Package ‘manynet’

October 14, 2026

Title Many Ways to Make, Manipulate, and Modify Myriad Networks

Version 2.4.0

Description Many tools for making, manipulating, and modifying many different types of networks.
All functions operate with matrices, edge lists, and 'igraph', 'network', and 'tidygraph' objects,
on directed, multiplex, multimodal, signed, and other networks.
The package includes functions for importing and exporting, creating and generating networks,
modifying networks and node and tie attributes,
and describing networks with sensible defaults.

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

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

License MIT + file LICENSE

Language en-GB

Encoding UTF-8

LazyData true

Depends R (>= 4.1.0)

Imports cli, dplyr (>= 1.1.0), igraph (>= 2.1.0), network, pillar,
tidygraph

Suggests jsonlite, methods, readxl, RSiena, sna, testthat (>= 3.0.0),
tibble, xml2

Config/Needs/build roxygen2, devtools

Config/Needs/check covr, lintr, spelling

Config/Needs/website pkgdown, learnr

Config/testthat/parallel true

Config/testthat/edition 3

Config/testthat/start-first tutorials_manynet, mark_is

Config/roxygen2/version 8.1.1

NeedsCompilation no

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

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

Repository CRAN

Date/Publication 2026-10-14 08:40:22 UTC

Contents

class_describe . . . 4
coerce_graph . . . 5
coerce_list . . . 9
coerce_matrix . . . 12
data_overview . . . 14
fict_actually . . . 14
fict_friends . . . 16
fict_greys . . . 17
fict_lotr . . . 18
fict_marvel . . . 19
fict_potter . . . 21
fict_starwars . . . 22
fict_thrones . . . 24
glossary . . . 25
interface . . . 26
irps_911 . . . 27
irps_blogs . . . 28
irps_books . . . 30
irps_corruption . . . 31
irps_nuclear . . . 32
irps_revere . . . 34
irps_supremecourt . . . 35
irps_tribes . . . 36
irps_usgeo . . . 38
irps_wwi . . . 39
ison_adolescents . . . 40
ison_algebra . . . 41
ison_bankwiring . . . 42
ison_brandes . . . 44
ison_classmates . . . 45
ison_dolphins . . . 47
ison_emotions . . . 48
ison_florentine . . . 49
ison_fraternity . . . 51
ison_hightech . . . 52

ison_judo_moves . . . 54
ison_karateka . . . 55
ison_koenigsberg . . . 57
ison_laterals . . . 58
ison_lawfirm . . . 60
ison_monks . . . 62
ison_networkers . . . 64
ison_physicians . . . 65
ison_southern_women . . . 68
ison_tailorshop . . . 70
make_collect . . . 71
make_create . . . 73
make_ego . . . 77
make_emails . . . 78
make_explicit . . . 80
make_groups . . . 80
make_growth . . . 85
make_learning . . . 87
make_mnet . . . 89
make_play . . . 91
make_random . . . 94
make_read . . . 98
make_stocnet . . . 101
make_write . . . 108
manip_changes . . . 111
manip_globals . . . 113
manip_info . . . 115
manip_nodes_attr . . . 117
manip_nodes_num . . . 119
manip_ties_attr . . . 121
manip_ties_num . . . 123
mark_features . . . 125
mark_format_change . . . 128
mark_format_cognitive . . . 129
mark_format_node . . . 131
mark_format_tie . . . 133
mark_is . . . 135
measure_attributes_nodes . . . 136
measure_attributes_ties . . . 137
measure_dims . . . 140
measure_missingness . . . 142
member_names . . . 143
modif_backbone . . . 144
modif_concepts . . . 147
modif_direction . . . 149
modif_from_parts . . . 151
modif_from_ternary . . . 152
modif_labels . . . 155

modif_levels 156
modif_miss 157
modif_motifs 160
modif_paths 162
modif_permutation 165
modif_plexity 166
modif_project 169
modif_proximity 175
modif_scope 180
modif_split 183
modif_weight 186
progress 189

Index **191**

`class_describe` *Describe a network*

Description

These functions are used to describe components of a given network in terms of a particular phrase.

- `describe_network()` describes the features or properties of a network, such as whether it is two-mode, directed, or complex.
- `describe_nodes()` describes how many of each type of nodes there are and, if available, names the different nodesets or modes.
- `describe_ties()` describes how many of each type of ties there are and, if available, names the different types of ties.
- `describe_changes()` describes the changing features of a network, if any, such as how many waves there are.
- `describe_transformations()` describes how the network has been transformed since it was collected or generated, if at all.

These descriptions are constructed to be GRAND-consistent.

Usage

```
describe_network(.data)

describe_nodes(.data)

describe_ties(.data)

describe_changes(.data)

describe_transformations(.data, details = FALSE, width = cli::console_width())
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`details` Logical. Where FALSE, the default, the description is given at whichever level of detail fits the width of the console. Where TRUE, every method and its consequence is named whatever the width.

`width` Integer. The number of characters the description may fill, by default the width of the console. Where a caller adds words of its own, such as the header `print()` puts this in, it passes what is left.

Examples

```
describe_transformations(to_undirected(ison_southern_women))
```

---

coerce_graph *Coercing into graph objects*

Description

The `as_` functions in {manynet} coerce objects of any of the following common classes of social network objects in R into the declared class:

- `as_igraph()` coerces the object into an {igraph} `graph` object.
- `as_tidygraph()` coerces the object into a {tidygraph} `tbl_graph` object.
- `as_network()` coerces the object into a {network} `network` object.
- `as_siena()` coerces the (igraph/tidygraph) object into a SIENA dependent variable.
- `as_graphAM()` coerces the object into a graph adjacency matrix.
- `as_diffusion()` coerces a table of diffusion events into a `diff_model` object similar to the output of `play_diffusion()`.
- `as_diffnet()` coerces a `diff_model` object into a {netdiffuseR} `diffnet` object.

An effort is made for all of these coercion routines to be as lossless as possible, though some object classes are better at retaining certain kinds of information than others. Note also that there are some reserved column names in one or more object classes, which could otherwise lead to some unexpected results.

Usage

```
as_igraph(.data, twomode = FALSE)

as_tidygraph(.data, twomode = FALSE)

as_network(.data, twomode = FALSE)

as_stocnet(.data, twomode = FALSE, ...)

## S3 method for class 'node_measure'
as_stocnet(
  .data,
  twomode = FALSE,
  compare = c("absdiff", "diff", "sender", "receiver"),
  ...
)

## S3 method for class 'node_member'
as_stocnet(.data, twomode = FALSE, compare = "same", ...)

as_hypergraph(.data)

## S3 method for class 'array'
as_stocnet(
  .data,
  twomode = FALSE,
  ...,
  attribute = c("time", "by", "about", "layer")
)

as_siena(.data, twomode = FALSE)

as_graphAM(.data, twomode = NULL)

as_diffusion(.data, twomode = FALSE, events)

as_diffnet(.data, twomode = FALSE)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`twomode` Logical option used to override heuristics for distinguishing incidence (two-mode/bipartite) from adjacency (one-mode/unipartite) networks. By default FALSE.

