Package 'autograph'

November 16, 2026

Title Automatic Plotting and Theming of Many Graphs

Version 1.3.0

Description Visual exploration and presentation of networks should not be difficult.
This package includes functions for plotting networks and network-related metrics with sensible and pretty defaults.
It includes 'ggplot2'-based plot methods for many popular network package classes.
It also includes some novel layout algorithms, and options for straightforward, consistent themes.

URL https://stocnet.github.io/autograph/

BugReports https://github.com/stocnet/autograph/issues

License MIT + file LICENSE

Language en-GB

Encoding UTF-8

LazyData true

Depends R (>= 4.1.0), manynet (>= 2.5.0)

Imports dplyr (>= 1.1.0), ggraph (>= 2.2.0), ggplot2 (>= 4.0.0), graphlayouts (>= 1.1.0), igraph, patchwork

Suggests gganimate, ggforce (>= 0.5.0), gifski, messydates (>= 1.0.0), methods, netrics (>= 1.0.3), systemfonts, testthat (>= 3.0.0)

Enhances ergm, goldfish, RSiena

Config/Needs/build roxygen2, devtools

Config/Needs/check covr, lintr, spelling

Config/Needs/website pkgdown

Config/testthat/parallel true

Config/testthat/edition 3

Config/roxygen2/version 8.1.1

NeedsCompilation no

Author James Hollway [cre, aut, ctb] (IHEID, ORCID: <https://orcid.org/0000-0002-8361-9647>), Henrique Sposito [ctb] (ORCID: <https://orcid.org/0000-0003-3420-6085>)

Maintainer James Hollway <james.hollway@graduateinstitute.ch>

Repository CRAN

Date/Publication 2026-11-16 10:00:09 UTC

Contents

ag_call . . . 3
check_colors . . . 4
check_layout . . . 6
completion . . . 10
count_pages . . . 11
layout_concentric . . . 12
layout_configuration . . . 13
layout_correspondence . . . 14
layout_layered . . . 17
layout_levels . . . 20
layout_matching . . . 21
layout_scaling . . . 22
layout_spatial . . . 24
layout_valence . . . 25
list_fonts . . . 26
made_earlier . . . 27
map_measure . . . 29
map_member . . . 30
map_motifs . . . 31
mdate_scales . . . 31
model_mrqap . . . 32
plot.diagnose_outliers . . . 33
plot.diffusion . . . 36
plot.infernetConfusion . . . 38
plot.mobility . . . 39
plot.network_test . . . 40
plot.opinion . . . 41
plot.result.goldfish . . . 42
plot.transitions . . . 43
plot_convergence . . . 44
plot_gof . . . 45
plot_graphr . . . 47
plot_graphs . . . 52
plot_grapht . . . 54
plot_interp . . . 59
theme_colorblind . . . 60
theme_match . . . 61
theme_medium . . . 62
theme_set . . . 64

Index **66**

ag_call *Consistent palette calls*

Description

These functions assist in calling particular parts of a theme's palette. For example, `ag_base()` will return the current theme's base or background color, and `ag_highlight()` will return the color used in that theme to highlight one or more nodes, lines, or such. `ag_ink()` returns the darker colour that theme writes with: axis text, reference lines, and other chrome. `ag_missing()` returns the neutral that theme sets aside for data that should recede: missing values, isolates counted out of a drawing, and any "other" remainder left when small categories are grouped down. Keeping one colour for all three means a reader learns it once. Keeping the two apart lets the base be light enough to stand away from the highlight while the ink stays dark enough to read. Where the ground changes under a theme – the "print" medium forces white, whatever the theme prefers – `ag_ink()` falls back to black or white rather than return an ink that cannot be read on it. See `check_contrast()` and `stocnet_medium()`.

Using palettes that are high contrast, aesthetically pleasing, and institutionally or thematically consistent is not without its challenges.

Usage

```
ag_base()

ag_ink()

ag_missing()

ag_highlight()

ag_positive()

ag_negative()

ag_qualitative(number)

ag_sequential(number)

ag_divergent(number)

ag_font()
```

Arguments

`number`	Integer of how many category colours to return.

Value

One or more hexcodes as strings.

Colour blindness

The default palettes are designed to be colour-blind friendly. There are different types of colour-blindness. The most common type, red-green colour-blindness, finds it difficult to distinguish between the red and green hues used in the rainbow palette, for instance. Fortunately there are a range of palettes that function fairly well for those who are color-blind. These include the viridis palette, and the ColorBrewer palettes (included in the RColorBrewer package).

An institutional palette is not ours to change, but its order is. Each theme's categorical palette is therefore reordered when the theme is set, so that the first colours a plot draws on are those that stay distinct under each type of colour blindness, and `ag_qualitative()` takes those colours in order rather than interpolating between them. Divergent palettes pair a warm pole with a cool one for the same reason. Use `check_separation()` to check how your own colours fare, and `simulate_colorblind()` to see them as a colour-blind viewer would.

Two further questions are worth asking of a palette. Whether its text can be read on what it sits on is a matter of contrast rather than of hue, and `check_contrast()` scores it against the thresholds of WCAG 2.1. Whether it survives print is a matter of lightness alone, since a greyscale device keeps the luminance of a colour and discards the rest; `simulate_colorblind(type = "grey")` shows that view, and `check_separation()` reports the greyscale distances beside its own score. Most institutional palettes separate by hue and so collapse in greyscale. Where a figure has to print in black and white, use the "bw" theme, or add a second channel such as `node_shape`.

The "rainbow" theme is the exception, and is left in its own order. Its point is fidelity to the spectrum of an observed rainbow, which reordering would destroy, so `ag_qualitative()` samples across its whole length instead. A spectrum is not a colour-blind safe scheme: its reds and greens are exactly the pair that red-green colour blindness cannot separate. Choose it where the order of the categories is itself meaningful, and check the result with `check_separation()`; for categories with no order, another theme serves more readers.

Examples

```
# Single colours from the currently active theme
ag_base()
ag_ink()
ag_highlight()
ag_missing()
ag_positive()
ag_negative()
# Palettes of a requested length
ag_qualitative(3)
ag_sequential(5)
ag_divergent(5)
# The accessors follow whichever theme is set
ag_font()
```

`check_colors` *Checking colours for colour blindness and legibility*

Description

These functions score a set of colours rather than the plot that uses them, so that one palette can be compared with another.

`check_separation()` reports how far apart the colours are, taking the worst case over normal vision and each type of colour blindness, so that a palette is only credited for a difference that every viewer can see.

`check_contrast()` reports whether text can be read on a ground.

Usage

```
check_separation(colors, background = NULL)

check_contrast(colors, background = NULL)
```

Arguments

`colors`	One or more colours, given as hexcodes or as names R knows.
`background`	Optionally, a colour to include in the comparison, so that a colour too pale or too dark to be seen against it is not counted as distinct. By default the current theme's background is used.

Details

The two functions answer different questions, and a palette needs both answered. `check_separation()` asks whether two marks can be told apart, and `check_contrast()` asks whether text can be read on what it sits on. `simulate_colorblind()` asks a third question, which is whether either survives a photocopier.

Distances are Euclidean distances in CIELAB space, the same measure `match_color()` uses. As a rule of thumb, a distance below 10 means two colours are easily confused, 10 to 25 means they are separable but close, and above 25 means they are comfortably distinct. Ratios are those of WCAG 2.1, which asks for at least 4.5 for body text and at least 3 for large text and for graphical objects.

Colour blindness affects about 8% of men and 0.5% of women, and the worst case is taken over deuteranopia, protanopia and tritanopia as well as normal vision.

Value

`check_separation()` returns a square matrix of worst-case distances, with the colours as its dimnames and a missing diagonal, so that `min(x, na.rm = TRUE)` gives the closest pair. A "grey" attribute holds the same matrix as seen in greyscale.

`check_contrast()` returns a square matrix of WCAG contrast ratios, shaped the same way.

References

World Wide Web Consortium. 2018. *Web Content Accessibility Guidelines (WCAG) 2.1*. https://www.w3.org/TR/WCAG21/

See Also

check_layout(), which scores a drawing rather than a palette.

Other themes: list_fonts(), theme_colorblind, theme_medium, theme_set

Examples

```
# How well does the current theme's palette separate five categories?
check_separation(ag_qualitative(5))
# The closest pair in it
min(check_separation(ag_qualitative(5)), na.rm = TRUE)
# And the closest pair once it is printed in greyscale
min(attr(check_separation(ag_qualitative(5)), "grey"), na.rm = TRUE)
# A red and a green that only look different to some viewers
check_separation(c("#B7352D", "#627313"))[1, 2]
# Can the current theme's ink be read on its ground?
check_contrast(ag_ink())[1, 2]
```

---

check_layout *Checking how well a layout draws a network*

Description

These functions score a drawing rather than the network it draws, so that one layout can be compared with another on the same network. Each measures one of the qualities a layout algorithm may set out to optimise.

check_crossings() reports how many other ties each tie crosses. A drawing with no crossings at all is a planar drawing.

check_slopes() reports the slope each tie is drawn at. The number of distinct slopes is the slope number of the drawing, and a drawing of few slopes reads as an orderly one.

check_lengths() reports how long each tie is drawn, as a share of the diagonal of the drawing.

check_angles() reports the smallest angle between the ties that meet at each node. The smallest of these is the angular resolution of the drawing, and a wide angle is what keeps two ties from reading as one.

check_stress() reports how far the distances drawn depart from the distances through the network. A layout that draws two nodes twice as far apart as two others should be drawing a path twice as long.

check_span() reports how many rows of nodes each tie crosses. A layered layout should send most ties to the next row down, and a long tie is one that skips rows to get where it is going.

check_offset() reports how far each tie travels sideways, as a share of the width of the whole drawing. A tie that drops straight down scores zero.

check_drawing() runs every check above except the two for rows, and returns their headline numbers in one row.

Usage

```
check_span(x)

check_offset(x)

check_stress(x)

check_crossings(x, max_full = 2000L)

check_slopes(x, tolerance = 1)

check_lengths(x)

check_angles(x)

check_drawing(x)
```

Arguments

x	A plot, as `graphr()` returns.
max_full	The largest number of ties to compare in full. By default 2000.
tolerance	How many degrees apart two slopes must be to count as two. By default 1.

Value

`check_crossings()` returns one whole number for each tie, being how many other ties it crosses, with `total` and `mean` attributes, and a `sampled` attribute saying whether the ties were sampled.

`check_slopes()` returns one angle in degrees for each tie, between 0 and 180, with `distinct` and `tolerance` attributes.

`check_lengths()` returns one number between 0 and 1 for each tie, with `total`, `max`, `mean`, `variance` and `cv` attributes.

`check_angles()` returns one angle in degrees for each node, with `min`, `mean` and `ideal` attributes.

Each of those returns `NA` for a loop, or for a node with fewer than two ties.

`check_stress()` returns a single number of 0 or more, with a `scale` attribute holding the factor the drawn distances were scaled by, and a `pairs` attribute holding how many pairs were scored.

`check_span()` returns one whole number for each tie, with `total` and `mean` attributes holding the sum and the average.

`check_offset()` returns one number between 0 and 1 for each tie, with a `mean` attribute.

`check_drawing()` returns a one row data frame of class "check_drawing", holding the headline number from each check. Printing it names the direction each column is read in.

Reading the scores

`check_drawing()` reports one number from each check, and the columns are not all read the same way. Every column is better small except `angle_min`, which is better large. `nodes`, `ties` and

`angle_ideal` are not scores at all: the first two say what was drawn, so that two rows of one network can be told from two rows of different ones, and `angle_ideal` is the ceiling that `angle_min` is read against. No layout wins on every column. A layout is chosen by which columns the reader of the figure needs, rather than by how many of them it wins.

Each column is read as follows.

- `crossings` counts the pairs of ties that cross, so fewer is better, and zero is a planar drawing. Zero is not always available: a network of more than three times its nodes, less six, ties cannot be drawn on a plane without a crossing, and a two-mode network of more than twice its nodes, less four, ties cannot either. Where that floor is above zero, read the score against another layout of the same network rather than against zero.
- `slopes` counts the directions the ties are drawn in, so fewer is better. The fewest any drawing can use is half the largest degree, rounded up, since the ties at the busiest node need that many directions to leave it by.
- `length_total`, `length_max` and `length_cv` are all better small. Each length is a share of the diagonal, so it runs between 0 and 1, and 1 is a tie drawn corner to corner. A `length_cv` of 0 means every tie is drawn the same length. The total grows with the number of ties, so compare two drawings of one network by their total, and two networks by the `mean` attribute of `check_lengths()`.
- `angle_min` is the angular resolution in degrees, and is better large. Read it as a share of `angle_ideal`, which is 360 degrees divided by the largest degree, and is the best any drawing of that network could do.
- `stress` is Kruskal's stress-1, so 0 is a perfect drawing. Kruskal read 20% as poor, 10% as fair, 5% as good, and 2.5% as excellent. Those figures were set for psychometric data rather than for networks, which are harder: most pairs of nodes in a small-world network sit two or three steps apart, and a plane holds few such distances at once, so a score near 30% is ordinary and one near 5% is rare. A layout that never set out to draw path distances, such as "layered", "circle" or "configuration", scores poorly by design.

`check_span()` and `check_offset()` are left out of the row, since they measure rows of nodes and so only mean something for a layered layout. Run them beside it where the layout has rows. Both are better small. They answer different questions, and a layered layout needs both answered: `check_span()` asks whether the rows were well chosen, and `check_offset()` asks whether the nodes were well placed within them. The "layered" layout minimises each in turn, and its `ranks` and `alignment` arguments choose how.

The scores belong to the drawing rather than to the network, which is what separates `check_stress()` from the share of distance variance that `graphr()` reports beside it. Draw one network two ways and the stress changes, since one drawing holds its distances better than the other; the share of variance does not, since two dimensions can hold just as much of that network either way. A network whose variance is held poorly sets a floor that no layout gets under.

How the drawing is measured

Every check reads the straight line between two nodes. `graphr()` may draw a tie as an arc, a fan, or a bundle, in which case the score is an approximation of what is drawn. A loop is left out, since it joins a node to itself and so has neither direction nor length on the plane.

`check_crossings()` counts a proper crossing, where two ties meet away from their ends. Two ties that share a node are not counted, since they must meet there. Comparing every pair of ties

costs the square of their number, so above `max_full` ties an evenly spaced sample of the ties is scored instead, and the `sampled` attribute records this. The sample is taken by position rather than at random, so that the same drawing scores the same on every call.

`check_slopes()` reads a slope as an angle between 0 and 180 degrees, since a tie drawn from left to right has the same slope as the same tie drawn from right to left. Two angles closer than `tolerance` are counted as one slope, because two lines that differ by a fraction of a degree read as parallel.

`check_lengths()` divides by the diagonal of the drawing, since a layout may place its nodes on any scale it likes, and two layouts can only be compared once both are on the same one. The spread is reported twice, since the two answer different questions: `variance` is on the scale of the drawing, and `cv` divides by the mean, so only `cv` compares a drawing of long ties with a drawing of short ones.

`check_angles()` compares the directions of the ties at a node, so a node with fewer than two ties scores `NA`. Two ties drawn one on top of the other count once, since a multiplex pair is drawn apart by `graphr()` rather than at the same angle.

`check_stress()` scales the drawn distances to the path distances before it compares them, since a layout may place its nodes on any scale, and the ties are counted unweighted, as `layout_scaling()` counts them. Where a network is disconnected, the pairs with no path between them are left out of the score.

`check_span()` and `check_offset()` read the rows from the plot, as the axis on which the nodes take fewer distinct positions. This is the y axis for "layered" and the x axis for "lineage", so the same score can be compared across the two. For a layout with no rows at all, such as "stress", `check_span()` reports the distance in that axis' ranks, which is not meaningful; the two are for layered layouts.

