, `fontsize`, `fontface`, `fontfamily`, `cex`
`asp`	Numeric value for the y/x aspect ratio. If `NA` or `NULL` (default), the aspect ratio is automatically determined to fill the available space. Use `asp = 1` to ensure that one unit on the x-axis equals one unit on the y-axis, which respects the layout coordinates. Values other than 1 will stretch the plot accordingly (e.g., `asp = 2` makes the y-axis twice as tall as the x-axis for the same data range).
`outer_margin`	Grid unit specifying outer margin around the plot. Default is `grid::unit(2, "mm")`.
`title_gap`	Grid unit specifying gap between title and graph. Default is `grid::unit(1, "lines")`.

Value

A `caugi_plot` object that wraps a `gTree` for grid graphics display. The plot is automatically drawn when printed or explicitly plotted.

See Also

Other plotting: `add-caugi_plot-caugi_plot`, `caugi_layout()`, `caugi_layout_bipartite()`, `caugi_layout_circle()`, `caugi_layout_fruchterman_reingold()`, `caugi_layout_kamada_kawai()`, `caugi_layout_sugiyama()`, `caugi_layout_tiered()`, `caugi_plot()`, `divide-caugi_plot-caugi_plot`

Examples

```
cg <- caugi(
  A %-->% B + C,
  B %-->% D,
  C %-->% D,
  class = "DAG"
```

```
)

plot(cg)

# Use a specific layout method (as string)
plot(cg, layout = "kamada-kawai")

# Use a layout function
plot(cg, layout = caugi_layout_sugiyama)

# Pre-compute layout and use it
coords <- caugi_layout_fruchterman_reingold(cg)
plot(cg, layout = coords)

# Bipartite layout with a function
cg_bp <- caugi(A %-->% X, B %-->% X, C %-->% Y)
partition <- c(TRUE, TRUE, TRUE, FALSE, FALSE)
plot(cg_bp, layout = caugi_layout_bipartite, partition = partition)

# Customize nodes
plot(cg, node_style = list(fill = "lightgreen", padding = 0.8))

# Customize edges by type
plot(
  cg,
  edge_style = list(
    directed = list(col = "blue", arrow_size = 4),
    undirected = list(col = "red")
  )
)

# Add a title
plot(cg, main = "Causal Graph")

# Customize title
plot(
  cg,
  main = "My Graph",
  title_style = list(fontsize = 18, col = "blue", fontface = "italic")
)

# Respect aspect ratio (1:1)
plot(cg, asp = 1)
```

`posteriors` *Get posteriors of nodes in a* `caugi`

Description

Get the posterior set of nodes in a graph. The posterior set (dual of the anterior set from Richardson and Spirtes, 2002) includes all nodes reachable by following paths where every edge is either undirected or directed away from the source node.

For DAGs, the posterior set equals the descendant set (since there are no undirected edges). For PDAGs, it includes both descendants and nodes reachable via undirected edges.

Usage

```
posteriors(
  cg,
  nodes = NULL,
  index = NULL,
  open = caugi_options("use_open_graph_definition")
)
```

Arguments

`cg`	A `caugi` object of class DAG, PDAG, or AG.
`nodes`	A character vector of node names.
`index`	A vector of node indexes.
`open`	Boolean. Determines how the graph is interpreted when retrieving posteriors. Default is taken from `caugi_options("use_open_graph_definition")`, which by default is TRUE.

Value

Either a character vector of node names (if a single node is requested) or a list of character vectors (if multiple nodes are requested).