`...` Further arguments passed to or from other methods, such as `attribute` for an array.

`compare` For a node measure or a node membership, how the values of two nodes are compared to give the tie between them; see `as_matrix()`. Comparing by "same" or "absdiff" gives an undirected network, and comparing by "diff", "sender", or "receiver" a directed one. A comparison of zero is no tie.

`attribute` For a three-dimensional array, what its third dimension holds, which becomes a column of the ties: "time" for the waves of a panel, "by" for the reporters of a cognitive social structure, "about" for the targets of gossip, or "layer" for the layers of a multiplex network. An array cannot say which of these it holds. Where its third dimension is as long as the network has nodes, it could hold any of them, so this must be given. Otherwise it cannot hold reporters or targets, and "time" is assumed.

`events` A table (data frame or tibble) of diffusion events with columns `t` indicating the time (typically an integer) of the event, `nodes` indicating the number or name of the node involved in the event, and `event`, which can take on the values "I" for an infection event, "E" for an exposure event, or "R" for a recovery event.

Value

The currently implemented coercions or translations are:

```
              array data.frame default diff_model diffnet hypergraph igraph
as_diffnet                                     *
as_diffusion                                   *       *                *
as_graphAM                          *
as_hypergraph                       *                            *
as_igraph                   *                  *       *                *
as_network                  *                          *                *
as_siena                            *
as_stocnet        *         *                                    *      *
as_tidygraph                *                  *       *                *
              list matrix mnet network network.goldfish networkDynamic
as_diffnet
as_diffusion                *
as_graphAM             *
as_hypergraph
as_igraph              *            *                *              *
as_network             *            *                *              *
as_siena
as_stocnet             *            *
as_tidygraph     *     *            *                *              *
              node_measure node_member siena sienadata stocnet tbl_graph
as_diffnet
as_diffusion                                                 *
as_graphAM
```

as_hypergraph						
as_igraph			*	*	*	*
as_network			*	*	*	*
as_siena					*	
as_stocnet	*	*		*	*	
as_tidygraph			*	*	*	*

`as_diffusion()` and `play_diffusion()` return a 'diff_model' object that contains two different tibbles (tables) – a table of diffusion events and a table of the number of nodes in each relevant component (S, E, I, or R) – as well as a copy of the network upon which the diffusion ran. By default, a compact version of the component table is printed (to print all the changes at each time point, use `print(..., verbose = T)`). To retrieve the diffusion events table, use `summary(...)`.

Hypergraphs

`as_hypergraph()` returns a 'hypergraph' object of the {HyperG} package, which is a sparse incidence matrix of hyperedges by nodes, and `as_stocnet()` reads one. A network that is not a hypergraph is reformatted as one first, as by `to_hypergraph()`. A 'hypergraph' object holds which nodes each hyperedge has and their names, and nothing else, so weights, attributes, directions, and times are not carried. Neither function needs {HyperG} to be installed.

Note that `is_hypergraph()` marks a network of any class that holds hyperedges, of which a 'stocnet' object holds the most (see the Hypergraphs section of `make_stocnet()`).

See Also

Other coercions: `coerce_list`, `coerce_matrix`

Examples

```
test <- data.frame(from = c("A","B","B","C","C"), to = c("I","G","I","G","H"))
as_edgelist(test)
as_matrix(test)
as_igraph(test)
as_tidygraph(test)
as_network(test)
as_hypergraph(create_ring(4))
  # How to create a diff_model object from (basic) observed data
  events <- data.frame(time = c(0,1,1,2,3),
                       node = c(1,2,3,2,4),
                       var = "diffusion",
                       value = c("I","I","I","R","I"))
  bind_changes(create_filled(4), events)
```

coerce_list *Coercing into lists*

Description

These functions coerce objects into different objects by extracting and translating the information contained in the original object:

- `as_edgelist()` coerces the object into an edgelist, as data frames or tibbles.
- `as_nodelist()` coerces the object into a nodelist, as a data frame or tibble.
- `as_changelist()` coerces the object into a changelist, as a data frame or tibble.
- `as_globallist()` coerces the object into a globallist, as a data frame or tibble.
- `as_missinglist()` coerces the object into a list of the ties it records as missing, as a tibble.
- `as_infolist()` coerces the object into a list of network-level information, such as the names of the nodes and ties, if not given in the nodelist or edgelist.

These coercions are extractive in the sense that they will lose any information that cannot be contained in the target format. For coercion into a matrix, see `as_matrix()`.

Usage

```
as_nodelist(.data)

as_changelist(.data, time = NULL)

as_edgelist(.data, twomode = FALSE)

as_infolist(.data)

as_globallist(.data)

as_missinglist(.data)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`time` A moment to gather the changes in force at. By default `NULL`, in which case every change the network records is returned. Where a moment is given, only the changes recorded up to and including it are returned, one for each nodal variable a node changes, since a change states what a variable becomes from that moment on and is carried forward until another change states otherwise.

`twomode` Logical option used to override heuristics for distinguishing incidence (two-mode/bipartite) from adjacency (one-mode/unipartite) networks. By default FALSE.

Details

Edgelists are expected to be held in data.frame or tibble class objects. The first two columns of such an object are expected to be the senders and receivers of a tie, respectively, and are typically named "from" and "to" (even in the case of an undirected network). These columns can contain integers to identify nodes or character strings/factors if the network is labelled. If the sets of senders and receivers overlap, a one-mode network is inferred. If the sets contain no overlap, a two-mode network is inferred. If a third, numeric column is present, a weighted network will be created.

Value

The currently implemented coercions or translations are:

```
               data.frame default igraph matrix network network.goldfish siena
as_changelist           *             *      *       *
as_edgelist             *             *      *       *                *     *
as_globallist           *             *      *       *
as_infolist             *             *      *       *
as_missinglist          *       *     *      *       *
as_nodelist             *             *      *       *
               stocnet tbl_graph
as_changelist        *         *
as_edgelist          *         *
as_globallist        *
as_infolist          *         *
as_missinglist       *         *
as_nodelist          *         *
```

Missing ties

A missing tie is one that could have been observed and was not, which is neither a tie nor the absence of one. `as_missinglist()` returns them as a tibble of 'from' and 'to', with 'layer' and 'time' where the network records them.

Compared to an **observed** tie, in which a tie is observed to be present or absent, with only present ties appearing as rows in the ties component, a **missing** tie is one that could have been observed but was not. There are four different reasons a tie could be missing, and they are treated differently depending on the context.

First, there are the nodal reasons a tie could be missing, which are recorded in the nodes and changes components.

1. **Non-availability**. A node was not in the network and so cannot send or receive ties. This is recorded in the 'active' column of the nodes component, and can change over time through the changes component. Note that this renders all outgoing and incoming ties missing.
2. **Non-response**. A node was in the network but chose not to respond or report its ties. This is recorded in the 'na' column of the nodes component, and can also change over time through the changes component. Note that this renders only outgoing ties missing for a directed layer, and both directions for an undirected layer.

Second, a tie might be missing or missing some information, even though both nodes were in the network. This is recorded in the ties.

1. **Unobserved tie**. One or more specific ties could have been reported and were not, such as one name a respondent skipped. These are held in the missings component, as a tibble of 'from' and 'to', with 'layer' and 'time' where available.
2. **Unobserved weight**. One or more ties are reported as existing, but the strength of the tie is not known. This is recorded as an `NA` in the 'weight' column of the ties component.

Note that missing ties are not ties. `net_ties()` does not count them, `as_edgelist()` does not return them, and they are not drawn or measured unless a function asks for them by name. Where necessary, `as_missinglist()` returns a list of all missing ties together. `net_tie_missing()` reports how many there are, and `impute_ties()` imputes them.

Each class holds them differently. A stocnet object records which nodes did not report, from which `as_missinglist()` derives the ties; see `make_stocnet()` for how those records are held. So, unlike the other `as_*list()` functions, this one does not return a component verbatim: it returns the missings component together with the ties that the non-responsive nodes imply, which is nearly always the larger part. An 'igraph' or 'tbl_graph' object carries the list in a graph attribute, since igraph has no way to mark an edge as missing. A 'network' object holds each as an edge marked in the reserved 'na' attribute, which is that package's own format and the one `{ergm}` expects. A matrix holds each as a missing cell.

See Also

Other coercions: `coerce_graph`, `coerce_matrix`

Examples