Source

Kruskal, Joseph B. 1964. "Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis", *Psychometrika* 29(1): 1-27. doi:10.1007/BF02289565

See Also

`check_colors()`, which scores a palette rather than a drawing.

Other mapping: `completion`, `layout_concentric()`, `layout_configuration()`, `layout_correspondence()`, `layout_layered()`, `layout_levels()`, `layout_matching()`, `layout_scaling()`, `layout_spatial()`, `layout_valence()`, `plot_graphr`, `plot_graphs`, `plot_grapht`

Examples

```
sw <- manynet::ison_southern_women
# Every check at once, for two layouts side by side
circled <- graphr(sw, layout = "circle")
rbind(stress = check_drawing(graphr(sw, layout = "stress")),
      circle = check_drawing(circled))
# How many crossings does the default layout draw?
attr(check_crossings(graphr(sw)), "total")
# How many slopes does a circle draw, and how evenly does it draw its ties?
attr(check_slopes(circled), "distinct")
attr(check_lengths(circled), "cv")
```

```
# Is there room between the ties that meet at a node?
attr(check_angles(circled), "min")
# And where the layout has rows, how long and how straight are its ties?
thrones <- manynet::to_layer(manynet::fict_thrones, "parent")
drawn <- graphr(thrones)
attr(check_span(drawn), "total")
attr(check_offset(drawn), "mean")
```

`completion` *Completing argument values as you type*

Description

`graphr()` and its relatives take the names of node and tie variables, layouts, and themes as strings, which means remembering what a network holds. This offers those names to RStudio's completion system, so that writing `graphr(fict_lotr, node_color = "` and pressing Tab lists the variables `fict_lotr` holds, `layout = "` lists the layouts available, and so on for every argument with a known set of values.

This is off until it is asked for, because it works by replacing one of RStudio's internal functions. That function is not part of a public interface, so a future version of RStudio can change it. Nothing else about completion changes: any line that is not one of these calls is passed to RStudio untouched, as is any line this cannot make sense of.

`stocnet_completion(FALSE)` puts RStudio's function back.

Usage

```
stocnet_completion(activate, persist = FALSE)

set_completion(activate, persist = FALSE)
```

Arguments

`activate`	Logical, by default TRUE. If TRUE, completion of argument values is switched on. If FALSE, RStudio's own completions are restored. If missing, the current state is reported and nothing changes.
`persist`	Logical, by default FALSE. If TRUE, the choice is remembered across sessions, by writing it to the user's configuration directory (see `tools::R_user_dir()`). Nothing is written to disk unless this is set explicitly. Use `stocnet_completion(persist = FALSE)` when activating to forget a previously persisted choice.

Value

Invisibly, TRUE where completion is now active and FALSE otherwise. Called for the effect it has on the IDE.

See Also

Other mapping: check_layout, layout_concentric(), layout_configuration(), layout_correspondence(), layout_layered(), layout_levels(), layout_matching(), layout_scaling(), layout_spatial(), layout_valence(), plot_graphr, plot_graphs, plot_grapht

Examples

```
## Not run:
# In RStudio, switch completion on for this session:
stocnet_completion()
# Then type graphr(fict_lotr, node_color = " and press Tab.
# To switch it off again:
stocnet_completion(FALSE)

## End(Not run)
```

`count_pages` *How many pages a paged diagnostic figure has*

Description

The page count of plot() on a per-term diagnostic, derivable **without rendering** so a loop can write every page.

A method that only discovers it is on the last page once it gets there cannot be scripted, and scripting is the case this exists for: fits go to a cluster, so a figure has to be producible with nobody at a screen to press return.

Usage

```
count_pages(x, nrow = 2, ncol = 2)
```

Arguments

x	a diagnostic object with one panel per term – as returned by test_gof(), test_time(), diagnose_onset(), or a fitted goldfish model.
nrow, ncol	panels per page, matching what will be passed to plot().

Value

A single integer, at least 1.

Examples

```
count_pages(goldfish_gof)
```

`layout_concentric` *Concentric layout*

Description

The "concentric" layout places the nodes on one or more circles, with each group of nodes on a circle of its own, and the groups ordered around those circles so that adjacent nodes are drawn close together. Where one group holds a single node, that node occupies the centre.

Usage

```
layout_concentric(
  .data,
  membership,
  radius = NULL,
  order.by = NULL,
  circular = FALSE,
  times = 1000
)

layout_tbl_graph_concentric(
  .data,
  membership,
  radius = NULL,
  order.by = NULL,
  circular = FALSE,
  times = 1000
)
```

Arguments

`.data`	Some {manynet} compatible network data.
`membership`	A node attribute or a vector to draw concentric circles. By default this is the two modes of a two-mode network.
`radius`	A vector of radii at which the concentric circles should be located. By default this is equal placement around an empty centre, unless one (the core) is a single node, in which case this node occupies the centre of the graph.
`order.by`	An attribute label indicating the (decreasing) order for the nodes around the circles. By default ordering is given by a bipartite placement that reduces the number of edge crossings.
`circular`	Should the layout be transformed into a radial representation. Only possible for some layouts. Defaults to FALSE. Required for {ggraph} compatibility.
`times`	Maximum number of iterations, where appropriate. Required for {ggraph} compatibility, and ignored by the layouts that do not iterate.

Value

Returns a table of nodes' x and y coordinates.

Source

Diego Diez, Andrew P. Hutchins and Diego Miranda-Saavedra. 2014. "Systematic identification of transcriptional regulatory modules from protein-protein interaction networks". *Nucleic Acids Research*, 42 (1) e6.

See Also

Other mapping: `check_layout`, `completion`, `layout_configuration()`, `layout_correspondence()`, `layout_layered()`, `layout_levels()`, `layout_matching()`, `layout_scaling()`, `layout_spatial()`, `layout_valence()`, `plot_graphr`, `plot_graphs`, `plot_grapht`

Examples

```
#graphr(ison_southern_women, layout = "concentric", membership = "type",
#           node_color = "type", node_size = 3)
```

---

layout_configuration *Layout algorithms based on configurational positions*

Description

Configurational layouts locate nodes at symmetric coordinates to help illustrate particular configurations. Currently configurational layouts are available for 2-6 nodes. The "configuration" layout will choose the appropriate configurational layout automatically.

Usage

```
layout_configuration(.data, circular = TRUE, times = 1)

layout_tbl_graph_configuration(.data, circular = TRUE, times = 1)

layout_dyad(.data, circular = TRUE, times = 1)

layout_triad(.data, circular = TRUE, times = 1)

layout_tetrad(.data, circular = TRUE, times = 1)

layout_pentad(.data, circular = TRUE, times = 1)

layout_hexad(.data, circular = TRUE, times = 1)
```

Arguments

.data	Some {manynet} compatible network data.
circular	Should the layout be transformed into a radial representation. Only possible for some layouts. Defaults to FALSE. Required for {ggraph} compatibility.
times	Maximum number of iterations, where appropriate. Required for {ggraph} compatibility, and ignored by the layouts that do not iterate.

Value

Returns a table of nodes' x and y coordinates.

See Also

Other mapping: check_layout, completion, layout_concentric(), layout_correspondence(), layout_layered(), layout_levels(), layout_matching(), layout_scaling(), layout_spatial(), layout_valence(), plot_graphr, plot_graphs, plot_grapht

Examples

```
# "configuration" picks the layout matching the number of nodes
graphr(manynet::create_ring(4), layout = "configuration")
# the specific configurations are also available as functions
layout_tetrad(manynet::create_ring(4))
```

---

layout_correspondence *Correspondence layout*

Description

The "correspondence" layout places nodes by correspondence analysis, so that two nodes are drawn together where they have similar ties. Where the "scaling" layout reads the paths between nodes, this one reads the profile of each node's ties, and so two nodes with no tie between them can still be drawn together if they are tied to the same others.

This is the usual way to draw a two-mode network, since correspondence analysis takes a rectangular table and places its rows and its columns in one space. Both modes are therefore drawn on one pair of axes.

Like the "scaling" layout, the coordinates can be read, and so this layout draws labelled axes at a fixed ratio. Each axis is labelled with the share of the network's inertia that the dimension holds.

Usage

```
layout_correspondence(
  .data,
  direction = c("all", "out", "in"),
  double = FALSE,
  circular = FALSE,
```

```
  times = 1
)

layout_tbl_graph_correspondence(
  .data,
  direction = c("all", "out", "in"),
  double = FALSE,
  circular = FALSE,
  times = 1
)
```

Arguments

.data	Some {manynet} compatible network data.
direction	Which ties to read for a directed network, as one of "all", "out", or "in". By default this is "all", which reads a tie in either direction, so that each node has one position. "out" places each node by the ties it sends, and "in" by the ties it receives. This is ignored where the network is undirected or two-mode.
double	Whether to split each tie into a positive and a negative part, so that a signed network can be drawn. By default this is FALSE, and a signed network is not drawn, since correspondence analysis is not defined for a negative tie.
circular	Should the layout be transformed into a radial representation. Only possible for some layouts. Defaults to FALSE. Required for {ggraph} compatibility.
times	Maximum number of iterations, where appropriate. Required for {ggraph} compatibility, and ignored by the layouts that do not iterate.

Details

Correspondence analysis divides the ties of each node by how many ties that node has, and so places nodes by the shape of their ties rather than by how many they have. The distance drawn is the chi-square distance between two such profiles.

A two-mode network is read as its incidence matrix, one row for each node of the first mode and one column for each of the second. A one-mode network is read as its adjacency matrix instead, as is a multimodal network that has ties within its modes as well as between them, so that no tie is dropped.

Tie weights are read as they are, since correspondence analysis was built for counts and a weight counts in the same way. A negative weight has no such reading, which is why a signed network needs double = TRUE. That stacks the positive network and the negative network side by side, doubling the width of the table, so that a node is placed by both who it is tied to positively and who it is tied to negatively. A pair of nodes with no tie between them counts in neither half.

Value

Returns a table of nodes' x and y coordinates.

Reading the plot

Two nodes of the same mode drawn together have similar ties. A node drawn near the origin has a profile close to the average, or is held poorly by the two dimensions drawn: these are not the same thing, and `graphr()` names the nodes for which it is the second.

A node of one mode drawn near a node of the other mode is *not* necessarily tied to it. Only the distances within a mode can be read this way.

Where a network runs along one strong gradient, correspondence analysis draws it as an arch rather than as a line. This is expected of the method, and the second dimension then repeats the first rather than adding to it.

Where a network is disconnected, the first dimensions merely separate its components, and say little about the nodes within them.

Reading the inertia

The share of inertia a dimension holds is not a share of variance explained, and does not have a fixed ceiling to be read against. It is a share of however many dimensions the table has, which `attr(x, "fit")$scree` reports in full. Two dimensions of a table that has twelve start from a base of a sixth; two of a table that has thirty start from a base of a fifteenth. Compare the share drawn against that base rather than against 100%, and note that this can reverse the ranking the raw shares suggest. Bear in mind that an even share is a lenient base, since inertia is never spread evenly; the broken stick model asks what the dimensions would hold if the inertia were divided at random, and is the harder test. Neither is a standard statistic, and neither carries a threshold, so read them as a check on the raw share rather than as a verdict. `graphr()` says so at the console where two dimensions hold no more than a random division of the inertia would give them. To choose a number of dimensions properly, see Lorenzo-Seva (2011).

These shares need no correction. The Benzécri correction, and Greenacre's adjusted version of it, exist because the indicator matrix that *multiple* correspondence analysis is run on invents dimensions that deflate every share. This layout runs simple correspondence analysis on one two-way table, which invents nothing, so the shares reported are already exact.

Source

Greenacre, Michael. 2017. *Correspondence Analysis in Practice*, 3rd ed. Boca Raton: Chapman and Hall. doi:10.1201/9781315369983

Lorenzo-Seva, Urbano. 2011. "Horn's parallel analysis for selecting the number of dimensions in correspondence analysis", *Methodology* 7(3): 96-105. doi:10.1027/16142241/a000027

Constantine, A.G., and John C. Gower. 1978. "Graphical representation of asymmetric matrices", *Journal of the Royal Statistical Society C* 27(3): 297-304. doi:10.2307/2347234

See Also

Other mapping: `check_layout`, `completion`, `layout_concentric()`, `layout_configuration()`, `layout_layered()`, `layout_levels()`, `layout_matching()`, `layout_scaling()`, `layout_spatial()`, `layout_valence()`, `plot_graphr`, `plot_graphs`, `plot_grapht`

Examples

```
graphr(manynet::ison_southern_women, layout = "correspondence")
```

`layout_layered` *Layered layouts*

Description

These algorithms assign each node to a layer, which becomes one axis, and a position within that layer, which becomes the other. They are recommended for use with `graphr()` or {ggraph}, and suit two-mode networks and directed acyclic networks.

The four layouts are one engine drawn four ways, and differ only in which axis carries the layers and in how each layer is spread out:

	Layers stacked flat	Layers standing up
`alignment = "straight"`	"layered"	"lineage"
`alignment = "rungs"`	"railway"	"ladder"

That is, the "layered" layout places the first node set along the bottom and the second node set along the top, sequenced and spaced as necessary to minimise tie overlap. The "lineage" layout is the same layout with the axes exchanged, so that successive layers run left to right rather than bottom to top. The "railway" and "ladder" layouts are "layered" and "lineage" with every layer given the same spacing, so that the nodes line up across the layers like the rails and rungs the names describe.

Usage

```
layout_layered(
  .data,
  center = NULL,
  ranks = c("tight", "generation", "compact"),
  alignment = c("straight", "rungs"),
  circular = FALSE,
  times = 1000
)

layout_tbl_graph_layered(
  .data,
  center = NULL,
  ranks = c("tight", "generation", "compact"),
  alignment = c("straight", "rungs"),
  circular = FALSE,
  times = 1000
)
```

```
layout_lineage(
  .data,
  ranks = c("tight", "generation", "compact"),
  alignment = c("straight", "rungs"),
  circular = FALSE,
  times = 1000,
  rank = NULL
)

layout_tbl_graph_lineage(
  .data,
  ranks = c("tight", "generation", "compact"),
  alignment = c("straight", "rungs"),
  circular = FALSE,
  times = 1000,
  rank = NULL
)

layout_railway(
  .data,
  ranks = c("tight", "generation", "compact"),
  circular = FALSE,
  times = 1000
)

layout_tbl_graph_railway(
  .data,
  ranks = c("tight", "generation", "compact"),
  circular = FALSE,
  times = 1000
)

layout_ladder(
  .data,
  ranks = c("tight", "generation", "compact"),
  circular = FALSE,
  times = 1000
)

layout_tbl_graph_ladder(
  .data,
  ranks = c("tight", "generation", "compact"),
  circular = FALSE,
  times = 1000
)
```

Arguments

.data	Some {manynet} compatible network data.

`center` Further split a "layered" layout by declaring the "center" argument as the "events", "actors", or by declaring a node name. Defaults to NULL.

`ranks` How the layers are assigned: "tight" (the default) chooses the layers that make the total tie length as short as possible, while still pointing every tie down at least one layer; "generation" ranks each node by its distance from a root, so that a layer is a generation, at the cost of some longer ties; "compact" asks `igraph::layout_with_sugiyama()` for the layers. The first two need an acyclic network, and fall back to "compact" where the network is not. Ignored for a two-mode network, whose layers are its modes.