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `same_nodes()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
# PDAG example with directed and undirected edges
cg <- caugi(
  A %-->% B %---% C,
  B %-->% D,
  class = "PDAG"
)

posteriors(cg, "A") # B, C, D
posteriors(cg, "A", open = FALSE) # A, B, C, D
```

```
posteriors(cg, "B") # C, D
posteriors(cg, "D") # NULL (no posteriors)

# For DAGs, posteriors equals descendants
cg_dag <- caugi(
  A %-->% B %-->% C,
  class = "DAG"
)
posteriors(cg_dag, "A") # B, C
```

`print` *Print a* `caugi`

Description

Print a `caugi`

Arguments

`x`	A `caugi` object.
`max_nodes`	Optional numeric; maximum number of node names to consider. If `NULL`, the method automatically prints as many as fit on one console line (plus a separate truncation line if needed).
`max_edges`	Optional numeric; maximum number of edges to consider. If `NULL`, the method automatically prints as many edges as fit on two console lines (plus a separate truncation line if needed).
`...`	Not used.

Value

The input `caugi` object, invisibly.

See Also

Other caugi methods: `length()`

Examples

```
cg <- caugi(A %-->% B, class = "DAG")
print(cg)
```

`read_caugi` *Read caugi Graph from File*

Description

Reads a caugi graph from a file in the native caugi JSON format.

Usage

```
read_caugi(path, lazy)
```

Arguments

`path`	Character string specifying the file path.
`lazy`	DEPRECATED, no longer necessary. The graph is always built lazily, so this argument is ignored.

Details

The function validates the file format and version, ensuring compatibility with the current version of the caugi package.

Value

A `caugi` object.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-dot`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_graphml()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

Examples

```
cg <- caugi(
  A %-->% B + C,
  class = "DAG"
)

# Write and read
tmp <- tempfile(fileext = ".caugi.json")
write_caugi(cg, tmp)
cg2 <- read_caugi(tmp)

# Clean up
unlink(tmp)
```

`read_graphml` *Read GraphML File to caugi Graph*

Description

Imports a GraphML file as a caugi graph. Supports GraphML files exported from caugi with full edge type information.

Usage

```
read_graphml(path, class = NULL)
```

Arguments

`path`	File path to the GraphML file.
`class`	Graph class to assign. If `NULL` (default), attempts to read from the GraphML metadata. If not present, defaults to "UNKNOWN".

Details

This function provides basic GraphML import support. It reads:

- Nodes and their IDs
- Edges with source and target
- Edge types (if present in `edge_type` attribute)
- Graph class (if present in graph data)

For GraphML files not created by caugi, edge types default to "–>" for directed graphs and "—" for undirected graphs.

Value

A `caugi` object.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-dot`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

Examples

```
# Create and export a graph
cg <- caugi(
  A %-->% B,
  B %-->% C,
  class = "DAG"
)

tmp <- tempfile(fileext = ".graphml")
write_graphml(cg, tmp)

# Read it back
cg2 <- read_graphml(tmp)

# Clean up
unlink(tmp)
```

`register_caugi_edge` *Register a new edge type in the global registry.*

Description

Register a new edge type in the global registry.

Usage

```
register_caugi_edge(glyph, tail_mark, head_mark, class, symmetric = FALSE)
```

Arguments

`glyph`	A string representing the edge glyph (e.g., `"-->"`, `"<->"`).
`tail_mark`	One of "arrow", "tail", "circle", "other".
`head_mark`	One of "arrow", "tail", "circle", "other".
`class`	One of "directed","undirected","bidirected","partial".
`symmetric`	Logical.

Value

TRUE, invisibly.

See Also

Other registry: `registry`

Examples

```
# first, for reproducability, we reset the registry to default
reset_caugi_registry()

# create a new registry
reg <- caugi_registry()

# register an edge
register_caugi_edge(
  glyph = "<--",
  tail_mark = "arrow",
  head_mark = "tail",
  class = "directed",
  symmetric = FALSE
)

# now, this edge is available for caugi graphs:
cg <- caugi(A %-->% B, B %<--% C, class = "DAG")

# reset the registry to default
reset_caugi_registry()
```

`registry` `caugi` *edge registry*

Description

The `caugi` edge registry stores information about the different edge types that can be used in `caugi` graphs. It maps edge glyphs (e.g., `"-->"`, `"<->"`, `"o->"`, etc.) to their specifications, including tail and head marks, class, and symmetry. The registry allows for dynamic registration of new edge types, enabling users to extend the set of supported edges in `caugi`. It is implemented as a singleton, ensuring that there is a single global instance of the registry throughout the R session.

Usage