```
test <- data.frame(from = c("A","B","B","C","C"), to = c("I","G","I","G","H"))
as_edgelist(test)
as_igraph(test)
as_tidygraph(test)
as_network(test)
as_missinglist(ison_classmates)
```

coerce_matrix *Coercing into matrices*

Description

as_matrix() coerces the object into an adjacency (one-mode/unipartite) or incidence (two-mode/bipartite) matrix.

If the network is a cognitive social structure or an egocentric network (i.e. the edgelist contains a 'by' column naming who reported each tie), or a gossip network (with an 'about' column naming whom each tie is about), as_matrix() returns a three-dimensional array instead, with dimensions for senders, receivers, and reporters (or targets). Every node takes a row, a column, and a slice, in the order of the nodes, and a tie recorded as missing holds NA. Where such a network has several layers, a list of one array for each layer is returned.

Where a network holds parallel ties, i.e. where tie_is_parallel() is TRUE for any tie, the cells of the matrix report how many ties join each pair of nodes, and so may be greater than one even where the network is neither weighted nor signed.

as_matrix() also turns a node measure or a node membership, as the node_*() functions and node_attribute() return them, into a matrix of the pairs of nodes, compared as compare says. This gives the dyadic covariates of an MRQAP model, such as those migraph::net_regression() builds.

This coercion is extractive in the sense that it loses any information that a matrix cannot hold, such as tie attributes other than the weight, or the direction of a two-mode network.

Usage

```
as_matrix(.data, twomode = NULL, ...)

## S3 method for class 'node_measure'
as_matrix(
  .data,
  twomode = NULL,
  compare = c("absdiff", "diff", "sender", "receiver"),
  ...
)

## S3 method for class 'node_member'
as_matrix(.data, twomode = NULL, compare = "same", ...)
```

Arguments

.data An object of a {manynet}-consistent class:

- adjacency or incidence matrix from {base} R
- edgelist data.frame from {base} R or tbl/tbl_df from {tibble}
- stocnet stocnet, from the {manynet} package
- igraph igraph, from the {igraph} package

- network `network`, from the `{network}` package
- tidygraph `tbl_graph`, from the `{tidygraph}` package

`twomode` Logical option used to override heuristics for distinguishing incidence (two-mode/bipartite) from adjacency (one-mode/unipartite) networks. By default FALSE.

`...` Other arguments passed to or from other methods.

`compare` How the values of two nodes are compared in each cell of the matrix returned for a node measure or a node membership. For a node measure `x`:

- "absdiff" (the default) gives `abs(x[i] - x[j])`,
- "diff" gives `x[i] - x[j]`, which `t()` turns into `x[j] - x[i]`,
- "sender" gives `x[i]`, the value of the node in the row,
- "receiver" gives `x[j]`, the value of the node in the column.

For a node membership, "same" gives 1 where the two nodes belong to the same group and 0 where they do not. The diagonal of a one-mode result is 0, and a missing value gives a missing cell. Where the vector comes from a two-mode network, the rows are the nodes of the first mode and the columns the nodes of the second.

Details

Matrices can be either adjacency (one-mode) or incidence (two-mode) matrices. Incidence matrices are typically inferred from unequal dimensions, but since in rare cases a matrix with equal dimensions may still be an incidence matrix, an additional argument `twomode` can be specified to override this heuristic.

Value

The currently implemented coercions or translations are:

```
          data.frame diff_model igraph matrix network network.goldfish
as_matrix          *          *      *      *       *                *
          node_measure node_member siena stocnet tbl_graph
as_matrix            *           *     *       *         *
```

See Also

Other coercions: `coerce_graph`, `coerce_list`

Examples

```
test <- data.frame(from = c("A","B","B","C","C"), to = c("I","G","I","G","H"))
as_matrix(test)
as_matrix(node_attribute(ison_lawfirm, "age"))[1:5, 1:5]
as_matrix(node_attribute(ison_lawfirm, "age"), compare = "diff")[1:5, 1:5]
```

data_overview — *Obtain overview of available network data*

Description

This function makes it easy to get an overview of available data:

- `table_data()` returns a tibble with details of the network datasets included in the packages.

Usage

```
table_data(..., pkg = c("manynet", "migraph"))
```

Arguments

...	Network marks, e.g. directed, twomode, or signed, that are used to filter the results.
pkg	String, name of the package.

Examples

```
table_data()
# to obtain list of all e.g. directed networks:
table_data(pkg = "manynet", directed)
# to obtain overview of unique datasets:
table_data() |>
  dplyr::distinct(directed, weighted, twomode, signed,
                  .keep_all = TRUE)
```

fict_actually — *Multilevel network of Love Actually characters, their relationships, and their scene appearances (Robinson 2015)*

Description

Love Actually is a 2003 British romantic comedy film. David Robinson's data is a two-mode network of 20 characters and their appearances across 76 scenes, as parsed from the script.

Added to this are multiplex, one-mode networks of character relationships, including "romantic", "family", "friendship", and "professional" ties. These were added by Korakot Janteerasakul from the following source: https://en.wikipedia.org/wiki/Love_Actually#/media/File:Love_Actually_(2003)_Interconnections.svg.

No tie is directed. Unlike `fict_marvel`, the layers here are not simply the levels: four of the five run within the characters, so the network is multiplex as well as multilevel, and `describe_network()` reports both.

Usage