A node attribute can be given here instead, either as the name of a numeric node attribute or as a numeric vector as long as the network has nodes. Then the layers are those values, and nodes are placed along that axis in proportion to them rather than at even steps, so that a network of dated nodes is drawn as a timeline. The values run in the same direction as the layers the engine works out: down the page in a "layered" or "railway" layout, and left to right in a "lineage" or "ladder" layout, so that the smallest value comes first. A "lineage" given values lays out all of its nodes together, rather than one component at a time, so that an isolate or a small component sits in the column of its own value. `graphr()` draws it as a timeline. A node that a tie would run straight through, other than the two nodes the tie joins, is moved off that tie, so that several works citing one earlier work can be told from a chain of works each citing the one before. Two tied nodes with the same value are put next to each other, and nodes are moved within their layer where that leaves fewer ties crossing. That last step is left out of larger networks, where the number of nodes times the number of ties is more than 5000, to save time.

`alignment` How each layer is spread out: "straight" (the default) draws the ties as close to straight as the ordering allows, which groups the nodes that belong together; "rungs" gives every layer the same integer spacing, so that the nodes line up across the layers.

`circular` Should the layout be transformed into a radial representation. Only possible for some layouts. Defaults to FALSE. Required for {ggraph} compatibility.

`times` Maximum number of iterations, where appropriate. Required for {ggraph} compatibility, and ignored by the layouts that do not iterate.

`rank` Deprecated. Use `ranks` instead, which now takes a node attribute as well as a method.

Value

Returns a table of nodes' x and y coordinates.

See Also

Other mapping: `check_layout`, `completion`, `layout_concentric()`, `layout_configuration()`, `layout_correspondence()`, `layout_levels()`, `layout_matching()`, `layout_scaling()`, `layout_spatial()`, `layout_valence()`, `plot_graphr`, `plot_graphs`, `plot_grapht`

Examples

```
#graphr(ison_southern_women, layout = "layered", center = "events",
#           node_color = "type", node_size = 3)
#graphr(ison_southern_women, layout = "lineage")
# ison_adolescents |>
#   mutate_nodes(year = rep(c(1985, 1990, 1995, 2000), times = 2)) |>
#   graphr(layout = "lineage", ranks = "year")
```

`layout_levels` *Levels layout*

Description

The "levels" layout draws each level of a multilevel network as a plane of its own, projected at an angle, with the ties within each level drawn on its plane and the ties between levels drawn between them.

Note that {graphlayouts} offers a layout of the same idea under the name "multilevel". This one is named for its `level` argument.

Usage

```
layout_levels(
  .data,
  level,
  method = c("all", "separate", "fix1", "fix2"),
  circular = FALSE,
  times = 1,
  alpha = 25,
  beta = 45,
  FUN1 = graphlayouts::layout_with_stress,
  FUN2 = graphlayouts::layout_with_stress
)

layout_tbl_graph_levels(
  .data,
  level,
  method = c("all", "separate", "fix1", "fix2"),
  circular = FALSE,
  times = 1,
  alpha = 25,
  beta = 45,
  FUN1 = graphlayouts::layout_with_stress,
  FUN2 = graphlayouts::layout_with_stress
)
```

Arguments

.data	Some {manynet} compatible network data.
level	A node attribute or a vector to hierarchically order levels. By default the levels are those already recorded in a "lvl" node attribute, as manynet::to_multilevel() writes, or, for a two-mode network, the two modes, with whichever mode holds the ties within itself placed at the first level.
method	How the levels should be laid out: "all" (the default) lays every level out at once, "separate" lays each level out independently, and "fix1" and "fix2" lay out the first or second level respectively and derive the other from it. Note that all but "all" require ties within the levels they lay out.
circular	Should the layout be transformed into a radial representation. Only possible for some layouts. Defaults to FALSE. Required for {ggraph} compatibility.
times	Maximum number of iterations, where appropriate. Required for {ggraph} compatibility, and ignored by the layouts that do not iterate.
alpha, beta	The angles, in degrees, at which the levels are projected onto the plane.
FUN1, FUN2	The layout functions used for the first and second levels respectively by the "separate", "fix1" and "fix2" methods. By default both are graphlayouts::layout_with_stress().

Value

Returns a table of nodes' x and y coordinates.

See Also

Other mapping: check_layout, completion, layout_concentric(), layout_configuration(), layout_correspondence(), layout_layered(), layout_matching(), layout_scaling(), layout_spatial(), layout_valence(), plot_graphr, plot_graphs, plot_grapht

Examples

```
# fict_marvel interlocks a one-mode layer of ties among its characters
# with a two-mode layer of their affiliations, so it is laid out this way
# by default; the levels need not be named.
graphr(manynet::fict_marvel, labels = FALSE)
```

---

layout_matching *Matching layout*

Description

This layout works to position nodes opposite their matching nodes. See manynet::to_matching() for more details on the matching procedure.

Usage

```
layout_matching(.data, center = NULL, circular = FALSE, times = 1)

layout_tbl_graph_matching(.data, center = NULL, circular = FALSE, times = 1)
```

Arguments

.data	Some {manynet} compatible network data.
center	Required for {ggraph} compatibility, and not used here.
circular	Should the layout be transformed into a radial representation. Only possible for some layouts. Defaults to FALSE. Required for {ggraph} compatibility.
times	Maximum number of iterations, where appropriate. Required for {ggraph} compatibility, and ignored by the layouts that do not iterate.

Value

Returns a table of nodes' x and y coordinates.

See Also

Other mapping: check_layout, completion, layout_concentric(), layout_configuration(), layout_correspondence(), layout_layered(), layout_levels(), layout_scaling(), layout_spatial(), layout_valence(), plot_graphr, plot_graphs, plot_grapht

---

layout_scaling *Scaling layout*

Description

The "scaling" layout places nodes by multidimensional scaling, so that the distance drawn between two nodes approximates the number of steps of the shortest path between them. Unlike a force-directed layout, then, the coordinates can be read, and so this layout draws labelled axes, at a fixed ratio so that the two axes share one scale.

Which algorithm is used depends on the size of the network. Up to a hundred nodes, classical multidimensional scaling is used, as `igraph::layout_with_mds()` offers it. Above that, or where `pivots` is given, pivot multidimensional scaling is used instead, as `graphlayouts::layout_with_pmds()` offers it, which approximates the same solution from a sample of the nodes and is much the faster for a large network. Note that "mds" and "pmds" remain available as layouts in their own right, though "pmds" then requires its own `pivots`.

Two dimensions rarely hold every path distance of a network at once, so `graphr()` captions the plot with how well this one does: see `check_stress()` for how to read the score.

Usage

```
layout_scaling(.data, pivots = NULL, circular = FALSE, times = 1)

layout_tbl_graph_scaling(.data, pivots = NULL, circular = FALSE, times = 1)
```

Arguments

.data	Some {manynet} compatible network data.
pivots	The number of nodes to approximate the scaling from. By default this is NULL, which uses every node where the network has no more than a hundred, and samples the nodes otherwise. Giving a number selects the pivot algorithm whatever the size of network.
circular	Should the layout be transformed into a radial representation. Only possible for some layouts. Defaults to FALSE. Required for {ggraph} compatibility.
times	Maximum number of iterations, where appropriate. Required for {ggraph} compatibility, and ignored by the layouts that do not iterate.

Details

The distances scaled are those of the unweighted network, that is, the number of ties on the shortest path between two nodes. Tie weights are ignored, since the interpretation of a drawn distance is then the same whatever the network, and since a signed network has no shortest paths to speak of.

Where a network is disconnected, there is no path between its components, and so no distance to scale. Each component is laid out and the components are then placed beside one another, and the fit is reported over the pairs of nodes that a path does connect.

Value

Returns a table of nodes' x and y coordinates.

Source

Kruskal, Joseph B. 1964. "Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis", *Psychometrika* 29(1): 1-27. doi:10.1007/BF02289565

Brandes, Ulrik, and Christian Pich. 2007. "Eigensolver methods for progressive multidimensional scaling of large data", in *Graph Drawing*, 42-53. doi:10.1007/9783540709046_6

See Also

Other mapping: check_layout, completion, layout_concentric(), layout_configuration(), layout_correspondence(), layout_layered(), layout_levels(), layout_matching(), layout_spatial(), layout_valence(), plot_graphr, plot_graphs, plot_grapht

Examples

```
graphr(manynet::ison_southern_women, layout = "scaling")
```

`layout_spatial` *Spatial layout*

Description

The "spatial" layout places each node at its own position, as a spatial network records it (see `manynet::is_spatial()`): in 'x' and 'y' node attributes, or in 'lon' and 'lat' (or 'longitude' and 'latitude'). `graphr()` uses it by default for a spatial network, such as one read by `manynet::read_geojson()`. The two axes share one scale, so that a distance drawn is the same whichever way it runs.

Longitudes and latitudes are drawn on an equirectangular projection centred on the network's mean latitude: each longitude is multiplied by the cosine of that latitude, so that a degree of longitude is drawn as long as it is on the ground there. This keeps the shape of a city or a region, though not of a continent.

A node without a position is placed at the centre of those with one.

Usage

```
layout_spatial(.data, circular = FALSE, times = 1)

layout_tbl_graph_spatial(.data, circular = FALSE, times = 1)
```

Arguments

`.data`	Some {manynet} compatible network data.
`circular`	Should the layout be transformed into a radial representation. Only possible for some layouts. Defaults to FALSE. Required for {ggraph} compatibility.
`times`	Maximum number of iterations, where appropriate. Required for {ggraph} compatibility, and ignored by the layouts that do not iterate.

Value

Returns a table of nodes' x and y coordinates.

See Also

Other mapping: `check_layout`, `completion`, `layout_concentric()`, `layout_configuration()`, `layout_correspondence()`, `layout_layered()`, `layout_levels()`, `layout_matching()`, `layout_scaling()`, `layout_valence()`, `plot_graphr`, `plot_graphs`, `plot_grapht`

Examples

```
net <- manynet::create_ring(4) |>
  manynet::mutate_nodes(x = c(0, 1, 1, 0), y = c(0, 0, 1, 1))
graphr(net)
```

`layout_valence` *Valence layout*

Description

The "valence" layout places the nodes of a signed network so that positively tied nodes are drawn together and negatively tied nodes apart.

Usage

```
layout_valence(
  .data,
  times = 500,
  center = NULL,
  circular = FALSE,
  repulsion_coef = 1,
  attraction_coef = 0.05
)

layout_tbl_graph_valence(
  .data,
  times = 500,
  center = NULL,
  circular = FALSE,
  repulsion_coef = 1,
  attraction_coef = 0.05
)
```

Arguments

`.data`	Some {manynet} compatible network data.
`times`	Maximum number of iterations, where appropriate. Required for {ggraph} compatibility, and ignored by the layouts that do not iterate.
`center`	Required for {ggraph} compatibility, and not used here.
`circular`	Should the layout be transformed into a radial representation. Only possible for some layouts. Defaults to FALSE. Required for {ggraph} compatibility.
`repulsion_coef`	Coefficient for global repulsion force. Default is 1.
`attraction_coef`	Coefficient for edge-based attraction/repulsion force. Default is 0.05.

Value

Returns a table of nodes' x and y coordinates.

See Also

Other mapping: check_layout, completion, layout_concentric(), layout_configuration(), layout_correspondence(), layout_layered(), layout_levels(), layout_matching(), layout_scaling(), layout_spatial(), plot_graphr, plot_graphs, plot_grapht

Examples

```
edges <- data.frame(
  from = c("A", "B", "C", "D"),
  to   = c("B", "C", "D", "A"),
  weight = c(2, 3, 1, 4),
  sign = c(1, -1, 1, -1)  # 1 = positive, -1 = negative
  )
graphr(as_igraph(edges), layout="valence")
```

---

list_fonts *Listing the fonts available to R*

Description

list_fonts() reports the font families that R can currently see, which is what a theme's preferred fonts are matched against. A font that is installed on the system but missing from this list is not available to R yet; see the Fonts section of theme_set for how to make it so.

Usage

```
list_fonts(pattern = NULL)
```

Arguments

pattern	Optionally, a string with which to filter the font families returned, matched without regard to case. For example, list_fonts("sans") returns every family whose name includes "sans".

Value

A vector of font family names.

See Also

Other themes: check_colors, theme_colorblind, theme_medium, theme_set

Examples

```
head(list_fonts())
```

made_earlier *Precooked results for demonstrating plotting*

Description

These are all pre-cooked results objects, saved here to save time in testing and demonstrating how autograph plots look.

Usage

```
data(res_infernet_reg)

data(res_netrics_robust)

data(res_migraph_reg)

data(res_migraph_test)

data(res_migraph_diff)

data(res_manynet_diff)

data(siena_gof)

data(siena_influence)

data(siena_selection)

data(monan_conv)

data(monan_gof)

data(ergm_gof)

data(goldfish_outliers)

data(goldfish_changepoints)

data(goldfish_margins)

data(goldfish_gof)

data(goldfish_time)

data(goldfish_onset)

data(goldfish_fit)
```

Format

An object of class `netreg_lm` (inherits from `netreg`) of length 15.

An object of class `list` of length 2.

An object of class `netlm` of length 15.

An object of class `network_test` of length 9.

An object of class `diffs_model` (inherits from `data.frame`) with 20 rows and 11 columns.

An object of class `diff_model` (inherits from `tbl_df`, `tbl`, `data.frame`) with 4 rows and 10 columns.

An object of class `sienaGOF` of length 1.

An object of class `influenceTable` (inherits from `data.frame`) with 25 rows and 4 columns.

An object of class `selectionTable` (inherits from `data.frame`) with 25 rows and 4 columns.

An object of class `traces.monan` of length 3.

An object of class `gof.stats.monan` of length 2.

An object of class `gof.ergm` (inherits from `gof`) of length 30.

An object of class `goldfishOutliers` (inherits from `tbl_df`, `tbl`, `data.frame`) with 439 rows and 11 columns.

An object of class `goldfishChangepoints` (inherits from `tbl_df`, `tbl`, `data.frame`) with 115 rows and 10 columns.

An object of class `goldfishMargins` (inherits from `tbl_df`, `tbl`, `data.frame`) with 308 rows and 5 columns.

An object of class `goldfishGOF` (inherits from `list`) of length 3.

An object of class `goldfishTimeTest` (inherits from `list`) of length 3.

An object of class `goldfishOnset` (inherits from `list`) of length 3.

An object of class `goldfishFit` of length 31.

Details

`res_netrics_robust` is a list of two robustness results from `netrics::net_by_robustness()`, each a "network_measure" with a "curve" attribute, on a 50-node scale-free network: `failure`, robustness to random failure (100 simulations), and `attack`, robustness to an adaptive attack on degree. It was made with:

```
set.seed(1234)
g <- manynet::generate_scalefree(50, 1)
res_netrics_robust <- list(
  failure = netrics::net_by_robustness(g, times = 100),
  attack = netrics::net_by_robustness(g,
             node_measure = netrics::node_by_degree))
```

map_measure *Plotting logical marks Plotting numeric measures*

Description

These functions plot distributions for node, tie, and network measures, as defined in the {manynet} package.

Usage

```
## S3 method for class 'node_measure'
plot(x, type = c("h", "d"), ...)

## S3 method for class 'tie_measure'
plot(x, type = c("h", "d"), ...)

## S3 method for class 'network_measures'
plot(x, ...)

## S3 method for class 'network_measure'
plot(x, ..., labels = NULL)
```

Arguments

x	An object of "node_measure", "tie_measure", "network_measure", or "network_measures" class.
type	For node and tie measures, whether the plot should be "h" a histogram or "d" a density plot. By default "h".
...	Other arguments to be passed on.
labels	For network measures with a curve, such as robustness, an optional character vector of labels, one for each result. By default, the names given in the call are used, and otherwise the measure's own name.

Value

`plot.node_measure()` and `plot.tie_measure()` returns a histogram and/or density plot of the distribution of the measure.

`plot.network_measures()` returns a plot of the measure traced over time.

`plot.network_measure()` returns a plot of the curve that some network measures carry, such as the robustness curve returned by `netrics::net_by_robustness()`. The share of nodes in the largest component is drawn against the share of nodes removed. The dashed line is the best case, where every remaining node stays in one component, and the area under it is the largest robustness possible, 0.5. Several results can be passed together to compare them in one plot. A measure without a curve is a single value, so nothing is plotted.