```
caugi_registry()

list_caugi_edges()

reset_caugi_registry()

seal_caugi_registry()
```

Details

The intended use of the `caugi` registry is mostly for advanced users and developers. The registry enables users who need to define their own custom edge types in `caugi` directly. . It currently mostly supports the *representation* of new edges, but for users that might want to represent reverse edges, this preserves correctness of reason over these edges.

Value

An `edge_registry` external pointer.

A `data.table` with columns `glyph`, `tail`, `head`, `class`, and `symmetric`, where each row corresponds to a registered edge type.

Functions

- `caugi_registry()`: Access the global edge registry, creating it if needed.
- `list_caugi_edges()`: List all registered edge types in the global registry.
- `reset_caugi_registry()`: Reset the global edge registry to its default state.
- `seal_caugi_registry()`: Seal the global edge registry to prevent further modifications.

See Also

Other registry: `register_caugi_edge()`

Examples

```
# first, for reproducability, we reset the registry to default
reset_caugi_registry()

# create a new registry
reg <- caugi_registry()

# list registered edges
list_caugi_edges()

# register an edge
register_caugi_edge(
  glyph = "<--",
  tail_mark = "arrow",
  head_mark = "tail",
  class = "directed",
  symmetric = FALSE
)

# now, this edge is available for caugi graphs:
list_caugi_edges()
cg <- caugi(A %-->% B, B %<--% C, class = "DAG")

# reset the registry to default
reset_caugi_registry()
```

`same_nodes` *Same nodes?*

Description

Check if two `caugi` objects have the same nodes.

Usage

```
same_nodes(cg1, cg2, throw_error = FALSE)
```

Arguments

`cg1`	A `caugi` object.
`cg2`	A `caugi` object.
`throw_error`	Logical; if `TRUE`, throws an error if the graphs do not have the same nodes.

Value

A logical indicating if the two graphs have the same nodes.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `spouses()`, `subgraph()`, `topological_sort()`

Examples

```
cg1 <- caugi(
  A %-->% B,
  class = "DAG"
)
cg2 <- caugi(
  A %-->% B + C,
  class = "DAG"
)
same_nodes(cg1, cg2) # FALSE
```

`shd` *Structural Hamming Distance*

Description

Compute the Structural Hamming Distance (SHD) between two graphs.

Usage

```
shd(cg1, cg2, normalized = FALSE)
```

Arguments

`cg1`	A `caugi` object.
`cg2`	A `caugi` object.
`normalized`	Logical; if `TRUE`, returns the normalized SHD.

Value

An integer representing the Hamming Distance between the two graphs, if `normalized` = `FALSE`, or a numeric between 0 and 1 if `normalized` = `TRUE`.

See Also

Other metrics: `aid()`, `hd()`

Examples

```
cg1 <- caugi(A %-->% B %-->% C, D %-->% C, class = "DAG")
cg2 <- caugi(A %-->% B %-->% C, D %---% C, class = "PDAG")
shd(cg1, cg2) # 1
```

`simulate_data` *Simulate data from a `caugi` DAG.*

Description

Simulate data from a `caugi` object of class DAG using a linear structural equation model (SEM). As standard, the data is simulated from a DAG, where each node is generated as a linear combination of its parents plus Gaussian noise, following the topological order of the graph. Nodes without custom equations are simulated using auto-generated linear Gaussian relationships.

Usage

```
simulate_data(
  cg,
  n,
  ...,
  standardize = TRUE,
  coef_range = c(0.1, 0.9),
  error_sd = 1,
  seed = NULL
)
```

Arguments

`cg`	A `caugi` object of class DAG.
`n`	Integer; number of observations to simulate.
`...`	Named expressions for custom structural equations. Names must match node names in the graph. Expressions can reference parent node names and the variable `n` (sample size). Nodes without custom equations use auto-generated linear Gaussian relationships.
`standardize`	Logical; if `TRUE`, standardize all variables to have mean 0 and standard deviation 1. Default is `TRUE`.
`coef_range`	Numeric vector of length 2; range for random edge coefficients that will be sampled uniformly. Default is `c(0.1, 0.9)`.
`error_sd`	Numeric; standard deviation for error terms in auto-generated equations. Default is `1`.
`seed`	Optional integer; random seed for reproducibility.

Value

A `data.frame` with `n` rows and one column per node, ordered according to the node order in the graph.

See Also

Other simulation functions: `generate_graph()`

Examples