```
data(fict_actually)
```

Format

```
#> -- # Love Actually ------------------------------------------------------------
#> # A labelled, multiplex, multilevel network of 20 characters and 76 scenes and
#> 7 romance ties, 162 appearance ties, 7 friendship ties, 10 professional ties,
#> and 3 family ties
#>
#> -- Nodes
#> # A tibble: 96 x 6
#>   label   mode       Actor          Gender Nationality Occupation
#>   <chr>   <chr>      <chr>          <chr>  <chr>       <chr>
#> 1 Aurelia characters Lúcia Moniz    Female Portuguese  Assistant
#> 2 Billy   characters Bill Nighy     Male   British     Singer
#> 3 Colin   characters Kris Marshall  Male   British     Travels to America
#> 4 Daniel  characters Liam Neeson    Male   British     Widower
#> 5 Harry   characters Alan Rickman   Male   British     Agency Director
#> 6 John    characters Martin Freeman Male   British     Nude stand-in
#> # i 90 more rows
#>
#> -- Ties
#> # A tibble: 189 x 3
#>    from    to layer
#>   <int> <int> <chr>
#> 1     1     7 romance
#> 2     1    51 appearance
#> 3     1    57 appearance
#> 4     1    58 appearance
#> 5     1    63 appearance
#> 6     1    95 appearance
#> # i 183 more rows
#>
```

Author(s)

David Robinson

References

Robinson, David. 2015. "Analyzing networks of characters in 'Love Actually'". http://varianceexplained.org/r/love-actually-network/

`fict_friends` *One-mode undirected Friends character scene co-appearances (McNulty, 2020)*

Description

One-mode network collected by McNulty (2020) on the connections between the Friends TV series characters from Seasons 1 to 10. The series aired from 1994 to 2004, and the network was collected from the scripts of its episodes. `fict_friends` is an undirected network of 650 named characters (e.g. Ross, Rachel, Monica, Chandler, Joey, and Phoebe). Each tie joins a pair of characters that appear in the same scene, and the 'weight' of the tie is the number of scenes, across all ten seasons, in which the pair appears together.

Usage

```
data(fict_friends)
```

Format

```
#> -- # Friends ----------------------------------------------------------------
#> # A labelled, weighted, undirected network of 650 characters and 2976 scene
#> co-appearance ties
#>
#> -- Nodes
#> # A tibble: 650 x 1
#>   label
#>   <chr>
#> 1 Actor
#> 2 Alan
#> 3 Andrea
#> 4 Angela
#> 5 Aunt Iris
#> 6 Aunt Lillian
#> # i 644 more rows
#>
#> -- Ties
#> # A tibble: 2,976 x 3
#>    from    to weight
#>   <int> <int>  <dbl>
#> 1     8     9      3
#> 2     4    10      1
#> 3     8    12      1
#> 4     9    12      1
#> 5     2    14      1
#> 6     3    14      1
#> # i 2,970 more rows
#>
```

References

McNulty, Keith. 2020. "Network analysis of Friends scripts". https://github.com/keithmcnulty/friends_analysis/

---

fict_greys — *One-mode undirected network of characters hook-ups on Grey's Anatomy TV show*

Description

Grey's Anatomy is an American medical drama television series running on ABC since 2005. It focuses on the personal and professional lives of surgical interns, residents, and attendings at Seattle Grace Hospital, later renamed as the Grey Sloan Memorial Hospital. Gary Weissman collected data on the sexual contacts between characters on the television show through observation of the story lines in the episodes and fan pages, and this data was extended by Benjamin Lind including nodal attributes:

- 'name': first and, where available, surname
- 'sex': `F` for female and `M` for male
- 'race': `White`, `Black`, or `Other`
- 'birthyear': year born (some missing data)
- 'position': `"Chief"`, `"Attending"`, `"Resident"`, `"Intern"`, `"Nurse"`, `"Non-Staff"`, `"Other"`
- 'season': season that the character joined the show
- 'sign': character's astrological starsign, if known

The characters are those that joined the show in its first ten seasons. A tie records that two characters had a sexual contact, and is neither directed nor weighted.

Usage

```
data(fict_greys)
```

Format

```
#> -- # Grey's Anatomy ------------------------------------------------------------
#> # A labelled, undirected network of 53 characters and 56 hook-up ties
#>
#> -- Nodes
#> # A tibble: 53 x 7
#>   label              sex   race  birthyear position  season sign
#>   <chr>              <chr> <chr>     <dbl> <chr>      <dbl> <chr>
#> 1 Addison Montgomery F     White      1967 Attending      1 Libra
#> 2 Adele Webber       F     Black      1949 Non-Staff      2 Leo
#> 3 Teddy Altman       F     White      1969 Attending      6 Pisces
#> 4 Amelia Shepherd    F     White      1981 Attending      7 Libra
```

```
#> 5 Arizona Robbins    F     White      1976 Attending      5 Leo
#> 6 Rebecca Pope       F     White      1975 Non-Staff      3 Gemini
#> # i 47 more rows
#>
#> -- Ties
#> # A tibble: 56 x 2
#>    from    to
#>   <int> <int>
#> 1     5    47
#> 2    21    47
#> 3     5    46
#> 4     5    41
#> 5    18    41
#> 6    21    41
#> # i 50 more rows
#>
```

Author(s)

Gary Weissman and Benjamin Lind

References

Weissman, Gary. "Grey's Anatomy network of sexual relations". https://gweissman.github.io/post/grey-s-anatomy-network-of-sexual-relations/

Lind, Benjamin. 2012. "Lessons on exponential random graph modeling from Grey's Anatomy hook-ups". http://badhessian.org/2012/09/lessons-on-exponential-random-graph-modeling-from-greys-ana

---

`fict_lotr` *One-mode network of Lord of the Rings character interactions*

Description

The Lord of the Rings is a beloved, epic high fantasy novel written by J.R.R. Tolkien. This is a network of 36 Lord of the Rings book characters and 66 interactional relationships.

The ties are unweighted and concern only interaction. Interaction can be cooperative or conflictual.

In addition, the race of these characters has been coded, though not without debate. The most contentious is the coding of Tom Bombadil and Goldberry as Maiar, presumably coded as such to avoid having categories of one.

Usage

```
data(fict_lotr)
```

Format

```
#> -- # Lord of the Rings -------------------------------------------------------
#> # A labelled, complex, undirected network of 36 characters and 66 interaction
#> ties
#>
#> -- Nodes
#> # A tibble: 36 x 2
#>   name     Race
#>   <chr>    <chr>
#> 1 Aragorn  Human
#> 2 Beregond Human
#> 3 Bilbo    Hobbit
#> 4 Celeborn Elf
#> 5 Denethor Human
#> 6 Elladan  Elf
#> # i 30 more rows
#>
#> -- Ties
#> # A tibble: 66 x 2
#>    from    to
#>   <int> <int>
#> 1     1     7
#> 2     1     8
#> 3     5     9
#> 4     1    10
#> 5     3    10
#> 6     9    10
#> # i 60 more rows
#>
```

---

fict_marvel — *Multilevel signed network of Marvel comic book characters and their teams (Yuksel 2017)*

Description

This multilevel network ties 53 Marvel *comic book* characters to each other and to the teams they belong to. The "affiliation" ties offer a two-mode affiliation network of 53 Marvel comic book characters and their affiliations to 141 teams. The "relationship" ties offer a one-mode signed network of friendships and enmities between the 53 Marvel comic book characters. Friendships are indicated by a positive sign in the tie `sign` attribute, whereas enmities are indicated by a negative sign in this edge attribute.

The two layers are the two levels of the network: the relationships run within the characters, and the affiliations between the characters and the teams. Since the layers are the levels, `describe_network()` reports the network as multilevel and not also as multiplex or two-mode, though `is_multiplex()` and `is_twomode()` both mark it TRUE.

Note that 69 pairs of characters hold two relationship ties rather than one, which `tie_is_parallel()` marks, covering 138 of the 1241 ties. Since `as_matrix()` sums the signs of the ties joining a pair of nodes, the 68 pairs holding both a friendship and an enmity report 0 in the matrix, which is indistinguishable from an absent tie, and the one pair holding two enmities reports -2. Read these relationships from the ties table rather than from a matrix.

Usage