Examples

```
plot(netrics::node_by_deg(ison_karateka))
plot(netrics::tie_by_betweenness(ison_karateka))
plot(res_netrics_robust$failure)
plot(res_netrics_robust$failure,
     res_netrics_robust$attack)
```

`map_member` *Plotting categorical memberships*

Description

This plotting method operates on "node_member" class objects from the {manynet} package, plotting the dendrogram of their membership.

Usage

```
## S3 method for class 'node_member'
plot(x, ...)

## S3 method for class 'matrix'
plot(x, ..., membership = NULL)
```

Arguments

`x`	An object of "node_member" class, for example as a result of running `netrics::node_in_community()`.
`...`	Other arguments to be passed on.
`membership`	A "node_member" membership vector.

Value

`plot.node_member()` returns a dendrogram, with labels colored to indicate the different clusters, and with the optimal cutpoint shown by a dashed highlight line.

`plot.matrix()` returns a plot of an adjacency or incidency matrix, potentially with the rows and columns reordered to illustrate an additional membership vector.

Examples

```
plot(netrics::node_in_walktrap(ison_southern_women, "e"))
plot(as_matrix(ison_adolescents),
  membership = netrics::node_in_walktrap(ison_adolescents, "e"))
plot(as_matrix(ison_southern_women),
  membership = netrics::node_in_walktrap(ison_southern_women, "e"))
```

map_motifs *Plotting tabular motifs*

Description

These functions will plot graphs of the motifs used in a vector of results of e.g. a triad census.

Usage

```
## S3 method for class 'node_motif'
plot(x, ...)

## S3 method for class 'network_motif'
plot(x, ...)
```

Arguments

x	An object of "node_motif" class, e.g. resulting from a call to `manynet::node_by_triad()`.
...	Other arguments to be passed on.

Value

`plot.node_motif()` returns a set of graphs that illustrate the motifs mentioned in the results from a node_motif function in {manynet}.

`plot.network_motif()` returns a set of graphs that illustrate the motifs mentioned in the results from a net_motif function in {manynet}.

mdate_scales *Scales for messy dates*

Description

These scales place a 'messy' date, of the `mdate` class that {messydates} defines, on the x or y axis of a {ggplot2} plot. The dates are then spaced by how far apart they are, and the axis is marked with date breaks. Without them, {ggplot2} reads an `mdate` column as a character vector and draws a discrete axis, which orders the dates as text and spaces them evenly.

Usage

```
scale_x_mdate(..., FUN = messydates::vmin)

scale_y_mdate(..., FUN = messydates::vmin)
```

Arguments

...	Arguments passed on to ggplot2::scale_x_date() or ggplot2::scale_y_date(), such as name, breaks, date_breaks, date_labels, or limits.
FUN	The function that resolves each messy date to the one date at which it is drawn. messydates::vmin by default, as for messydates::as.Date(), which also takes vmax, vmean, vmedian, vmodal, or vrandom.

Details

{ggplot2} chooses a scale from the class of the column, so a plot that maps an mdate column to x or y picks these scales up without naming them. Add the scale to the plot to pass FUN, date_breaks, or date_labels.

A position on an axis is a single point, but a messy date may be a range, a set, or an unspecified component, and so covers a span of dates. These scales therefore resolve each date to one date with FUN, as messydates::as.Date() does. To draw the span itself, resolve both of its ends in the mapping, as in aes(x = vmin(date), xend = vmax(date)) with geom_segment(); both ends are mdate vectors, so they share this scale.

Value

A {ggplot2} scale to add to a plot.

Examples

```
library(ggplot2)
dates <- messydates::as_messydate(c("2012-01-01", "2012-06", "2013~"))
df <- data.frame(date = dates, y = 1:3)
ggplot(df, aes(x = date, y = y)) + geom_point()
ggplot(df, aes(x = date, y = y)) + geom_point() +
  scale_x_mdate(FUN = messydates::vmax, date_labels = "%Y-%m")
```

---

model_mrqap *Plotting methods for MRQAP models*

Description

These plotting methods are for results obtained by fitting an MRQAP model. The S3 class "netreg" comes from {infernet}, and includes its "netreg_lm", "netreg_glm", and "netreg_css" results. The S3 classes "netlm" and "netlogit" come from the {sna} package or from earlier versions of {migraph}.

Usage

```
## S3 method for class 'netreg'
plot(x, ...)

## S3 method for class 'netlm'
plot(x, ...)

## S3 method for class 'netlogit'
plot(x, ...)
```

Arguments

x	An object obtained by fitting an MRQAP model to some data. For example, `infernet::net_regression()`.
...	Further arguments to be passed on to plot.

Value

A plot showing the location of observed statistics compared to the distribution of statistics from permuted networks. The vertical lines mark the 2.5% and 97.5% quantiles of each distribution. A term with no permuted statistics, such as the intercept where predictors are permuted, shows only its observed statistic.

Examples

```
# Here's something I cooked up with infernet earlier:
plot(res_infernet_reg)
# And with an earlier version of migraph:
plot(res_migraph_reg)
```

---

plot.diagnose_outliers

Plotting adequacy diagnostics

Description

These plotting methods are for diagnosing the adequacy of model specification, such as those used in goldfish. These plots are useful for identifying whether there might be significant outliers affecting the results, whether there is significant time heterogeneity, and which actors' activity the model does not reproduce.

Usage

```
## S3 method for class 'diagnose_outliers'
plot(x, ...)

## S3 method for class 'outliers.goldfish'
```

```
plot(x, ...)

## S3 method for class 'diagnose_changepoints'
plot(x, ...)

## S3 method for class 'changepoints.goldfish'
plot(x, ...)

## S3 method for class 'goldfishOutliers'
plot(x, ...)

## S3 method for class 'goldfishChangepoints'
plot(x, ...)

## S3 method for class 'goldfishMargins'
plot(x, ..., top = 25)

## S3 method for class 'goldfishGOF'
plot(x, ..., level = 0.95, page = NULL, nrow = 2, ncol = 2)

## S3 method for class 'goldfishTimeTest'
plot(x, ..., page = NULL, nrow = 2, ncol = 2)

## S3 method for class 'goldfishOnset'
plot(
  x,
  ...,
  view = c("both", "path", "accrual"),
  tolerance_band = TRUE,
  page = NULL,
  nrow = 2,
  ncol = 2
)
```

Arguments

x	An object of class goldfishOutliers, goldfishChangepoints, goldfishMargins, goldfishGOF, goldfishTimeTest or goldfishOnset, as returned by diagnose_outliers(), diagnose_changepoints(), margin_table(), test_gof(), test_time() and diagnose_onset() in goldfish.
...	Additional plotting parameters, currently unused.
top	The number of actors to draw, those furthest from the reference.
level	The confidence level of the reference bands, defaulting to 0.95. The band is the two-sided Kolmogorov quantile of the supremum of a Brownian bridge, which is the reference the event-clock p-value uses.
page	Which page to draw, for the per-term figures. NULL (the default) draws every panel in one figure, exactly as before. A number draws that page alone; a number

	past the last is an error naming the count. Use `count_pages()` to learn the count without rendering, so a loop can write every page with nobody at a screen.
`nrow`, `ncol`	Panels per page when `page` is given.
`view`	Which panels to draw: `"both"` (default), or `"path"` or `"accrual"` alone, which is the escape hatch when a model has too many coefficients for a composed figure to stay readable.
`tolerance_band`	Whether to draw each coefficient's stabilization band, the `+/- tolerance * std_error` corridor the path had to re-enter.

Details

`plot.diagnose_outliers()`, `plot.outliers.goldfish()`, `plot.diagnose_changepoints()` and `plot.changepoints.goldfish()` are aliases for `plot.goldfishOutliers()` and `plot.goldfishChangepoints()`, kept so that an object carrying one of the older class names plots as before. Each reads the columns the current methods read. They will be removed.

goldfish emits these objects plot-ready. Each is a tibble carrying the diagnostic metadata contract — which function produced it, which model and sub-model it came from, and the arguments that shape how it is read — so these methods take their series, their labels and their reference lines from the object rather than inferring them from the columns that happen to be present.

The `.series` column is the series the diagnostic actually analysed: the per-interval log-likelihood by default, and the selected term's own series when the diagnostic was called with `effect =`. It is `NA` on the intervals that took no part, which on a rate or REM fit are the right-censored ones.

`plot.goldfishMargins()` shows each actor's observed activity against what the model expected of them. Which comparison it draws follows the scales the fit's model class defines, which the object records: where a compensator is defined (the exact-time sub-models) the difference `observed - expected_count` is the per-actor martingale residual, read against zero; on the multinomial sub-models, which have no exposure-time term and so no compensator, the ratio `observed / expected_probability` is a calibration ratio, read against one.

These are descriptives rather than per-actor tests: the differences are plug-in quantities and are negatively correlated across actors. Read the plot as a map screening for unmodelled actor heterogeneity.

A node set large enough to make one row per actor unreadable is the ordinary case, so only the `top` actors furthest from the reference are drawn, and the subtitle says how many were left out. Actors are ranked by their largest deviation over the roles they appear in, so an actor kept for one margin keeps the other beside it. Pass `top = Inf` for all of them.

`plot.goldfishGOF()` draws each effect's standardized cumulative score process against the Brownian-bridge bands its p-value was read from. At the maximum the per-event scores sum to zero, so every path starts and ends at zero; under a correctly specified model it is a bridge, and a path that wanders outside the bands is an effect whose contribution is concentrated somewhere in the sequence.

The x axis is the object's own process-time axis, taken from its `u` column and labelled by the `clock` it records. This is not a presentational detail: the bands are valid on whichever clock produced the process, and re-deriving an event-index axis here would draw the path on one clock and the reference on another. On the information clock the spacing of the steps is itself the diagnostic — a path that crosses most of the axis in a few steps is an effect whose information arrives late.

`plot.goldfishTimeTest()` draws the scaled Schoenfeld residuals of each tested effect against time, with a smooth and the fitted estimate as the reference. A residual scatter is centred on the

coefficient the model estimated; a smooth that drifts away from that line over the sequence is the coefficient failing to be constant, which is what the test's p-value states formally.

Under `method = "periods"` the intervals are coloured by their period, so the regimes the test compared are visible against the same scatter.

`plot.goldfishOnset()` composes two panels: each coefficient's leave-the-first-`m`-events-out path, and the share of the model's information those events delivered.

Both panels are **windowed on the excursion rather than the sequence**, because the full range is mostly bridge tail — the path returns to the estimate by construction, so drawing all of it squashes the part being read into a few percent of the axis. Each coefficient gets its own window and its own x scale, since coefficients settle at very different points and a window shared across facets re-creates the squashing it exists to prevent. A coefficient whose path never left its band takes the full range, there being no excursion to window on.

The accrual panel is drawn full-range with the onset window shaded, and carries the proportional diagonal `y = x / n`. Without the diagonal a monotone curve from 0 to 1 says nothing: the signal is the *departure* from proportional, which is what makes an opening segment that carries little information visible.

Coefficients held fixed through `offset()` are not drawn. Their path is a flat line at the imposed value by construction.

Value

A ggplot object.

Examples

```
plot(goldfish_outliers)
plot(goldfish_changepoints)
plot(goldfish_margins)
plot(goldfish_gof)
plot(goldfish_time)
plot(goldfish_onset)
```

`plot.diffusion` *Plotting diffusion models*

Description

Plotting diffusion models

Usage

```
## S3 method for class 'diff_model'
plot(x, ..., all_steps = TRUE)

## S3 method for class 'diffs_model'
plot(x, ..., observed = NULL, level = 0.95)
```

```
## S3 method for class 'diffusionSweep'
plot(x, ..., level = 0.95)

## S3 method for class 'learn_model'
plot(x, ...)

## S3 method for class 'diffusionModel'
plot(x, ..., all_steps = TRUE)
```

Arguments

x	A "diff_model", "diffs_model", or "diffusionSweep" class of object. E.g. as a result from `manynet::play_diffusion()`.
...	Other arguments to be passed.
all_steps	Whether all steps should be plotted or just those where there is change in the distributions.
observed	Optionally, an observed diffusion to draw over the simulated diffusions: a network with a record of a diffusion, as from `manynet::play_diffusion()`, or a 'diffusionModel', as from `manynet::as_diffusion()`.
level	The share of the simulated diffusions that the band holds at each step. By default 0.95.

Details

`plot.diff_model()` draws one line for each value of the attribute that the diffusion changes, and bars for the nodes that become infectious at each step. A diffusion in the values S, E, I, R, and D names them "Susceptible", "Exposed", "Infected", "Recovered", and "Dead". A diffusion in values of its own, as from `manynet::play_diffusion(compartments = )`, names the lines by these values. The roles in `attr(x, "compartments")` give the colours: blue for S, yellow for E, red for I, green for R, and black for D. The bars count the new nodes in the first value of I.

`plot.diffs_model()` draws the simulated diffusions, as from `infernet::play_diffusions()`. Without `observed`, it draws a smoothed line for each value. With `observed`, it draws the proportion of the nodes in the infectious value at each step: a band that holds the middle `level` of the simulated diffusions, a dashed line for their median, and the observed diffusion over them. This is the curve that `infernet::test_gof()` compares. A diffusion that ends before the others keeps its last value until the last step, as in `infernet::test_gof()`.

`plot.diffusionSweep()` draws the share of the nodes that were infected for each setting of a sweep, as from `infernet::sweep_diffusion()`. The first argument in `vary` is on the horizontal axis. A sweep compares settings, so each setting is a bar, even where the settings are numbers: the median of its runs, labelled with its value where there are 12 bars or fewer, with a line for the middle `level` of the runs where they differ. Each value of any other argument in `vary` has a panel of its own. A node that is reinfected counts again, so the share can be above 100%.

`plot.diffusionModel()` draws a 'diffusionModel', as from `manynet::as_diffusion()`, as `plot.diff_model()` does.

Value

`plot.diff_model()` returns a bar chart of the number of new infected nodes at each time point, as well as an overlay line plot of the total in each value

`plot.diffs_model()` returns a line plot of the simulated diffusions, or, with `observed`, a band of the simulated diffusions with the observed diffusion over it.

`plot.diffusionSweep()` returns a plot of the share of the nodes infected for each setting of a sweep.

Examples

```
plot(res_manynet_diff)
plot(res_migraph_diff)
plot(play_learning(ison_networkers, beliefs = runif(net_nodes(ison_networkers))))
set.seed(1)
plot(manynet::as_diffusion(manynet::play_diffusion(manynet::ison_adolescents,
  seeds = 1, transmissibility = 0.5)))
```

---

plot.infernetConfusion

Plotting how well a model classifies ties

Description

This plotting method is for the result of `infernet::diagnose_confusion()`, which compares the ties that a model of a binary outcome predicts with the ties that were observed. The S3 class is "infernetConfusion".

- The ROC curve gives the true positive rate against the false positive rate at every threshold. The diagonal is a model that guesses, and the area under the curve (AUC) is in the title.
- The confusion matrix gives the mean count of each cell over the draws of the predicted probabilities, with its 95% interval.
- The metrics give the accuracy, sensitivity, specificity, precision, and F1 on one scale from 0 to 1.

Usage

```
## S3 method for class 'infernetConfusion'
plot(x, panel = c("all", "roc", "confusion", "metrics"), ...)
```

Arguments

`x`	An object of class "infernetConfusion". For example, `infernet::diagnose_confusion()`.
`panel`	By default "all", which shows the three panels together. "roc", "confusion", or "metrics" shows one panel.
`...`	Further arguments to be passed on to plot.

Value

A ggplot2 object, or a patchwork of three of them.