```
cg <- caugi(A %-->% B, B %-->% C, A %-->% C, class = "DAG")

# Fully automatic simulation
df <- simulate_data(cg, n = 100)

# With standardization
df <- simulate_data(cg, n = 100, standardize = TRUE)

# Custom equations for some nodes
df <- simulate_data(cg, n = 100,
```

```
  A = rnorm(n, mean = 10, sd = 2),
  B = 0.5 * A + rnorm(n, sd = 0.5)
)

# Reproducible simulation
df <- simulate_data(cg, n = 100, seed = 42)
```

`skeleton` *Get the skeleton of a graph*

Description

The skeleton of a graph is obtained by replacing all directed edges with undirected edges.

Usage

```
skeleton(cg)
```

Arguments

`cg`	A `caugi` object. Either a DAG or PDAG.

Details

This changes the graph from any class to an Undirected Graph (UG), also known as a Markov Graph.

Value

A `caugi` object representing the skeleton of the graph (UG).

See Also

Other operations: `condition_marginalize()`, `dag_from_pdag()`, `exogenize()`, `latent_project()`, `meek_closure()`, `moralize()`, `mutate_caugi()`, `normalize_latent_structure()`

Examples

```
cg <- caugi(A %-->% B, class = "DAG")
skeleton(cg) # A --- B
```

`spouses` *Get spouses (bidirected neighbors) of nodes in an ADMG*

Description

Get nodes connected via bidirected edges in an ADMG.

Usage

```
spouses(cg, nodes = NULL, index = NULL)
```

Arguments

`cg`	A `caugi` object of class ADMG.
`nodes`	A character vector of node names.
`index`	A vector of node indexes.

Value

Either a character vector of node names (if a single node is requested) or a list of character vectors (if multiple nodes are requested).

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `subgraph()`, `topological_sort()`

Examples

```
cg <- caugi(
  A %-->% B,
  A %<->% C,
  B %<->% C,
  class = "ADMG"
)
spouses(cg, "A") # "C"
spouses(cg, "C") # c("A", "B")
```

`subgraph` *Get the induced subgraph*

Description

Get the induced subgraph

Usage

```
subgraph(cg, nodes = NULL, index = NULL)
```

Arguments

`cg`	A `caugi` object.
`nodes`	A character vector of node names.
`index`	A vector of node indexes.

Value

A new `caugi` that is a subgraph of the selected nodes.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `topological_sort()`

Examples

```
cg <- caugi(
  A %-->% B,
  B %-->% C,
  class = "DAG"
)
sub_cg <- subgraph(cg, c("B", "C"))
cg2 <- caugi(B %-->% C, class = "DAG")
all(nodes(sub_cg) == nodes(cg2)) # TRUE
all(edges(sub_cg) == edges(cg2)) # TRUE
```

`topological_sort` *Get a topological ordering of a DAG*

Description

Returns a topological ordering of the nodes in a DAG. For every directed edge u -> v in the graph, u will appear before v in the returned ordering.

Usage

```
topological_sort(cg)
```

Arguments

`cg`	A `caugi` object of class DAG.

Value

A character vector of node names in topological order.

See Also

Other queries: `ancestors()`, `anteriors()`, `children()`, `descendants()`, `districts()`, `edge_types()`, `edges()`, `exogenous()`, `is_acyclic()`, `is_admg()`, `is_ag()`, `is_caugi()`, `is_cpdag()`, `is_dag()`, `is_empty_caugi()`, `is_mag()`, `is_mpdag()`, `is_pdag()`, `is_simple()`, `is_ug()`, `m_separated()`, `markov_blanket()`, `neighbors()`, `nodes()`, `parents()`, `posteriors()`, `same_nodes()`, `spouses()`, `subgraph()`

Examples