```
data(fict_marvel)
```

Format

```
#>
#> -- # Marvel universe ------------------------------------------------------------
#> # A labelled, complex, signed, multilevel network of 53 characters and 141
#> teams and 558 relationship ties and 683 affiliation ties (138 parallel)
#>
#> -- Nodes
#> # A tibble: 194 x 11
#>   label       mode   Gender Appearances Attractive  Rich Intellect Omnilingual
#>   <chr>       <chr>  <chr>       <int>     <int> <int>    <int>      <int>
#> 1 Abomination  chara~ Male          427         0     0        1          1
#> 2 Ant-Man      chara~ Male          589         1     0        1          0
#> 3 Apocalypse   chara~ Male         1207         0     0        1          1
#> 4 Beast        chara~ Male         7609         1     0        1          0
#> 5 Black Panther chara~ Male         2189         1     1        1          0
#> 6 Black Widow  chara~ Female       2907         1     0        1          0
#> # i 188 more rows
#> # i 3 more variables: PowerOrigin <chr>, UnarmedCombat <int>, ArmedCombat <int>
#>
#> -- Ties
#> # A tibble: 1,241 x 4
#>    from    to weight layer
#>   <int> <int>  <dbl> <chr>
#> 1     1     1     -1 relationship
#> 2     1     4     -1 relationship
#> 3     1    11     -1 relationship
#> 4     1    12     -1 relationship
#> 5     1    23     -1 relationship
#> 6     1    24     -1 relationship
#> # i 1,235 more rows
#>
```

Details

Additional nodal variables have been coded and included by Dr Umut Yuksel:

- **Gender**: binary character, 43 "Male" and 10 "Female"

- **PowerOrigin**: binary character, 2 "Alien", 1 "Cyborg", 5 "God/Eternal", 22 "Human", 1 "Infection", 16 "Mutant", 5 "Radiation", 1 "Robot"
- **Appearances**: integer, in how many comic book issues they appeared in
- **Attractive**: binary integer, 41 1 (yes) and 12 0 (no)
- **Rich**: binary integer, 11 1 (yes) and 42 0 (no)
- **Intellect**: binary integer, 39 1 (yes) and 14 0 (no)
- **Omnilingual**: binary integer, 8 1 (yes) and 45 0 (no)
- **UnarmedCombat**: binary integer, 51 1 (yes) and 2 0 (no)
- **ArmedCombat**: binary integer, 25 1 (yes) and 28 0 (no)

See also https://graphics.straitstimes.com/STI/STIMEDIA/Interactives/2018/04/marvel-cinematic-universe-whos-who-interactive/index.html.

Source

Umut Yuksel, 31 March 2017

`fict_potter` *Six complex one-mode support data in Harry Potter books (Bossaert and Meidert 2013)*

Description

Goele Bossaert and Nadine Meidert coded peer support ties among 64 characters in the Harry Potter books. Each author coded four of seven books using NVivo, with the seventh book coded by both and serving to assess inter-rater reliability. The first six books concentrated on adolescent interactions, were studied in their paper, and are made available here. The peer support ties mean voluntary emotional, instrumental, or informational support, or praise from one living, adolescent character to another within the book's pages. In addition, nodal attributes name, schoolyear (which doubles as their age), gender, and their house assigned by the sorting hat are included.

Usage

```
data(fict_potter)
```

Format

```
#> -- # Harry Potter support network ---------------------------------------------
#> # A longitudinal, labelled, complex, directed network of 64 students and 544
#> support arcs over 6 waves
#>
#> -- Nodes
#> # A tibble: 64 x 5
#>   label             schoolyear gender house      active
#>   <chr>                  <int> <chr>  <chr>      <logi>
#> 1 Adrian Pucey            1989 male   Slytherin  TRUE
```

```
#> 2 Alicia Spinnet          1989 female Gryffindor TRUE
#> 3 Angelina Johnson        1989 female Gryffindor TRUE
#> 4 Anthony Goldstein       1991 male   Ravenclaw  TRUE
#> # i 60 more rows
#>
#> -- Changes
#> # A tibble: 81 x 4
#>    time  node var         value
#>   <int> <int> <chr>      <list>
#> 1     2     9 active  TRUE<lgl>
#> 2     2    21 active  TRUE<lgl>
#> 3     2    35 active  TRUE<lgl>
#> 4     2    39 active FALSE<lgl>
#> # i 77 more rows
#>
#> -- Ties
#> # A tibble: 544 x 3
#>    from    to  time
#>   <int> <int> <dbl>
#> 1    11    11     1
#> 2    11    25     1
#> 3    11    26     1
#> 4    11    44     1
#> # i 540 more rows
#>
```

References

Bossaert, Goele and Nadine Meidert (2013). "'We are only as strong as we are united, as weak as we are divided'. A dynamic analysis of the peer support networks in the Harry Potter books." *Open Journal of Applied Sciences*, 3(2): 174-185. doi:10.4236/ojapps.2013.32024

`fict_starwars` *Seven one-mode Star Wars character interactions (Gabasova 2016)*

Description

One-mode network dataset collected by Gabasova (2016) on the interactions between Star Wars characters in each movie from Episode 1 ("The Phantom Menace") to Episode 7 ("The Force Awakens"). The data was constructed by parsing the scripts, as described in https://evelinag.com/blog/2015/12-15-star-wars-social-network/index.html.

Characters are named (eg. R2-D2, Anakin, Chewbacca) and the following node attributes are provided where available: height, mass, hair color, skin color, eye color, birth year, sex, homeworld, and species. The node attribute 'faction' has also been added, denoting the faction (eg. Jedi, Rebel Alliance, etc) that Star Wars characters belong to in each episode (coding completed by Yichen Shen, Tiphaine Aeby, and James Hollway).

Weighted ties represent the number of times characters speak within the same scene of each film, indicated by the wave (1-7).

Change in the composition of the network is tracked by the variable 'active', though several other variables also change (mostly as Anakin becomes *spoiler alert*).

Usage