Examples

```
# Needs 'infernet' (https://github.com/stocnet/infernet)
## Not run:
  fit <- infernet::estimate_mrqap(. ~ ego(Age) + alter(Age),
                                  manynet::ison_adolescents |>
                                    manynet::mutate_nodes(Age = 12:19),
                                  times = 20, control = list(seed = 1))
  plot(infernet::diagnose_confusion(fit, times = 100))

## End(Not run)
```

`plot.mobility` *Plotting mobility models*

Description

Plotting mobility models

Usage

```
## S3 method for class 'mobilityModel'
plot(x, ...)
```

Arguments

`x`	A 'mobilityModel', as from `manynet::as_mobility()`.
`...`	Other arguments to be passed.

Details

`plot.mobilityModel()` draws the number of dissatisfied occupants after each move. A 'mobilityModel' alone does not hold the start, so its line begins at the first move.

`plot()` of a network of places with a record of moves, as from `manynet::play_mobility()`, draws more: a grid with a row for each step, from the start at step 0, and a column for each place. Each tile is filled by the attribute of its occupant, and an empty place is left blank. A cross marks a dissatisfied occupant, and an outline marks the place that an occupant moved to at that step. In the margin, a bar gives the number of dissatisfied occupants at each step.

To draw the places at the start, the middle, and the last move, use `graphs()` on the network of places.

Value

`plot.mobilityModel()` returns a line plot of the number of dissatisfied occupants at each step. `plot()` of a network of places returns a grid of the occupants of the places at each step, with a bar plot of the dissatisfied occupants.

Examples

```
places <- manynet::add_node_attribute(manynet::create_lattice(25), "group",
  rep(c(0, 1, NA, 1, 0), 5))
moved <- manynet::play_mobility(places, "group")
plot(moved)
plot(manynet::as_mobility(moved))
graphs(moved)
```

---

plot.network_test *Plotting methods for CUG and QAP tests*

Description

These plotting methods are for results obtained by testing some statistic against those produced in a reference distribution of conditional uniform graphs or as a quadratic assignment procedure. The S3 class is "network_test".

Usage

```
## S3 method for class 'network_test'
plot(x, ..., threshold = 0.95, tails = c("two", "one"))
```

Arguments

`x`	An object obtained from a conditional uniform graph or quadratic assignment procedure test. For example, `infernet::test_permutation()`.
`...`	Other arguments to be passed on.
`threshold`	The empirical threshold to shade in the plot.
`tails`	By default "two" indicating a two-tailed test, but "one" for a one-tailed test is also available.

Value

A distribution of the simulated or permuted statistics, with 2.5% shaded at each end, and a line highlighting where the observed statistic lies on this distribution.

Examples

```
# Here's something I cooked up with migraph earlier:
plot(res_migraph_test)
```

plot.opinion *Plotting opinion models*

Description

Plotting opinion models

Usage

```
## S3 method for class 'opinionModel'
plot(x, ..., stubbornness = NULL)

## S3 method for class 'opinionSweep'
plot(
  x,
  ...,
  outcome = c("clusters", "consensus", "spread", "mean", "steps"),
  level = 0.95
)
```

Arguments

x	An 'opinionModel', as from `manynet::as_opinion()`.
...	Other arguments to be passed.
stubbornness	Optionally, the stubbornness of each node, as one number or one number for each node, from 0 to 1. The nodes with a stubbornness above 0 are drawn with a thicker line, and a square at the last step. By default `NULL`, which marks no node.
outcome	For `plot.opinionSweep()`, the measure of the opinions at the end to draw: `"clusters"` (the default), `"consensus"` (the share of the runs that reach one), `"spread"`, `"mean"`, or `"steps"`.
level	For `plot.opinionSweep()`, the share of the runs that the band holds for each setting. By default 0.95.

Details

`plot.opinionModel()` draws one line for each node, from its first opinion at step 0 to its opinion at the last step. The colour of each line is the first opinion of its node, on the sequential palette of the theme, so that the colours meet where the opinions converge. Where all the nodes end at one opinion, a dashed line marks it.

`plot()` of a network with a record of opinions, as from `manynet::play_opinion()`, draws the same plot. It marks the stubborn nodes where the network has a node attribute named "stubbornness", as when `manynet::play_opinion(stubbornness = "stubbornness")` is used.

`plot.opinionSweep()` draws a measure of the opinions at the end for each setting of a sweep, as from `infernet::sweep_opinion()`: by default, the number of opinion clusters. It draws the settings as `plot.diffusionSweep()` does: a bar for each setting, and a panel for each value of any other argument in `vary`.

Value

`plot.opinionModel()` returns a line plot of the opinion of each node at each step.

`plot.opinionSweep()` returns a plot of a measure of the opinions at the end for each setting of a sweep.

Examples

```
set.seed(1)
net <- manynet::add_node_attribute(manynet::ison_networkers, "stubbornness",
  c(1, 0.5, rep(0, 30)))
opined <- manynet::play_opinion(net, opinions = stats::runif(32),
  stubbornness = "stubbornness")
plot(opined)
plot(manynet::as_opinion(opined))
```

---

plot.result.goldfish *Plotting a goldfish model fit at a glance*

Description

One call, four diagnostic panels: whether any interval is badly fitted, whether any coefficient drifts, whether each effect's contribution is spread over the sequence, and whether the waiting times are what the model says they are.

Usage

```
## S3 method for class 'result.goldfish'
plot(x, ..., effects = 4)

## S3 method for class 'goldfishFit'
plot(x, ..., effects = 4)
```

Arguments

`x`	A fitted model of class `goldfishFit`.
`...`	Additional plotting parameters, currently unused.
`effects`	The number of effects to draw in the Schoenfeld panel.

Details

`plot.result.goldfish()` is an alias for `plot.goldfishFit()`, kept so that a fit from a goldfish that still stamps the old class name plots as before. It will be removed.

Everything is drawn from what the **fit already stores** — no evaluation pass and no preprocessed statistics — so the figure costs a plot and not a re-fit. The consequence is that a panel needing a primitive the fit did not store is **left out** rather than erroring: which panels appear is itself a readout of what was requested at estimation.

deviance the per-interval log-likelihood with outlying intervals marked. Needs the `"loglik"` primitive.

scaled Schoenfeld a smooth per effect against the fitted estimate, flat under a constant coefficient. Needs `"scores"` on a multinomial sub-model, and `"conditional_scores"` on an exact-time one, where the score carries an exposure term the Schoenfeld residual does not.

cumulative score each effect's standardized process against its Brownian-bridge band. Needs `"scores"`.

waiting times the Cox-Snell residuals against the unit exponential they follow under the model. Exact-time sub-models only: an ordinal likelihood conditions the timing away, so there is no waiting time to check.

The Schoenfeld panel is capped at the `effects` most worth looking at, ranked by their cumulative-score statistic, since a model with a dozen terms makes a facet grid unreadable at overview size.

Value

A patchwork composition of the available panels.

Examples

```
if (requireNamespace("goldfish", quietly = TRUE)) {
  plot(goldfish_fit)
}
```

---

plot.transitions *Plotting the transitions of a diffusion*

Description

Plotting the transitions of a diffusion

Usage

```
## S3 method for class 'diffusionTransitions'
plot(x, ..., compartments = NULL)
```

Arguments

`x`	A 'diffusionTransitions' table, as `manynet::play_diffusion()` keeps in `info$transitions` of the network that it returns.
`...`	Other arguments to be passed.
`compartments`	Optionally, the roles of the values, as a named list such as `list(S = "S", I = "I")`, as `manynet::play_diffusion()` keeps in `info$compartments`. The roles give the colours of the values. By default `NULL`, which reads the values S, E, I, R, and D as these roles, and otherwise reads the values that a contagion moves nodes from as S, and the values that it comes from as I.

Details

`plot.diffusionTransitions()` draws the values of the attribute as circles in a row, and each transition as an arrow from one value to another, labelled with its rate. A transition to the next value in the row is straight, a transition back is curved under the row, and a transition that skips a value is curved over it. A transition by contagion gets a dashed arrow from each value that it comes from (its `source`), labelled with how a node is exposed: "threshold", "cascade", or "voter".

A rate that differs between nodes is labelled with its range, a rate from a node attribute with the name of the attribute, and a rate from a function with "f(x)".

Value

`plot.diffusionTransitions()` returns a diagram of the values and the transitions between them.

Examples

```
set.seed(1)
net <- manynet::play_diffusion(manynet::ison_adolescents, seeds = 1,
  transmissibility = 0.5, onset = 0.6, recovery = 0.2, waning = 0.05)
plot(net$info$transitions)
```

---

plot_convergence *Plotting convergence diagnostics*

Description

These plotting methods are for diagnosing the convergence of simulation-based estimation procedures, such as those used in MoNAn and ergm. These plots are useful for identifying whether the estimation procedure has adequately explored the state space and converged to a stable distribution.

Usage

```
## S3 method for class 'ag_conv'
plot(x, ...)

## S3 method for class 'traces.monan'
plot(x, ...)

## S3 method for class 'ergm'
plot(x, ...)

load_ergm_res()
```

Arguments

`x`	An object of class "traces.monan".
`...`	Additional plotting parameters, currently unused.

Value

The function shows a line plot tracing the statistics obtained at each simulation step, as well as a density plot showing the distribution of the statistics over the entire simulation.

See Also

Other MoNAn: plot_gof

Other ergm: plot_gof

Examples

```
plot(monan_conv)
ergm_res <- load_ergm_res()
plot(ergm_res)
```

---

plot_gof *Plotting goodness-of-fit results*

Description

These plot methods plot goodness of fit objects created using `RSiena::sienaGOF()`, `MoNAn::monanGOF()`, or the 'ergm' package's gof() function. Internally, the GOF object is translated into a common class (`ag_gof`), which has its own plot method to ensure a consistent look and feel. It is not expected that users will create `ag_gof` class objects themselves.

The plot shows a violin plot of the distribution of statistics from the simulations, with a boxplot inside the violin to show the interquartile range, and dashed lines connecting the 5th and 95th percentiles. The boxplot also shows outliers as crosses. The observed statistics are shown as points and connected by a line. The observed statistics are also labelled with their value. If a p-value is available (as in the case of `RSiena::sienaGOF()`), it is shown beneath the x-axis.

Usage

```
## S3 method for class 'ag_gof'
plot(x, ...)

## S3 method for class 'gof.stats.monan'
plot(x, cumulative = FALSE, ...)

## S3 method for class 'sienaGOF'
plot(x, cumulative = FALSE, ...)

## S3 method for class 'gof.ergm'
plot(
  x,
  cumulative = FALSE,
 statistic = c("degree", "odegree", "idegree", "b1degree", "b2degree", "espartners",
```

```
    "dspartners", "distance"),
  ...
)
```

Arguments

x	An object of class "sienaGOF", "gof.stats.monan", or "gof.ergm".
...	Other parameters to be passed to the plotting function, for example `main = "Title"` for a different title than the default.
cumulative	Logical, indicating whether the statistics should be plotted cumulatively (default FALSE). This is typically treated in `sienaGOF()` for {RSiena}, but treated within the plotting function for {MoNAn} and 'ergm'.
statistic	Character, indicating which statistic to plot. Since 'ergm' package GOFs include goodness of fit on multiple statistics, the user must specify which statistic to plot. Options are `"deg"` (degree distribution), `"espart"` (edgewise shared partners), and `"dist"` (geodesic distance). The default is `"deg"`.

Details

Since these plots methods are in {autograph}, the plots are automatically themed according to the current theme set using `stocnet_theme()`. The function uses the highlight colour defined in the current theme to highlight the observed statistics. The function also uses the base colour defined in the current theme to draw the violin and box plots.

It is however completely customisable. While a title is automatically generated so that the graph is informative, this can be customised by specifying the `main` argument in the plotting function, or added after the fact using {ggplot2} functions such as `ggtitle()` or `labs()`.

The user can choose whether to plot the statistics cumulatively or not. This is typically handled within `RSiena::sienaGOF()`, but for `MoNAn::monanGOF()` and the 'ergm' package's gof() function the cumulative option is handled here. The default is to plot the non-cumulative statistics. This is because the non-cumulative statistics are often more interpretable, and the cumulative statistics can be obtained by setting `cumulative = TRUE`.

The function also checks whether any of the statistics have zero variance across the simulations, and if so, these statistics are not plotted, with a message to the user indicating which statistics were omitted.

Note that these methods overwrite any plot methods for these classes that may be provided by the original packages. You may receive such a warning in the console when loading the package. Please load {autograph} after these other packages to ensure the plotting methods included in this package are used, or specify the package when calling the plotting method directly, e.g., `autograph::plot.sienaGOF(res_siena_gof)`.

Value

A violin plot showing the distribution of statistics from the simulations and a line joining points showing the observed statistics.

References

Hintze, J. L. and Nelson, R. D. 1998. "Violin plots: A box plot-density trace synergism". *The American Statistician*, 52:181–184. doi:10.1080/00031305.1998.10480559

See Also

Other MoNAn: `plot_convergence`

Other RSiena: `plot_interp`

Other ergm: `plot_convergence`

Examples

```
plot(monan_gof)
plot(siena_gof, cumulative = TRUE)
plot(ergm_gof, statistic = "espart")
```

---

plot_graphr *Easily graph networks with sensible defaults*

Description

This function provides users with an easy way to graph (m)any network data for exploration, investigation, inspiration, and communication.

`graphr()` builds upon {ggplot2} and {ggraph} to offer pretty, easy, and extensible graphing solutions. Just passing the function some network data will often be sufficient to return a reasonable-looking graph.

The function also makes it easy to modify many of the most commonly adapted aspects of a graph, including node and edge size, colour, and shape, as arguments rather than additional functions that you need to remember. These can be defined outright, e.g. `node_size = 8`, or in reference to an attribute of the network, e.g. `node_size = "wealth"`.

Lastly, `graphr()` uses {ggplot2}-related theme information, so it is easy to make colour palette and fonts institution-specific and consistent. See e.g. `theme_iheid()` for more.

To learn more about what can be done visually, try `run_tute("Visualisation")`.

Usage

```
graphr(
  .data,
  layout = NULL,
  labels = TRUE,
  node_color,
  node_shape,
  node_size,
  node_group,
  edge_color,
```

```
  edge_size,
  isolates = c("legend", "caption", "keep"),
  layout_snap = FALSE,
  label_repel = TRUE,
  edge_bundle = FALSE,
  edge_arrows = NULL,
  backbone = NULL,
  ...,
  node_colour,
  edge_colour,
  snap,
  label_dist
)
```

Arguments

.data	A manynet-consistent object.
layout	An igraph, ggraph, or manynet layout algorithm. If not declared, defaults to "configuration" for networks of up to six nodes, "levels" for connected multilevel networks, "layered" for other two mode networks, and "stress" for all other networks. The "stress" layout of a network of more than 1000 nodes is run at a looser tolerance (`tolerance = 1e-3`) to save time; give `tolerance = 1e-4` for the finer layout. For "layered" layout, one can further split graph by declaring the "center" argument as the "events", "actors", or by declaring a node name. For "concentric" layout algorithm please declare the "membership" as an extra argument. The "membership" argument expects either a quoted node attribute present in data or vector with the same length as nodes to draw concentric circles. For "levels" layout algorithm one may declare the "level" as extra argument. The "level" argument expects either a quoted node attribute present in data or vector with the same length as nodes to hierarchically order categories. If "level" is missing, the levels are taken from a 'lvl' node attribute where there is one, or else from the two modes of a two mode network. The layered layouts ("layered", "lineage", "railway" and "ladder") accept a "ranks" argument, which takes either one of the methods named at `?layout_layered` or a numeric node attribute to lay the layers out by, as a quoted attribute name or a vector with one value for each node. A "lineage" ranked by such values, such as `ranks = "year"`, is drawn as a timeline, as a historiograph of citations is: the x axis is broken at the values, with light vertical gridlines, the ties are curved, and the labels are repelled. Every "lineage" draws its ties curved and its mapped node sizes by area over a wider range. The "scaling" layout places the nodes by multidimensional scaling, so that the distance between two nodes approximates the number of steps between them. Since those coordinates can be read, this layout is drawn with labelled axes on one scale, and captioned with how well two dimensions hold the distances; see `?layout_scaling` and `check_stress()`. Note that those axes carry distances rather than named dimensions: the drawing can be turned or mirrored without fitting the network any better or any worse. The "correspondence" layout places the nodes by correspondence analysis, so that two nodes with similar ties are drawn together, whether or not they are

tied to each other. It is the usual way to draw a two mode network, since it places both modes against the same pair of axes, and it accepts a "direction" argument for a directed network and a "double" argument for a signed one; see `?layout_correspondence`. Each axis names the share of the network's inertia that it holds.