```
# Simple DAG: A -> B -> C
cg <- caugi(
  A %-->% B,
  B %-->% C,
  class = "DAG"
)
topological_sort(cg) # Returns c("A", "B", "C") or equivalent valid ordering

# DAG with multiple valid orderings
cg2 <- caugi(
  A %-->% C,
  B %-->% C,
  class = "DAG"
)
# Could return c("A", "B", "C") or c("B", "A", "C")
topological_sort(cg2)
```

`to_dot` *Export caugi Graph to DOT Format*

Description

Converts a caugi graph to the Graphviz DOT format as a string. The DOT format can be used with Graphviz tools for visualization and analysis.

Usage

```
to_dot(x, graph_attrs = list(), node_attrs = list(), edge_attrs = list())
```

Arguments

`x`	A caugi object.
`graph_attrs`	Named list of graph attributes (e.g., `list(rankdir = "LR")`).
`node_attrs`	Named list of default node attributes.
`edge_attrs`	Named list of default edge attributes.

Details

The function handles different edge types:

- Directed edges (`-->`) use `->` in DOT
- Undirected edges (`---`) use `--` in DOT (or `->` with `dir=none` in digraphs)
- Bidirected edges (`<->`) use `->` with `[dir=both]` attribute
- Partial edges (`o->`) use `->` with `[arrowtail=odot, dir=both]` attribute

Value

A `caugi_dot` object containing the DOT representation.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-dot`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

Examples

```
cg <- caugi(
  A %-->% B + C,
  B %-->% D,
  C %-->% D,
  class = "DAG"
)
```

```
# Get DOT string
dot <- to_dot(cg)
dot@content

# With custom attributes
dot <- to_dot(
  cg,
  graph_attrs = list(rankdir = "LR"),
  node_attrs = list(shape = "box")
)
```

`to_graphml` *Export caugi Graph to GraphML Format*

Description

Converts a caugi graph to the GraphML XML format as a string. GraphML is widely supported by graph analysis tools and libraries.

Usage

```
to_graphml(x)
```

Arguments

`x`	A `caugi` object.

Details

The GraphML export includes:

- Node IDs and labels
- Edge types stored as a custom `edge_type` attribute
- Graph class stored as a graph-level attribute

Edge types are encoded using the caugi DSL operators (e.g., "–>", "<->"). This allows for perfect round-trip conversion back to caugi.

Value

A `caugi_graphml` object containing the GraphML representation.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-dot`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

Examples

```
cg <- caugi(
  A %-->% B + C,
  B %-->% D,
  C %-->% D,
  class = "DAG"
)

# Get GraphML string
graphml <- to_graphml(cg)
cat(graphml@content)

# Write to file
## Not run:
write_graphml(cg, "graph.graphml")

## End(Not run)
```

`to_mermaid` *Export caugi Graph to Mermaid Format*

Description

Converts a caugi graph to the Mermaid flowchart format as a string. Mermaid diagrams can be rendered in Quarto, R Markdown, GitHub, and many other platforms.

Usage

```
to_mermaid(x, direction = "TD")
```

Arguments

`x`	A `caugi` object.
`direction`	Graph direction: "TB" (top-bottom), "TD" (top-down), "BT" (bottom-top), "LR" (left-right), or "RL" (right-left). Default is "TD".

Details

The function handles different edge types:

- Directed edges (`-->`) use `-->` in Mermaid
- Undirected edges (`---`) use `---` in Mermaid
- Bidirected edges (`<->`) use `<-->` in Mermaid
- Partial edges (`o->`) use `o-->` in Mermaid (circle end)

Node names are automatically escaped if they contain special characters.

Value

A `caugi_mermaid` object containing the Mermaid representation.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-dot`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_graphml()`, `write_caugi()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

Examples

```
cg <- caugi(
  A %-->% B + C,
  B %-->% D,
  C %-->% D,
  class = "DAG"
)

# Get Mermaid string
mmd <- to_mermaid(cg)
mmd@content

# With custom direction
mmd <- to_mermaid(cg, direction = "LR")
```

`write_caugi` *Write caugi Graph to File*

Description

Writes a caugi graph to a file in the native caugi JSON format. This format is designed for reproducibility, caching, and sharing caugi graphs across R sessions.

Usage