```
data(fict_starwars)
```

Format

```
#> -- # Star Wars network data ----------------------------------------------------
#> # A longitudinal, labelled, complex, weighted, directed network of 110
#> characters and 563 interaction arcs over 7 waves
#>
#> -- Nodes
#> # A tibble: 110 x 12
#>   label         species homeworld sex   height hair_color eye_color skin_color
#>   <chr>           <chr>   <chr>     <chr>  <int> <chr>      <chr>     <chr>
#> 1 Admiral Ackbar  Mon Ca~ Mon Cala  male    180 none      orange    brown mot~
#> 2 Admiral Statura Human   Garel     male     172 black      brown     light
#> 3 Anakin          Human   Tatooine  male     188 blond      blue      fair
#> 4 Bail Organa     Human   Alderaan  male     191 black      brown     tan
#> # i 106 more rows
#> # i 4 more variables: birth_year <dbl>, mass <dbl>, faction <chr>, active <lgl>
#>
#> -- Changes
#> # A tibble: 184 x 4
#>    time  node var         value
#>   <int> <int> <chr>      <list>
#> 1     2     7 active  TRUE<chr>
#> 2     2    10 active  TRUE<chr>
#> 3     2    11 active FALSE<chr>
#> 4     2    13 active FALSE<chr>
#> # i 180 more rows
#>
#> -- Ties
#> # A tibble: 563 x 4
#>    from    to weight  time
#>   <int> <int>  <int> <int>
#> 1    80    73     11     1
#> 2    80    79     14     1
#> 3    80     3     16     1
#> 4    80   106      3     1
#> # i 559 more rows
#>
```

Details

The network for each episode may be extracted and used separately, eg. `to_time(fict_starwars, 1)` for Episode 1.

References

Gabasova, Evelina. 2016. *Star Wars social network (Version 1.0.1). Zenodo*. doi:10.5281/zenodo.1411479

---

fict_thrones *One-mode Game of Thrones kinship (Glander 2017)*

Description

The original dataset was put together by Erin Pierce and Ben Kahle for an assignment for a course on Bayesian statistics. The data included information on when characters died in the Song of Ice and Fire books, and some predictive factors such as whether they were nobles, married, etc. Shirin Glander extended this data set on character deaths in the TV series Game of Thrones with the kinship relationships between the characters, by scraping "A Wiki of Ice and Fire" and adding missing information by hand. The network holds two layers of kinship among 208 characters. A "parent" tie runs from the parent to the child, and a "spouse" tie is recorded in both directions. There is certainly more that can be done here.

Usage

```
data(fict_thrones)
```

Format

```
#> -- # Game of Thrones Kinship ----------------------------------------------------
#> # A labelled, multiplex, directed network of 208 characters and 156 spouse arcs
#> and 248 parent arcs
#>
#> -- Nodes
#> # A tibble: 208 x 10
#>   label        culture house  popularity Gender title birth death noble married
#>   <chr>         <fct>   <chr>      <dbl> <chr>  <chr> <int> <int> <lgl> <lgl>
#> 1 Alys Arryn    <NA>    House~     0.0803 female ""       NA    NA FALSE TRUE
#> 2 Elys Waynwood <NA>    House~     0.0702 female "Ser"    NA    NA TRUE  TRUE
#> 3 Jasper Arryn  <NA>    House~     0.0435 male   "Eyr~    NA    NA TRUE  FALSE
#> 4 Jeyne Royce   <NA>    House~     0      female  <NA>    NA    NA NA    NA
#> 5 Jon Arryn     Valemen House~     0.836  male   "Eyr~   217   298 TRUE  TRUE
#> 6 Lysa Arryn    <NA>    House~     0      female "Lad~   266   300 TRUE  TRUE
#> # i 202 more rows
#>
#> -- Ties
#> # A tibble: 404 x 3
#>    from    to layer
```

```
#>   <int> <int> <chr>
#> 1     1     2 spouse
#> 2     2     1 spouse
#> 3     3     1 parent
#> 4     3     5 parent
#> 5     4     5 spouse
#> 6     5     4 spouse
#> # i 398 more rows
#>
```

References

Pierce, Erin, and Ben Kahle. 2015. "Bayesian Survival Analysis in A Song of Ice and Fire".

Glander, Shirin. 2017. "Network analysis of Game of Thrones".

`glossary` *Adding network glossary items*

Description

This function adds a glossary item, useful in tutorials. Where no glossary entry exists for the term, the text is simply returned italicised.

Usage

```
gloss(text, ref = NULL)

print_glossary()

clear_glossary()
```

Arguments

`text`	The text to appear.
`ref`	The name of the glossary item to index. If NULL, then the function will search the glossary for 'text' instead.

`interface` *Console command line interface*

Description

These functions wrap `{cli}` functions and elements to build an attractive command line interface (CLI). They divide into those that change what a function does and those that only report what it did.

A call that changes control flow always fires, whatever the verbosity, because silencing it would let the code it guards run on and return a wrong answer instead of an explanation:

- `snet_abort()` for an error the user has to fix.
- `snet_unavailable()` for a feature that is not yet available.
- `snet_warn()` for a result the user should not trust without reading, such as a value that is dropped or a name that does not match.

A call that only reports is silenced under the default verbosity:

- `snet_info()` for what a function chose on the user's behalf.
- `snet_success()` for the completion of a long task.
- `snet_progress_step()` and the other `snet_progress_*()` functions for a progress bar.
- `snet_minor_info()` for the bookkeeping a function did on the way, such as which column it renamed.

`snet_prompt()` asks the user something, so it always shows.

Usage

```
snet_info(..., .envir = parent.frame())

snet_minor_info(..., .envir = parent.frame())

snet_warn(..., .envir = parent.frame())

snet_abort(..., .envir = parent.frame())

snet_success(..., .envir = parent.frame())

snet_prompt(..., .envir = parent.frame())

snet_unavailable(..., .envir = parent.frame())
```

Arguments

`...`	One or more character strings. For most of these functions, if multiple strings are passed these will be pasted together.
`.envir`	This argument is just to inherit the parent frame in the (likely) event that the function is used within another function.

Verbosity

The `snet_verbosity` option takes four levels, each adding one thing to the one before it:

- `'quiet'`, the default, reports nothing that is not an error, a warning, or a prompt.
- `'normal'` adds `snet_info()` and `snet_success()`, so that a user learns what a function chose on their behalf.
- `'verbose'` adds the progress bars, which reassure a user that a long task is running.
- `'debug'` adds `snet_minor_info()`, which reports exhaustively what a function did on the way.

Set one with, for example, `options(snet_verbosity = 'verbose')`.

---

irps_911 — *One-mode multiplex network of relationships between 9/11 hijackers (Krebs 2002)*

Description

This network records two different types of relationships between and surrounding the hijackers of four planes in the United States on September 11, 2001, culminating in those planes crashing into four locations: New York's World Trade Center (North and South buildings), as well as the Pentagon and a location in Somerset County, Pennsylvania.

The hijackers were members of al-Qaeda. Valdis Krebs collected further information on the broader network of associates of these hijackers from what major newspapers, such as the New York Times, the Wall Street Journal, the Washington Post, and the Los Angeles Times, reported in the weeks after the attacks, reflecting on the challenges of collecting this information even after the fact. He began with the 19 hijackers, and added an associate or a relationship as it was reported.

The data includes two types of ties: "trust"ed prior contacts among the hijackers, and "association" ties among the hijackers but also their broader associates. All associates are named, along with a logical vector about whether they were a hijacker or not, and if so which their (eventual) target was.

Usage