`labels` Which nodes to label, if the network is labelled. `TRUE` (the default) labels every node and `FALSE` none of them, but a label for every node of a large network hides the network behind them, so a *selection* of the nodes can be given instead:

- a number, e.g. `labels = 5`, labels the nodes within the top five ranks by degree. Note that this is a depth of ranks rather than a count of nodes: nodes tied at the cut are labelled together, so more than five labels may appear.
- a measure to rank by, e.g. `labels = "betweenness"`, labels just the node or nodes that measure singles out. `"degree"`, `"betweenness"`, `"cutpoints"` (every node the mark flags) and `"random"` (a small random sample) are available. The two can be combined by naming the number, as in `labels = c(betweenness = 5)`.
- the name of a logical node attribute, e.g. `labels = "is_broker"`, labels the nodes it marks.
- a logical vector, one value per node, e.g. `labels = netrics::node_is_cutpoint(net)`; or the names or positions of the nodes to label, e.g. `labels = c("Alice", "Betty")`.

Where a length-one string could mean more than one of these, a node attribute is preferred to a measure, and a measure to a node name. A single number is always read as a depth of ranks rather than as one node's position, so a lone node is best named, as in `labels = "Alice"`. For networks of more than 30 nodes, `labels` defaults to a selection rather than to every node; pass `labels = TRUE` for all of them. For a concept lattice from `manynet::to_concepts()`, `labels` defaults to the concepts at which a node of the original network first appears, so that each of those nodes is named once, as in a Hasse diagram, and the concepts named only by their position are left unlabelled. Ranking nodes uses the {netrics} package, which is suggested rather than required: without it installed, an automatic selection falls back to a random sample. Two-mode and multilevel networks are ranked within each mode or level, so that every level is labelled and not just the densest.

`node_color`, `node_colour`

Node variable to be used for coloring the nodes. It is easiest if this is added as a node attribute to the graph before plotting. A categorical variable gives one colour to each category. A measure, such as a centrality or coreness score, is drawn instead as a gradient from the theme's base colour to its highlight colour, with a colourbar in place of the legend. Nodes can also be colored by declaring a color instead.

`node_shape` Node variable to be used for shaping the nodes. It is easiest if this is added as a node attribute to the graph before plotting. Nodes can also be shaped by declaring a shape instead.

`node_size` Node variable to be used for sizing the nodes. This can be any continuous variable on the nodes of the network. Since this function expects this to be an

existing variable, it is recommended to calculate all node-related statistics prior to using this function. Nodes can also be sized by declaring a numeric size or vector instead.

`node_group` Node variable to be used for grouping the nodes. It is easiest if this is added as a hull over groups before plotting. Group variables should have a minimum of 3 nodes, if less, number groups will be reduced by merging categories with lower counts into one called "other". A membership vector can also be given here. Where nodes belong to several groups at once, as they can to several cliques, give a membership matrix instead: one row for each node, one column for each group, and a one wherever the node belongs to the group. One hull is then drawn for each column, and the hulls overlap where the groups do. A measure that returns such a matrix, such as `netrics::node_x_clique()`, can be named without its network, which is taken to be the network being drawn.

`edge_color`, `edge_colour` Tie variable to be used for coloring the nodes. It is easiest if this is added as an edge or tie attribute to the graph before plotting. Edges can also be colored by declaring a color instead.

`edge_size` Tie variable to be used for sizing the edges. This can be any continuous variable on the nodes of the network. Since this function expects this to be an existing variable, it is recommended to calculate all edge-related statistics prior to using this function. Edges can also be sized by declaring a numeric size or vector instead.

`isolates` Character scalar, how to treat isolates. "keep" will keep isolates in the graph as they are. "legend" (default) will remove isolates from the graph but note them in the legend. "caption" will remove isolates from the graph but note them in the caption. If there are no isolates, this argument will be ignored. The "lineage" layout defaults to "keep" instead, since it gives every node a place in its layers, such as a year; an `isolates` value that is given is used for it as for any other layout. If the default layout ("stress") is used, we recommend that the "legend" option is used to avoid isolates crowding out the giant component.

`layout_snap` Logical scalar, whether the layout should be snapped to a grid. Where the network repeats a structure, as a lattice does, the two steps it repeats are mapped onto the axes, which draws it as a rectangle of rows and columns. Where it does not, each node moves to the nearest vacant grid point. Layouts that already carry meaning in their coordinates, such as "layered" or "scaling", are left as they are.

`label_repel` Where to put the labels.

- `TRUE` (the default) repels the labels away from each other and from the nodes using `ggrepel` (via `ggraph`'s `repel` argument), leaving a small gap from each node's border: 5pt where every node is labelled, and 2pt where only a selection is, so that each label stays against the node it names. Node size is always accounted for.
- A number, e.g. `label_repel = 10`, repels the labels with a gap of that many points instead, similar to `igraph`'s `vertex.label.dist`. Set to `0` for labels right at the node border.
- `FALSE` places each label at a fixed offset from its node, in the direction the layout suits (outward in "circle" and "concentric", to the right in "lineage"),

without the repelling algorithm, which can be slow and, for some layouts, differs between runs.

- A direction, `"outside"`, `"above"`, `"below"`, `"left"`, or `"right"`, places every label at a fixed offset in that direction, whatever the layout. `"outside"` points away from the middle of the layout. A gap can be given with it, as in `label_repel = c(above = 3)`.

The layered layouts ("layered", "lineage", "railway" and "ladder") place each node in a layer, which is where the reader looks for it, so a repelled label there would say less about which node it labels than a fixed offset does. They offset the labels instead, unless a direction is given.

`edge_bundle` Edge bundling, off by default (`FALSE`). When `TRUE` (or equivalently `"force"`), edges are bundled together using ggraph's force-directed edge bundling (`geom_edge_bundle_force()`), which pulls nearby edges into shared paths to reduce visual clutter in dense networks. Alternative non-hierarchical algorithms can be selected by name: `"path"` (`geom_edge_bundle_path()`) or `"minimal"` (`geom_edge_bundle_minimal()`). Bundling only makes a visible difference when a network has enough edges; for directed networks arrowheads are retained, but the slight reciprocal-tie curvature used for unbundled edges does not apply.

`edge_arrows` Whether, and at what size, to draw arrowheads on the ties of a directed network. By default (`NULL`) this is decided by the network: a directed network is drawn with arrowheads, except for a concept lattice from `manynet::to_concepts()` in the "layered" (its default) or "railway" layout. That is read as a Hasse diagram, where the direction of each tie is already given by which of its ends is drawn higher. In any other layout a concept lattice keeps its arrowheads. `TRUE` draws arrowheads whatever the network, and `FALSE` none. Arrowheads follow the width the ties are drawn at: 2mm long at the default `edge_size` of 0.5, and longer on wider ties, up to 4mm. Where the widths vary, as when they are mapped from a tie attribute, one size is chosen for all of the arrowheads from the mean width drawn. A number, e.g. `edge_arrows = 3`, draws arrowheads of that length in millimetres instead, whatever the width of the ties. This is the unit that `{ggplot2}` sizes are given in, as `node_size` is, so an arrowhead of 3 is a little longer than a node of 3 is wide. Ties that are not drawn, as with `edge_size = 0`, have no arrowheads. An undirected network has no direction to show, so the argument is ignored there.

`backbone` How to treat the network's backbone: the ties that a local null model keeps, because they carry more weight, or sit in more triangles, than chance alone would put there. Where a backbone is used, those ties are drawn as the shortest, so that the layout pulls apart the groups they hold together, and every tie is still drawn, with the ties the filter does not keep faded well back. This is what to reach for when a network is dense enough to draw as a hairball. By default (`NULL`) this is decided by the network: a network of at least 50 nodes and a mean degree of at least 8 is drawn this way, and reported. `FALSE` draws every tie alike, and `TRUE` asks for a backbone whatever the network's size. One of `manynet`'s filters can be named instead: "disparity", "lans", "noise", "mlf", or "simmelian". Where none is named, `manynet` uses "lans" for a weighted network and "simmelian" for an unweighted one. A number between 0 and 1 sets the threshold instead of the filter: a smaller number keeps fewer ties. Only the layouts that read tie lengths

– "stress" (the default), "fr", "drl" and "kk" – are laid out this way. Every other layout, including those that already carry meaning in their coordinates such as "layered" or "scaling", keeps its coordinates and only fades its ties. Does not apply to signed networks.

`...` Extra arguments to pass on to the layout algorithm, if necessary.

`snap`, `label_dist`
Deprecated. Use `layout_snap` instead of `snap`, and give the gap to `label_repel` instead of `label_dist`.

Value

A `ggplot2::ggplot()` object. The last plot can be saved to the file system using `ggplot2::ggsave()`.

See Also

Other mapping: `check_layout`, `completion`, `layout_concentric()`, `layout_configuration()`, `layout_correspondence()`, `layout_layered()`, `layout_levels()`, `layout_matching()`, `layout_scaling()`, `layout_spatial()`, `layout_valence()`, `plot_graphs`, `plot_grapht`

Examples

```
graphr(ison_adolescents)
ison_adolescents |>
  mutate_nodes(color = rep(c("introvert","extrovert"), times = 4),
         size = c(3, 6, 3, 3, 3, 6, 6, 3)) |>
  mutate_ties(ecolor = rep(c("friends", "acquaintances"), times = 5)) |>
  graphr(node_color = "color", node_size = "size",
         edge_size = 1.5, edge_color = "ecolor")
graphr(ison_southern_women, labels = TRUE, label_repel = 10)
# Label a selection of the nodes rather than all of them
graphr(ison_southern_women, labels = 2)
graphr(ison_adolescents, labels = c("Alice", "Betty"))

graphr(ison_southern_women, labels = TRUE, label_repel = FALSE)
graphr(ison_karateka, label_repel = "outside")
graphr(ison_southern_women, labels = "betweenness")
graphr(manynet::generate_random(40, 0.1), edge_bundle = TRUE)
# Larger arrowheads than the width of the ties would give, or none at all
graphr(ison_networkers, edge_arrows = 4)
graphr(ison_networkers, edge_arrows = FALSE)
graphr(manynet::generate_random(80, 0.2), backbone = TRUE)
```

`plot_graphs` *Easily graph a set of networks with sensible defaults*

Description

This function provides users with an easy way to graph lists of network data for comparison.

It builds upon this package's `graphr()` function, and inherits all the same features and arguments. See `graphr()` for more. However, it uses the `{patchwork}` package to plot the graphs side by side and, if necessary, in successive rows. This is useful for lists of networks that represent, for example, ego or component subgraphs of a network, or a list of a network's different types of tie or across time. By default just the first and last network will be plotted, but this can be overridden by the "waves" parameter.

Where the graphs are of the same network (the same nodes in the same order), the graphs may share a layout to facilitate comparison. By default, successive graphs will use the layout calculated for the "first" network, but other options include the "last" layout, or a mix, "both", of them.

Usage

```
graphs(netlist, waves, based_on = c("first", "last", "both"), ...)
```

Arguments

`netlist`	A list of manynet-compatible networks. This can also be a single manynet network object that encodes time, which is split automatically (as in `grapht()`) by `manynet::to_times()` into the network as it stood at each moment it records: longitudinal or changing networks into waves, dynamic (time-stamped, event-based) networks such as `manynet::irps_nuclear` into the state accumulated by each moment, and interval (spell) networks that record tie `begin`/`end` lifespans, such as `manynet::irps_wwi`, into one snapshot per change point. It can also be a diffusion model result from e.g. `manynet::play_diffusion()`. A network of places with a record of moves, as from `manynet::play_mobility()`, is drawn at the start, the middle, and the last move, coloured by the attribute of the occupants, with the empty places blank. For such a network, `waves` select the steps, where 1 is the start. A single network that encodes no time is drawn with `graphr()`.
`waves`	Numeric, the number of plots to be displayed side-by-side. If missing, the number of plots will be reduced to the first and last when there are more than four plots. This argument can also be passed a vector selecting the waves to plot.
`based_on`	Whether the layout of the joint plots should be based on the "first" or the "last" network, or "both".
`...`	Additional arguments passed to `graphr()`.

Value

Multiple `ggplot2::ggplot()` objects displayed side-by-side.

See Also

Other mapping: `check_layout`, `completion`, `layout_concentric()`, `layout_configuration()`, `layout_correspondence()`, `layout_layered()`, `layout_levels()`, `layout_matching()`, `layout_scaling()`, `layout_spatial()`, `layout_valence()`, `plot_graphr`, `plot_grapht`

Examples

```
#graphs(to_egos(ison_adolescents))
#graphs(to_egos(ison_adolescents), waves = 8)
#graphs(to_egos(ison_adolescents), waves = c(2, 4, 6))
#graphs(play_diffusion(ison_adolescents))
```

`plot_grapht` *Easily animate dynamic networks with sensible defaults*

Description

This function provides users with an easy way to graph dynamic network data for exploration and presentation.

It builds upon this package's `graphr()` function, and inherits all the same features and arguments. See `graphr()` for more. However, it uses the {gganimate} package to animate the changes between successive iterations of a network. This is useful for networks in which the ties and/or the node or tie attributes are changing, including networks whose node composition changes over time: every node that ever appears is assigned a stable position, and nodes fade in and out in place as they enter and exit the network.

By default node positions transition smoothly between waves using the dynamic stress layout from {graphlayouts} (`graphlayouts::layout_as_dynamic()`), which anchors each wave's layout to a reference layout of the aggregate network. The `alpha` argument controls this trade-off: 0 lets each wave's layout follow that wave's structure freely, while 1 freezes every node at its aggregate position. When another `layout` is requested, a single static layout is computed on the aggregate (union of waves) network instead, so that positions remain constant. Unlike `graphr()`, `grapht()` uses this dynamic stress layout by default even for two-mode networks (rather than a layered layout, which would collapse many nodes onto a line); the two modes remain distinguishable by node shape. For networks with more than 30 nodes, node labels are suppressed by default to keep frames legible; pass `labels = TRUE` to force them.

`grapht()` returns a {ggplot2}-compatible object that can be extended with additional layers such as `ggplot2::labs()`, `ggplot2::theme()`, scale functions, and others, just like plots produced by `graphr()` and `graphs()`. The animation is rendered when the object is printed or displayed. Users who want more control over animation parameters can call `gganimate::animate()` directly on the returned object.

The visual appearance is consistent with `graphr()`: nodes use fillable shapes with the fill aesthetic, the same colour palettes are applied, directed networks receive arrowheads (see `edge_arrows`), signed networks distinguish positive from negative ties by linetype, and labels use the current theme font. Legends transition along with the mapped aesthetics.

A progress bar is shown if it takes some time to encode all the .png files into a .gif.