```
write_caugi(x, path, comment = NULL, tags = NULL)
```

Arguments

`x`	A caugi object or an object coercible to `caugi`.
`path`	Character string specifying the file path.
`comment`	Optional character string with a comment about the graph.
`tags`	Optional character vector of tags for categorizing the graph.

Details

The caugi format is a versioned JSON schema that captures:

- Graph structure (nodes and edges with their types)
- Graph class (DAG, PDAG, ADMG, UG, etc.)
- Optional metadata (comments and tags)

Edge types are encoded using their DSL operators (e.g., `"-->"`, `"<->"`, `"--"`).

For a complete guide to the format, see `vignette("serialization", package = "caugi")`. The formal JSON Schema is available at: `https://caugi.org/schemas/caugi-v1.schema.json`

Value

Invisibly returns the input `x`.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-dot`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_dot()`, `write_graphml()`, `write_mermaid()`

Examples

```
cg <- caugi(
  A %-->% B + C,
  B %-->% D,
  C %-->% D,
  class = "DAG"
)

# Write to file
tmp <- tempfile(fileext = ".caugi.json")
write_caugi(cg, tmp, comment = "Example DAG")

# Read back
cg2 <- read_caugi(tmp)
identical(edges(cg), edges(cg2))

# Clean up
unlink(tmp)
```

`write_dot` *Write caugi Graph to DOT File*

Description

Writes a caugi graph to a file in Graphviz DOT format.

Usage

```
write_dot(x, file, ...)
```

Arguments

`x`	A `caugi` object.
`file`	Path to output file.
`...`	Additional arguments passed to `to_dot()`, such as `graph_attrs`, `node_attrs`, and `edge_attrs`.

Value

Invisibly returns the path to the file.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-dot`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_graphml()`, `write_mermaid()`

Examples

```
cg <- caugi(
  A %-->% B + C,
  B %-->% D,
  C %-->% D,
  class = "DAG"
)

## Not run:
# Write to file
write_dot(cg, "graph.dot")

# With custom attributes
write_dot(
  cg,
  "graph.dot",
  graph_attrs = list(rankdir = "LR")
)
```

```
## End(Not run)
```

`write_graphml` *Write caugi Graph to GraphML File*

Description

Exports a caugi graph to a GraphML file.

Usage

```
write_graphml(x, path)
```

Arguments

`x`	A `caugi` object.
`path`	File path for the output GraphML file.

Value

Invisibly returns `NULL`. Called for side effects.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-dot`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_mermaid()`

Examples

```
cg <- caugi(A %-->% B + C, class = "DAG")

tmp <- tempfile(fileext = ".graphml")
write_graphml(cg, tmp)

# Read it back
cg2 <- read_graphml(tmp)

# Clean up
unlink(tmp)
```

`write_mermaid` *Write caugi Graph to Mermaid File*

Description

Writes a caugi graph to a file in Mermaid format.

Usage

```
write_mermaid(x, file, ...)
```

Arguments

`x`	A `caugi` object.
`file`	Path to output file.
`...`	Additional arguments passed to `to_mermaid()`, such as `direction`.

Value

Invisibly returns the path to the file.

See Also

Other export: `caugi_deserialize()`, `caugi_dot()`, `caugi_export()`, `caugi_graphml()`, `caugi_mermaid()`, `caugi_serialize()`, `export-classes`, `format-caugi`, `format-dot`, `format-graphml`, `format-mermaid`, `knit_print.caugi_export`, `read_caugi()`, `read_graphml()`, `to_dot()`, `to_graphml()`, `to_mermaid()`, `write_caugi()`, `write_dot()`, `write_graphml()`

Examples

```
cg <- caugi(
  A %-->% B + C,
  B %-->% D,
  C %-->% D,
  class = "DAG"
)

## Not run:
# Write to file
write_mermaid(cg, "graph.mmd")

# With custom direction
write_mermaid(cg, "graph.mmd", direction = "LR")

## End(Not run)
```

Index