```
data(irps_911)
```

Format

```
#> -- # 9/11 hijackers and their associates ----------------------------------------
#> # A labelled, complex, multiplex, undirected network of 60 terrorists and 126
#> association ties and 27 trust ties
#>
#> -- Nodes
#> # A tibble: 60 x 3
#>   label             hijacker target
#>   <chr>             <lgl>    <chr>
#> 1 Majed Moqed       TRUE     Pentagon
```

```
#> 2 Khalid Al Mihdhar TRUE     Pentagon
#> 3 Hani Hanjour      TRUE     Pentagon
#> 4 Nawaf Alhazmi     TRUE     Pentagon
#> 5 Salem Alhazmi     TRUE     Pentagon
#> 6 Ahmed Alnami      TRUE     Pennsylvania
#> # i 54 more rows
#>
#> -- Ties
#> # A tibble: 153 x 3
#>    from    to layer
#>   <int> <int> <chr>
#> 1     1     3 association
#> 2     1     3 trust
#> 3     2     3 association
#> 4     2     3 trust
#> 5     2     4 association
#> 6     2     4 trust
#> # i 147 more rows
#>
```

References

Krebs, Valdis. 2002. "Mapping networks of terrorist cells". *Connections* 24(3): 43-52.

---

irps_blogs — *One-mode directed network of links between US political blogs (Adamic and Glance 2005)*

Description

This network consists of the blogosphere around the time of the 2004 US presidential election until February 2005. The 2004 election was the first in which blogging played a significant role. Ties were constructed from a crawl of the front page of each blog.

Political leaning is indicated as "Liberal" (or left leaning) or "Conservative" (or right leaning), sourced from blog directories. Some blogs were labelled manually, based on incoming and outgoing links and posts.

Usage

```
data(irps_blogs)
```

Format

```
#> -- # US political blogosphere circa 2004 ----------------------------------------
#> # A labelled, complex, directed network of 1490 blogs and 19090 link arcs (130
#> parallel)
#>
```

```
#> -- Nodes
#> # A tibble: 1,490 x 3
#>   label                      Leaning Source
#>   <chr>                      <chr>   <chr>
#> 1 100monkeystyping.com       Liberal Blogarama
#> 2 12thharmonic.com/wordpress Liberal BlogCatalog
#> 3 40ozblog.blogspot.com      Liberal Blogarama,BlogCatalog
#> 4 4lina.tblog.com            Liberal Blogarama
#> 5 750volts.blogspot.com      Liberal Blogarama
#> 6 95theses.blogspot.com      Liberal Blogarama
#> # i 1,484 more rows
#>
#> -- Ties
#> # A tibble: 19,090 x 2
#>    from    to
#>   <int> <int>
#> 1   267  1394
#> 2   267   483
#> 3   267  1051
#> 4   904  1479
#> 5   904   919
#> 6   904  1045
#> # i 19,084 more rows
#>
```

Details

Adamic and Glance gathered the blog URLs from the eTalkingHead, BlogCatalog, CampaignLine, and Blogarama directories, which the 'Source' nodal attribute records. They retrieved a single front page for each blog on 8 February 2005, added the blogs those pages cited 17 or more times, and retrieved a front page for each of those on 22 February 2005. Libertarian, independent, and moderate blogs were not gathered.

A tie is any link from one blog's front page to another's, whether it appeared in a post or in the blogroll in the sidebar, and the two are not distinguished.

Note that 65 of the 19090 arcs repeat an arc already in the network, with nothing recorded to tell the two apart, so `tie_is_parallel()` marks 130 ties and `as_matrix()` reports two rather than one in those cells.

References

Adamic, Lada, and Natalie Glance. 2005. "The political blogosphere and the 2004 US Election: Divided they blog". *LinkKDD '05: Proceedings of the 3rd international workshop on Link discovery*, 36-43. doi:10.1145/1134271.1134277

irps_books *One-mode undirected network of co-purchased books about US politics on Amazon*

Description

This network consists of books about US politics sold by Amazon.com. Ties represent books that are often purchased together, as revealed by Amazon's 'customers who bought this book also bought these other books' section on those books' pages on the website.

Information about the book's leaning "Liberal", "Neutral", or "Conservative" were added separately by Mark Newman based on the abstracts, descriptions, and reviews posted on Amazon.

These data should be cited as V. Krebs, unpublished, http://www.orgnet.com/.

Usage

```
data(irps_books)
```

Format

```
#> -- # Co-purchased US political books -------------------------------------------
#> # A labelled, undirected network of 105 books and 441 co-purchasing ties
#>
#> -- Nodes
#> # A tibble: 105 x 2
#>   label                      Leaning
#>   <chr>                      <chr>
#> 1 1000 Years for Revenge     Neutral
#> 2 Bush vs. the Beltway       Conservative
#> 3 Charlie Wilson's War       Conservative
#> 4 Losing Bin Laden           Conservative
#> 5 Sleeping With the Devil    Neutral
#> 6 The Man Who Warned America Conservative
#> # i 99 more rows
#>
#> -- Ties
#> # A tibble: 441 x 2
#>    from    to
#>   <int> <int>
#> 1     1     2
#> 2     1     3
#> 3     1     4
#> 4     2     4
#> 5     1     5
#> 6     3     5
#> # i 435 more rows
#>
```

Author(s)

Valdis Krebs, Mark Newman

`irps_corruption` *One-mode multiplex network of the Czech Rath corruption affair (Diviak et al. 2019)*

Description

This network reconstructs a Czech political corruption case, known as the Rath affair, from publicly available archival data. David Rath was a Social Democrat politician who served as Minister of Health and then as governor of the Central Bohemian region. The police arrested him on 14 May 2012 as he carried a wine box that held seven million Czech crowns in cash. He and his associates manipulated public tenders in the region, several of them funded by the European Union, and took bribes from the firms that won them. The courts convicted Rath and most of his co-defendants, and he began a seven year prison sentence in 2019.

The network contains 11 actors connected through three binary, undirected layers. The "collaboration" layer records communication and tasks that two actors carried out together. The "transfers" layer records bribes and other transfers of resources. The "preexisting" layer records the ties that predate the affair, such as kinship, friendship, and shared political or professional affiliations.

The network also includes two node attributes. `politician` marks the actors who held political office, and `gender` records whether an actor is "male" or "female".

Diviak and colleagues use these data to show that the network is small, sparse, and centralised on Rath and two other actors, and that the preexisting ties underpin the criminal ones. This is a trade-off that covert networks face: ties that are already there are safe to use, but they also concentrate the network on a few actors and make it easier to disrupt.

Usage

```
data(irps_corruption)
```

Format

```
#> -- # Czech Rath affair corruption network ---------------------------------------
#> # A labelled, multiplex, undirected network of 11 actors and 24 collaboration
#> ties, 15 transfer ties, and 5 preexisting ties
#>
#> -- Nodes
#> # A tibble: 11 x 3
#>   label            politician gender
#>   <chr>            <lgl>      <chr>
#> 1 David Rath       TRUE       male
#> 2 Petr Kott        TRUE       male
#> 3 Lucia Novanska   FALSE      female
#> 4 Katerina Pancova TRUE       female
#> 5 Pavel Drazdansky FALSE      male
```