Usage

```
grapht(
  tlist,
  layout = NULL,
```

```
  labels = TRUE,
  node_color,
  node_shape,
  node_size,
  edge_color,
  edge_size,
  isolates = c("keep", "fade"),
  alpha = 0.5,
  label_repel = TRUE,
  edge_arrows = NULL,
  keep_isolates = NULL,
  ...,
  node_colour,
  edge_colour,
  label_dist
)

## S3 method for class 'grapht'
print(x, ...)
```

Arguments

tlist	A manynet-compatible network listed according to a time attribute, waves, or slices. This can also be a single manynet network object that encodes time, which is split automatically by `manynet::to_times()` into the network as it stood at each moment it records: longitudinal or changing networks into waves, dynamic (time-stamped, event-based) networks such as `manynet::irps_nuclear` into the state accumulated by each moment, and interval (spell) networks that record tie `begin`/`end` lifespans, such as `manynet::irps_wwi`, into one snapshot per change point showing the ties active in that spell. It can also be a diffusion model result from e.g. `manynet::play_diffusion()`.
layout	An igraph, ggraph, or manynet layout algorithm. If not declared, defaults to "configuration" for networks of up to six nodes, "levels" for connected multilevel networks, "layered" for other two mode networks, and "stress" for all other networks. The "stress" layout of a network of more than 1000 nodes is run at a looser tolerance (`tolerance = 1e-3`) to save time; give `tolerance = 1e-4` for the finer layout. For "layered" layout, one can further split graph by declaring the "center" argument as the "events", "actors", or by declaring a node name. For "concentric" layout algorithm please declare the "membership" as an extra argument. The "membership" argument expects either a quoted node attribute present in data or vector with the same length as nodes to draw concentric circles. For "levels" layout algorithm one may declare the "level" as extra argument. The "level" argument expects either a quoted node attribute present in data or vector with the same length as nodes to hierarchically order categories. If "level" is missing, the levels are taken from a 'lvl' node attribute where there is one, or else from the two modes of a two mode network. The layered layouts ("layered", "lineage", "railway" and "ladder") accept a "ranks" argument, which takes either one of the methods named at `?layout_layered` or a numeric node

attribute to lay the layers out by, as a quoted attribute name or a vector with one value for each node. A "lineage" ranked by such values, such as `ranks = "year"`, is drawn as a timeline, as a historiograph of citations is: the x axis is broken at the values, with light vertical gridlines, the ties are curved, and the labels are repelled. Every "lineage" draws its ties curved and its mapped node sizes by area over a wider range. The "scaling" layout places the nodes by multidimensional scaling, so that the distance between two nodes approximates the number of steps between them. Since those coordinates can be read, this layout is drawn with labelled axes on one scale, and captioned with how well two dimensions hold the distances; see `?layout_scaling` and `check_stress()`. Note that those axes carry distances rather than named dimensions: the drawing can be turned or mirrored without fitting the network any better or any worse. The "correspondence" layout places the nodes by correspondence analysis, so that two nodes with similar ties are drawn together, whether or not they are tied to each other. It is the usual way to draw a two mode network, since it places both modes against the same pair of axes, and it accepts a "direction" argument for a directed network and a "double" argument for a signed one; see `?layout_correspondence`. Each axis names the share of the network's inertia that it holds.

`labels` Which nodes to label, if the network is labelled. `TRUE` (the default) labels every node and `FALSE` none. A selection of the nodes can be given as in `graphr()`: a number of ranks, a measure to rank by, the name of a logical node attribute, a logical vector, or the names or positions of the nodes. The selection is made once over all the waves, so that the same nodes stay labelled from frame to frame. For networks of more than 30 nodes, `labels` defaults to no labels at all; pass `labels = TRUE` for all of them. Unlike `graphr()`, no selection is made for a concept lattice.

`node_color`, `node_colour`
Node variable to be used for coloring the nodes. It is easiest if this is added as a node attribute to the graph before plotting. A categorical variable gives one colour to each category. A measure, such as a centrality or coreness score, is drawn instead as a gradient from the theme's base colour to its highlight colour, with a colourbar in place of the legend. Nodes can also be colored by declaring a color instead.

`node_shape` Node variable to be used for shaping the nodes. It is easiest if this is added as a node attribute to the graph before plotting. Nodes can also be shaped by declaring a shape instead.

`node_size` Node variable to be used for sizing the nodes. This can be any continuous variable on the nodes of the network. Since this function expects this to be an existing variable, it is recommended to calculate all node-related statistics prior to using this function. Nodes can also be sized by declaring a numeric size or vector instead.

`edge_color`, `edge_colour`
Tie variable to be used for coloring the nodes. It is easiest if this is added as an edge or tie attribute to the graph before plotting. Edges can also be colored by declaring a color instead.

`edge_size` Tie variable to be used for sizing the edges. This can be any continuous variable on the nodes of the network. Since this function expects this to be an existing

variable, it is recommended to calculate all edge-related statistics prior to using this function. Edges can also be sized by declaring a numeric size or vector instead.

`isolates` One of `"keep"` (the default) or `"fade"`. `"keep"` retains isolated nodes at their layout positions in every wave in which they are present. `"fade"` fades nodes out during waves in which they are isolates, and fades them back in when they regain ties. Nodes that are absent from a wave altogether (composition change) always fade out.

`alpha` A number between 0 and 1 controlling the stability of node positions across waves when the default dynamic (stress) layout is used. 0 computes each wave's layout freely, 1 fixes all nodes at their aggregate-network positions. By default 0.5. Passed to `graphlayouts::layout_as_dynamic()`.

`label_repel` Where to put the labels.

- `TRUE` (the default) repels the labels away from each other and from the nodes using `ggrepel` (via `ggraph`'s `repel` argument), leaving a small gap from each node's border: 5pt where every node is labelled, and 2pt where only a selection is, so that each label stays against the node it names. Node size is always accounted for.
- A number, e.g. `label_repel = 10`, repels the labels with a gap of that many points instead, similar to `igraph`'s `vertex.label.dist`. Set to `0` for labels right at the node border.
- `FALSE` places each label at a fixed offset from its node, in the direction the layout suits (outward in "circle" and "concentric", to the right in "lineage"), without the repelling algorithm, which can be slow and, for some layouts, differs between runs.
- A direction, `"outside"`, `"above"`, `"below"`, `"left"`, or `"right"`, places every label at a fixed offset in that direction, whatever the layout. `"outside"` points away from the middle of the layout. A gap can be given with it, as in `label_repel = c(above = 3)`.

The layered layouts ("layered", "lineage", "railway" and "ladder") place each node in a layer, which is where the reader looks for it, so a repelled label there would say less about which node it labels than a fixed offset does. They offset the labels instead, unless a direction is given.

`edge_arrows` Whether, and at what size, to draw arrowheads on the ties of a directed network. By default (`NULL`) this is decided by the network: a directed network is drawn with arrowheads, except for a concept lattice from `manynet::to_concepts()` in the "layered" (its default) or "railway" layout. That is read as a Hasse diagram, where the direction of each tie is already given by which of its ends is drawn higher. In any other layout a concept lattice keeps its arrowheads. `TRUE` draws arrowheads whatever the network, and `FALSE` none. Arrowheads follow the width the ties are drawn at: 2mm long at the default `edge_size` of 0.5, and longer on wider ties, up to 4mm. Where the widths vary, as when they are mapped from a tie attribute, one size is chosen for all of the arrowheads from the mean width drawn. A number, e.g. `edge_arrows = 3`, draws arrowheads of that length in millimetres instead, whatever the width of the ties. This is the unit that `{ggplot2}` sizes are given in, as `node_size` is, so an arrowhead of 3 is a little

	longer than a node of 3 is wide. Ties that are not drawn, as with `edge_size = 0`, have no arrowheads. An undirected network has no direction to show, so the argument is ignored there.
`keep_isolates`	Deprecated. Use `isolates = "keep"` or `isolates = "fade"` instead.
`...`	Extra arguments to pass on to the layout algorithm, if necessary.
`label_dist`	Deprecated. Give the gap to `label_repel` instead.
`x`	A grapht object to print.

Details

Unlike `graphr()`, `grapht()` does not use ggrepel-based label repelling (there is no straightforward way to repel labels consistently across animation frames), so `label_repel` here instead gives a fixed offset nudging labels away from their nodes: above them for `TRUE`, none for `FALSE`, and in the direction named for a direction. A gap in points scales the size of that nudge rather than being used as repel padding.

`labels` can select which nodes to label here too, and the selection is resolved once over all the waves so that the same nodes stay labelled from frame to frame. Unlike `graphr()`, though, animations of more than 30 nodes default to no labels at all rather than to a selection of them.

Some further `graphr()` features are not available in animations: `node_group` hulls, edge bundling, curved arcs for reciprocated ties, and self-loops (loops are not drawn; a note is printed if present). Note too that, where no `layout` is named, `grapht()` defaults to the "stress" layout for every network rather than choosing one by the network's shape as `graphr()` does, so that nodes move smoothly from one wave to the next. A layout named explicitly is still used, computed on the aggregate network.

Value

A {ggplot2}-compatible object with {gganimate} animation layers. This object can be extended with additional {ggplot2} layers (e.g. `+ labs(subtitle = "My subtitle")`). When printed or displayed, the animation is rendered as a .gif. For more control over animation parameters, pass the result to `gganimate::animate()` directly.

Source

https://blog.schochastics.net/posts/2021-09-15_animating-network-evolutions-with-gganimate/

See Also

Other mapping: `check_layout`, `completion`, `layout_concentric()`, `layout_configuration()`, `layout_correspondence()`, `layout_layered()`, `layout_levels()`, `layout_matching()`, `layout_scaling()`, `layout_spatial()`, `layout_valence()`, `plot_graphr`, `plot_graphs`

Examples

```
# A dynamic signed network of shifting European alliances 1872-1918,
# split automatically into snapshots of the ties active in each spell.
# Wrapped in \donttest{} because rendering the animation to a .gif is
# slow, not because the code is unsafe to run.
```

```
grapht(irps_wwi)
```

`plot_interp` *Plotting effects interpretation*

Description

These functions support the interpretation of network and behavior effects found in stochastic actor-oriented models. They are S3 plotting methods for objects of class "selectionTable" or "influenceTable", created using `RSiena::selectionTable()` or `RSiena::influenceTable()`, respectively. They plot how the evaluation function for selection or influence changes based on ego's value and alter's value of some covariate. This helps to interpret the effect of that covariate on the network dynamics or behavior dynamics, respectively.

Usage

```
## S3 method for class 'selectionTable'
plot(x, quad = TRUE, separation = 0, ...)

## S3 method for class 'influenceTable'
plot(x, separation = 0, ...)
```

Arguments

`x`	An object of class "selectionTable" or "influenceTable", created using `RSiena::selectionTable()` or `RSiena::influenceTable()`, respectively.
`quad`	When TRUE (the default), a quadratic function (average and total alter) is plotted. Use `quad = FALSE` for similarity effects.
`separation`	This can be used to make the curves visually distinguishable if they overlap too much without it. An advisable value then is, e.g., 0.01.
`...`	Other arguments to be passed.

Details

These functions were originally written by Tom Snijders, and adapted for use in the `{autograph}` package.

Value

A plot showing how the selection/influence evaluation function changes based on ego's value and alter's value of some covariate.

Author(s)

Tom Snijders

Thanks to Steffen Triebel and Rene Veenstra for corrections.

References

For plotting selection tables, please consult the RSiena manual, Sections 13.1 and 13.3.

For plotting selection tables, please consult the RSiena manual, Sections 13.2 and 13.4.

See Also

Other RSiena: plot_gof

Examples

```
plot(siena_selection)
plot(siena_selection) + ggplot2::scale_colour_discrete()
plot(siena_influence)
```

---

theme_colorblind *Simulating how colours look to a colour-blind viewer*

Description

`simulate_colorblind()` returns what a set of colours looks like to a viewer with a given type of colour vision deficiency (CVD), which affects about 8% of men and 0.5% of women, or in greyscale, as a reader sees the plot in print.

Usage

```
simulate_colorblind(
  colors,
  type = c("deutan", "protan", "tritan", "grey", "normal"),
  severity = 1
)
```

Arguments

`colors`	One or more colours, given as hexcodes or as names R knows.
`type`	The type of colour blindness to simulate: "deutan" (green-blind, the most common), "protan" (red-blind), "tritan" (blue-blind), "grey" for greyscale, as a photocopier renders it, or "normal" for unaffected vision.
`severity`	How severe the colour blindness is, between 0 and 1. By default 1, which is dichromacy. A value between 0 and 1 is anomalous trichromacy. Ignored for the "grey" and "normal" types.

Details

Simulation uses the matrices of Machado, Oliveira and Fernandes (2009), applied in linear RGB. Those matrices are published for each severity of colour blindness; `severity` interpolates between the identity and the full-severity matrix, which approximates the published steps closely enough for a check. Full severity is dichromacy (deuteranopia, protanopia, tritanopia); a lower severity is anomalous trichromacy (deuteranomaly, protanomaly), which is the more common condition. Greyscale conversion takes the relative luminance of the colour, the same quantity `check_contrast()` scores with.

Value

`simulate_colorblind()` returns a vector of hexcodes as long as `colors`.

References

Machado, Gustavo M., Manuel M. Oliveira, and Leandro A. F. Fernandes. 2009. "A Physiologically-Based Model for Simulation of Color Vision Deficiency". *IEEE Transactions on Visualization and Computer Graphics* 15(6): 1291-98. doi:10.1109/TVCG.2009.113

See Also

`check_colors()`, which scores a palette rather than simulating it.

Other themes: `check_colors`, `list_fonts()`, `theme_medium`, `theme_set`

Examples

```
simulate_colorblind(c("#d73027", "#4575b4"), "deutan")
# A milder deuteranomaly, and the same colours in greyscale
simulate_colorblind(c("#d73027", "#4575b4"), "deutan", severity = 0.5)
simulate_colorblind(c("#d73027", "#4575b4"), "grey")
```

---

theme_match *Matching colors across palettes*

Description

Sometimes a palette or particular colours are chosen to symbolise or represent a particular idea, such as red for "stop" or green for "go", or to convey some other interpretation. Yet institutional palettes do not necessarily include all colours, which can constrain how interpretable visualisations are under institutional branding requirements. `match_color()` helps to find the closest matching colours in a given palette to one or more input colours.

There is also a helper function, `is_dark()`, to determine whether a color is dark or light, which can be useful when deciding whether to use white or black text on top of a colored background.

Usage

```
match_color(colors, pal)

is_dark(colors)
```

Arguments

colors	One or more hexcodes to match with colors from the palette.
pal	Optionally, a vector of hexcodes representing a palette in which to find matches. By default, the current theme's qualitative palette is used.

Details

This function uses the Euclidean distance of colours in CIELAB space to those of a target palette to find the closes corresponding colours. It also ensures that each input color is matched to a unique color in the palette. If there are more input colors than unique colors in the palette, an error is returned.

By default, the current theme's qualitative palette is used, but any vector of hexcodes can be passed to the pal argument.

Value

A vector of hexcodes the length of the first argument.

Examples

```
match_color("#4575b4")
is_dark(c("#000","#FFF"))
```

---

theme_medium *Setting the medium a plot is made for*

Description

A theme says how a plot should look. A medium says where it will be seen, which is a separate question: the same institutional theme serves a figure worked on at a desk, projected in a lecture theatre, printed in an article, and read on a phone, but each of those wants a different size of text and, in one case, a different ground. stocnet_medium() sets the medium for all subsequent plots, as stocnet_theme() sets the theme, and leaves the theme alone.

If no medium is specified (i.e. the function is called without argument), the current medium is reported. The default medium is "screen".

Usage

```
stocnet_medium(medium = NULL, persist = FALSE)

set_stocnet_medium(medium = NULL, persist = FALSE)

ag_size()
```

Arguments

medium	String naming a medium. By default "screen". The following media are currently available: screen, presentation, mobile, print. This string can be capitalised or not.
persist	Logical, by default FALSE. If TRUE, the medium is remembered across sessions, by writing it to the user's configuration directory (see `tools::R_user_dir()`). Nothing is written to disk unless this is set explicitly. Use `stocnet_medium(persist = FALSE)` when setting a medium to forget a previously persisted choice.

Details

The media available are:

- "screen", the default, which draws as {autograph} always has.
- "presentation", which enlarges text by half, for a figure read from the back of a room.
- "mobile", which enlarges text further, for a figure read in a narrow column on a handheld screen. Keep such a figure to one point, with few categories: a legend of more than about seven keys, or more than about three panels from `graphs()`, will not survive the width.
- "print", which leaves text at its usual size but draws on white, whatever ground the theme prefers. A dark or tinted ground costs ink and is often not reproduced.

The medium scales text, not marks. Node sizes are relative to the layout they sit in, so enlarging them without enlarging the layout would crowd it. Where a figure needs larger nodes as well, set `node_size` in `graphr()`.

The medium does not set the size of the file written. Give `ggplot2::ggsave()` the width, height and resolution the medium calls for as well.

Value

`stocnet_medium()` sets the medium to be used across all stocnet packages. The medium is written to an option and held there. `ag_size()` returns the multiplier the current medium applies to text sizes, which is 1 unless the medium says otherwise.

See Also

Other themes: `check_colors`, `list_fonts()`, `theme_colorblind`, `theme_set`

Examples

```
stocnet_medium("presentation")
ag_size()
stocnet_medium("screen")
```

`theme_set` *Setting a consistent theme for all plots*

Description

This function enables plots to be quickly, easily and consistently themed. This is achieved by setting a theme option, usually at the start of an R session, that enables the palette to be used for all autograph-consistent plotting methods. This includes thematic colours for backgrounds, highlights, sequential, divergent and categorical colour schemes. The function sets these palettes to options that are then used by the various plotting functions.

If no theme is specified (i.e. the function is called without argument), the current theme is reported. The default theme is "default". This theme uses a white background, blue and red for highlighting, and a blue-white-red divergent palette. The themes can be changed at any time by calling `stocnet_theme()` or its alias `set_stocnet_theme()` with a different theme name.

Other themes include those based on the colour schemes of various universities, including ETH Zurich, UZH, UNIBE, RUG, and Oxford. Other themes include "bw" for black and white, "crisp" for a high-contrast black and white theme, "neon" for a dark theme with neon highlights, and "rainbow" for a colourful theme. The "clay" theme follows the palette and fonts used in the slides and documents that Anthropic's Claude produces: an ivory background, a slate ink base, and a clay orange highlight. Most themes are designed to be colour-blind safe.

Usage

```
stocnet_theme(theme = NULL, persist = FALSE)

set_stocnet_theme(theme = NULL, persist = FALSE)
```

Arguments

`theme`	String naming a theme. By default "default". The following themes are currently available: default, bw, crisp, neon, clay, iheid, ethz, uzh, rug, unibe, oxf, unige, cmu, iast, hwu, rainbow. This string can be capitalised or not.
`persist`	Logical, by default FALSE. If TRUE, the theme is remembered across sessions, by writing it to the user's configuration directory (see `tools::R_user_dir()`). Nothing is written to disk unless this is set explicitly. Use `stocnet_theme(persist = FALSE)` when setting a theme to forget a previously persisted choice.

Value

This function sets the theme and palette(s) to be used across all stocnet packages. The palettes are written to options and held there.

Fonts

Some themes also set a preferred font for use in plots, if available on the system (a check is performed). In some cases, this includes a vector of options to try in sequence. If none of the preferred fonts are available, a sans-serif font is used. Themes then look much more alike than they should, since the typeface carries a good deal of an institution\'s identity. Call `list_fonts()` to see which font families R can currently see, and `ag_font()` to see which one the current theme settled on.

To make more fonts available, there are two steps.

1. Install the font on your computer. Many of the fonts these themes prefer are free: Google Fonts (https://fonts.google.com) offers Roboto, Open Sans, Source Sans 3, Source Serif 4, Noto Serif, Montserrat, and Playfair Display, among others. Download the family, then install it as you would any other font: double-click the files and choose "Install" on Windows, open them in Font Book on macOS, or copy them into `~/.local/share/fonts` and run `fc-cache -f` on Linux. Some fonts are licensed and are only available to members of the institution concerned, or for purchase; the theme falls back to a near relative where it can.
2. Make the font available to R. Install the `{systemfonts}` package and the fonts installed on your system are found directly, with no further step. Otherwise, use `extrafont::font_import()` once and `extrafont::loadfonts()` in each session. Restart R after installing a font, then call `list_fonts()` to check that the family is now listed, and set the theme again.

Note that a font is only used where the graphics device can draw it. The `{ragg}` devices (for example `ragg::agg_png()`) and `{svglite}` are the most reliable; the default PDF device needs the font embedded, for which `extrafont::embed_fonts()` is available.

Custom

If you have specific needs or preferences, you can set your own palettes or overwrite part of an existing one using `options()`. For example, to set a custom base color, you can use: `options(snet_highlight = c("#1b9e77", "#d95f02", "#7570b3"))`. This will set a custom highlight color palette. Similarly, you can set `snet_div` for divergent palettes and `snet_cat` for categorical palettes.

See Also

Other themes: `check_colors`, `list_fonts()`, `theme_colorblind`, `theme_medium`

Examples

```
stocnet_theme("default")
plot(netrics::node_by_degree(ison_karateka))
stocnet_theme("uzh")
plot(netrics::node_by_degree(ison_karateka))
```

Index

∗ **MoNAn**
  plot_convergence, 44
  plot_gof, 45
∗ **RSiena**
  plot_gof, 45
  plot_interp, 59
∗ **datasets**
  made_earlier, 27
∗ **ergm**
  plot_convergence, 44
  plot_gof, 45
∗ **mapping**
  check_layout, 6
  completion, 10
  layout_concentric, 12
  layout_configuration, 13
  layout_correspondence, 14
  layout_layered, 17
  layout_levels, 20
  layout_matching, 21
  layout_scaling, 22
  layout_spatial, 24
  layout_valence, 25
  plot_graphr, 47
  plot_graphs, 52
  plot_grapht, 54
∗ **themes**
  check_colors, 4
  list_fonts, 26
  theme_colorblind, 60
  theme_medium, 62
  theme_set, 64

ag_base *(*ag_call*)*, 3
ag_call, 3
ag_divergent *(*ag_call*)*, 3
ag_font *(*ag_call*)*, 3
ag_highlight *(*ag_call*)*, 3
ag_ink *(*ag_call*)*, 3
ag_missing *(*ag_call*)*, 3
ag_negative *(*ag_call*)*, 3
ag_positive *(*ag_call*)*, 3
ag_qualitative *(*ag_call*)*, 3
ag_sequential *(*ag_call*)*, 3
ag_size *(*theme_medium*)*, 62

check_angles *(*check_layout*)*, 6
check_colors, 4, *26*, *61*, *63*, *65*
check_colors(), *9*, *61*
check_contrast *(*check_colors*)*, 4
check_contrast(), *3*, *4*, *61*
check_crossings *(*check_layout*)*, 6
check_drawing *(*check_layout*)*, 6
check_layout, 6, *11*, *13*, *14*, *16*, *19*, *21–24*, *26*, *52*, *53*, *58*
check_layout(), *6*
check_lengths *(*check_layout*)*, 6
check_offset *(*check_layout*)*, 6
check_separation *(*check_colors*)*, 4
check_separation(), *4*
check_slopes *(*check_layout*)*, 6
check_span *(*check_layout*)*, 6
check_stress *(*check_layout*)*, 6
completion, *9*, 10, *13*, *14*, *16*, *19*, *21–24*, *26*, *52*, *53*, *58*
count_pages, 11
count_pages(), *35*

ergm_gof *(*made_earlier*)*, 27

ggplot2::scale_x_date(), *32*
ggplot2::scale_y_date(), *32*
goldfish_changepoints *(*made_earlier*)*, 27
goldfish_fit *(*made_earlier*)*, 27
goldfish_gof *(*made_earlier*)*, 27
goldfish_margins *(*made_earlier*)*, 27
goldfish_onset *(*made_earlier*)*, 27
goldfish_outliers *(*made_earlier*)*, 27
goldfish_time *(*made_earlier*)*, 27
graphr *(*plot_graphr*)*, 47

graphr(), *63*
graphs *(*plot_graphs*)*, 52
grapht *(*plot_grapht*)*, 54

is_dark *(*theme_match*)*, 61

layout_concentric, 12
layout_concentric(), *9*, *11*, *14*, *16*, *19*, *21–24*, *26*, *52, 53*, *58*
layout_configuration, 13
layout_configuration(), *9*, *11*, *13*, *16*, *19*, *21–24*, *26*, *52, 53*, *58*
layout_correspondence, 14
layout_correspondence(), *9*, *11*, *13, 14*, *19*, *21–24*, *26*, *52, 53*, *58*
layout_dyad *(*layout_configuration*)*, 13
layout_hexad *(*layout_configuration*)*, 13
layout_ladder *(*layout_layered*)*, 17
layout_layered, 17
layout_layered(), *9*, *11*, *13, 14*, *16*, *21–24*, *26*, *52, 53*, *58*
layout_levels, 20
layout_levels(), *9*, *11*, *13, 14*, *16*, *19*, *22–24*, *26*, *52, 53*, *58*
layout_lineage *(*layout_layered*)*, 17
layout_matching, 21
layout_matching(), *9*, *11*, *13, 14*, *16*, *19*, *21*, *23, 24*, *26*, *52, 53*, *58*
layout_pentad *(*layout_configuration*)*, 13
layout_railway *(*layout_layered*)*, 17
layout_scaling, 22
layout_scaling(), *9*, *11*, *13, 14*, *16*, *19*, *21, 22*, *24*, *26*, *52, 53*, *58*
layout_spatial, 24
layout_spatial(), *9*, *11*, *13, 14*, *16*, *19*, *21–23*, *26*, *52, 53*, *58*
layout_tbl_graph_concentric *(*layout_concentric*)*, 12
layout_tbl_graph_configuration *(*layout_configuration*)*, 13
layout_tbl_graph_correspondence *(*layout_correspondence*)*, 14
layout_tbl_graph_ladder *(*layout_layered*)*, 17
layout_tbl_graph_layered *(*layout_layered*)*, 17
layout_tbl_graph_levels *(*layout_levels*)*, 20
layout_tbl_graph_lineage *(*layout_layered*)*, 17
layout_tbl_graph_matching *(*layout_matching*)*, 21
layout_tbl_graph_railway *(*layout_layered*)*, 17
layout_tbl_graph_scaling *(*layout_scaling*)*, 22
layout_tbl_graph_spatial *(*layout_spatial*)*, 24
layout_tbl_graph_valence *(*layout_valence*)*, 25
layout_tetrad *(*layout_configuration*)*, 13
layout_triad *(*layout_configuration*)*, 13
layout_valence, 25
layout_valence(), *9*, *11*, *13, 14*, *16*, *19*, *21–24*, *52, 53*, *58*
list_fonts, 26
list_fonts(), *6*, *61*, *63*, *65*
load_ergm_res *(*plot_convergence*)*, 44

made_earlier, 27
map_measure, 29
map_member, 30
map_motifs, 31
match_color *(*theme_match*)*, 61
match_color(), *5*
mdate_scales, 31
model_mrqap, 32
monan_conv *(*made_earlier*)*, 27
monan_gof *(*made_earlier*)*, 27

plot(), *11*
plot.ag_conv *(*plot_convergence*)*, 44
plot.ag_gof *(*plot_gof*)*, 45
plot.changepoints.goldfish *(*plot.diagnose_outliers*)*, 33
plot.diagnose_changepoints *(*plot.diagnose_outliers*)*, 33
plot.diagnose_outliers, 33
plot.diff_model *(*plot.diffusion*)*, 36
plot.diffs_model *(*plot.diffusion*)*, 36
plot.diffusion, 36
plot.diffusionModel *(*plot.diffusion*)*, 36
plot.diffusionSweep *(*plot.diffusion*)*, 36
plot.diffusionTransitions *(*plot.transitions*)*, 43
plot.ergm *(*plot_convergence*)*, 44
plot.gof.ergm *(*plot_gof*)*, 45

plot.gof.stats.monan *(plot_gof)*, 45
plot.goldfishChangepoints *(plot.diagnose_outliers)*, 33
plot.goldfishFit *(plot.result.goldfish)*, 42
plot.goldfishGOF *(plot.diagnose_outliers)*, 33
plot.goldfishMargins *(plot.diagnose_outliers)*, 33
plot.goldfishOnset *(plot.diagnose_outliers)*, 33
plot.goldfishOutliers *(plot.diagnose_outliers)*, 33
plot.goldfishTimeTest *(plot.diagnose_outliers)*, 33
plot.infernetConfusion, 38
plot.influenceTable *(plot_interp)*, 59
plot.learn_model *(plot.diffusion)*, 36
plot.matrix *(map_member)*, 30
plot.mobility, 39
plot.mobilityModel *(plot.mobility)*, 39
plot.netlm *(model_mrqap)*, 32
plot.netlogit *(model_mrqap)*, 32
plot.netreg *(model_mrqap)*, 32
plot.network_measure *(map_measure)*, 29
plot.network_measures *(map_measure)*, 29
plot.network_motif *(map_motifs)*, 31
plot.network_test, 40
plot.node_measure *(map_measure)*, 29
plot.node_member *(map_member)*, 30
plot.node_motif *(map_motifs)*, 31
plot.opinion, 41
plot.opinionModel *(plot.opinion)*, 41
plot.opinionSweep *(plot.opinion)*, 41
plot.outliers.goldfish *(plot.diagnose_outliers)*, 33
plot.result.goldfish, 42
plot.selectionTable *(plot_interp)*, 59
plot.sienaGOF *(plot_gof)*, 45
plot.tie_measure *(map_measure)*, 29
plot.traces.monan *(plot_convergence)*, 44
plot.transitions, 43
plot_adequacy *(plot.diagnose_outliers)*, 33
plot_convergence, 44, *47*
plot_gof, *45*, 45, *60*
plot_goldfish_fit *(plot.result.goldfish)*, 42
plot_graphr, *9*, *11*, *13*, *14*, *16*, *19*, *21–24*, *26*, 47, *53*, *58*
plot_graphs, *9*, *11*, *13*, *14*, *16*, *19*, *21–24*, *26*, *52*, 52, *58*
plot_grapht, *9*, *11*, *13*, *14*, *16*, *19*, *21–24*, *26*, *52*, *53*, 54
plot_interp, *47*, 59
print.grapht *(plot_grapht)*, 54

res_infernet_reg *(made_earlier)*, 27
res_manynet_diff *(made_earlier)*, 27
res_migraph_diff *(made_earlier)*, 27
res_migraph_reg *(made_earlier)*, 27
res_migraph_test *(made_earlier)*, 27
res_netrics_robust *(made_earlier)*, 27

scale_x_mdate *(mdate_scales)*, 31
scale_y_mdate *(mdate_scales)*, 31
set_completion *(completion)*, 10
set_stocnet_medium *(theme_medium)*, 62
set_stocnet_theme *(theme_set)*, 64
siena_gof *(made_earlier)*, 27
siena_influence *(made_earlier)*, 27
siena_selection *(made_earlier)*, 27
simulate_colorblind *(theme_colorblind)*, 60
simulate_colorblind(), *4*, *5*
stocnet_completion *(completion)*, 10
stocnet_medium *(theme_medium)*, 62
stocnet_medium(), *3*
stocnet_theme *(theme_set)*, 64

theme_colorblind, *6*, *26*, 60, *63*, *65*
theme_match, 61
theme_medium, *6*, *26*, *61*, 62, *65*
theme_set, *6*, *26*, *61*, *63*, 64