```
#> 6 Martin Jires     FALSE      male
#> # i 5 more rows
#>
#> -- Ties
#> # A tibble: 44 x 3
#>    from    to layer
#>   <int> <int> <chr>
#> 1     1     2 collaboration
#> 2     1     4 collaboration
#> 3     1     7 collaboration
#> 4     1     8 collaboration
#> 5     2     3 collaboration
#> 6     2     4 collaboration
#> # i 38 more rows
#>
```

Details

The actor names are those that the court and media records use. Diacritics are removed from them, so "Drazdansky" appears here for "Drážďanský".

Author(s)

Tomáš Diviák

Source

The layer names shorten those that the article uses, so that each is a single word: "transfers" for resource transfer, and "preexisting" for pre-existing ties. The two node attributes are coded 0/1 in the source data. Here `politician` is logical, and `gender` is "male" for 0 and "female" for 1.

References

Diviák, Tomáš, Jan Kornelis Dijkstra, and Tom A. B. Snijders. 2019. "Structure, multiplexity, and centrality in a corruption network: The Czech Rath affair". *Trends in Organized Crime* 22(3): 274-297. doi:10.1007/s121170189334y

---

`irps_nuclear` *Two-mode dynamic discourse network of Germany's nuclear energy phase-out (Haunss and Hollway 2023)*

Description

Following the 11 March 2011 Fukushima nuclear disaster in Japan, there was a vigorous public debate in Germany about the future of nuclear energy. This network captures the discourse established by 337 actors, including individual politicians, experts, parties, and the media, and their claims about nuclear energy and German nuclear energy policy. These claims were with respect to 54 concepts coded, and could be supportive or critical, and could also be repeated.

Usage

```
data(irps_nuclear)
```

Format

```
#> -- # German nuclear discourse network ------------------------------------------
#> # A dynamic, labelled, signed, two-mode network of 337 speakers and 54 concepts
#> and 1164 claim ties (16 parallel) from 2011-03-11 to 2011-06-30
#>
#> -- Nodes
#> # A tibble: 391 x 10
#>   label        mode  active politician govt  coalition office org   party power
#>   <chr>        <chr> <lgl>  <lgl>     <lgl> <lgl>    <lgl>  <chr> <chr> <int>
#> 1 VfEW         spea~ TRUE   FALSE     FALSE NA       FALSE  VfEW  <NA>      0
#> 2 Angela Merkel spea~ TRUE   TRUE      TRUE  TRUE     TRUE   CDU   31        2
#> 3 Sunday Times  spea~ TRUE   FALSE     FALSE NA       FALSE  Sund~ <NA>      0
#> 4 SPD          spea~ TRUE   TRUE      FALSE FALSE    FALSE  SPD   32        1
#> 5 Norbert Rött~ spea~ TRUE   TRUE      TRUE  TRUE     TRUE   CDU   31        1
#> 6 Torsten Krau~ spea~ TRUE   FALSE     FALSE NA       FALSE  Die ~ <NA>      0
#> # i 385 more rows
#>
#> -- Ties
#> # A tibble: 1,164 x 5
#>    from    to time       weight default
#>   <int> <int> <date>      <int> <lgl>
#> 1     1   338 2011-03-11     -1 FALSE
#> 2     2   339 2011-03-12      1 TRUE
#> 3     3   340 2011-03-13     -1 FALSE
#> 4     4   341 2011-03-13      1 TRUE
#> 5     2   342 2011-03-13     -1 TRUE
#> 6     5   343 2011-03-13      1 TRUE
#> # i 1,158 more rows
#>
```

Details

Each tie is one claim by one speaker about one concept on one day, so the network records a stream of events and not a panel. The day is held in a 'time' column, and whether the claim was supportive or critical in a 'weight' column of 1 or -1.

A speaker may claim the same concept on more than one day, and 152 of the speaker-concept pairs do, one of them 15 times. Such claims follow one another rather than coexist, so they are repetitions and not parallel ties. Eight speaker-concept pairs make more than one claim on a single day, which `tie_is_parallel()` marks, covering 16 of the 1164 ties.

References

Haunss Sebastian, James Hollway. 2023. "Multimodal mechanisms of political discourse dynamics and the case of Germany's nuclear energy phase-out". *Network Science*, 11(2):205-223.

doi:10.1017/nws.2022.31

---

irps_revere *Two-mode network of Paul Revere's (Fischer 1995)*

Description

This network is of Paul Revere and 253 of his contemporary's overlapping memberships in seven colonial organisations. The data has been collected by Kieran Healy from the appendix to David Hackett Fischer's "Paul Revere's Ride". It highlights Paul Revere's centrality in this network, and thus his ability to mobilise the towns he rode through on horseback north from Boston on the night of April 18, 1775. This is in contrast to William Dawes, who set out the same night, but south. Despite both men coming from similar class and backgrounds, and riding through towns with similar demography and political leanings, only Paul Revere was able to mobilise those he encountered, and his social network was thought key to this.

Usage

```
data(irps_revere)
```

Format

```
#> # A labelled, two-mode network of 254 nodes and 7 nodes and 319 ties
#>
#> -- Nodes
#> # A tibble: 261 x 2
#>   type  name
#>   <lgl> <chr>
#> 1 FALSE Adams.John
#> 2 FALSE Adams.Samuel
#> 3 FALSE Allen.Dr
#> 4 FALSE Appleton.Nathaniel
#> 5 FALSE Ash.Gilbert
#> 6 FALSE Austin.Benjamin
#> # i 255 more rows
#>
#> -- Ties
#> # A tibble: 319 x 2
#>    from    to
#>   <int> <int>
#> 1     1   257
#> 2     1   258
#> 3     2   257
#> 4     2   258
#> 5     2   260
#> 6     2   261
#> # i 313 more rows
#>
```

References

Fischer, David Hackett. 1995. "Paul Revere's Ride". Oxford: Oxford University Press.

Han, Shin-Kap. 2009. "The Other Ride of Paul Revere: The Brokerage Role in the Making of the American Revolution". *Mobilization: An International Quarterly*, 14(2): 143-162. doi:10.17813/maiq.14.2.g360870167085210

Healy, Kieran. 2013. "Using Metadata to find Paul Revere".

---

irps_supremecourt *Two-mode network of votes on the Rehnquist court (Spaeth 1990)*

Description

A two-mode network of 376 US Supreme Court cases and the nine justices who sat together on the Rehnquist court between the 1995 and 2004 terms. This was the longest period in the court's history without a change in its membership, which makes the whole run comparable.

A tie indicates that a justice voted with the majority on a case, and is weighted `1` for a full vote with the majority and `0.5` for a partial concurrence.

One nodal attribute is included:

- *term*: for cases, the court term in which the case was decided, from 1995 to 2004. This is `NA` for the justices.

Usage

```
data(irps_supremecourt)
```

Format

```
#> -- # Rehnquist Court ------------------------------------------------------------
#> # A labelled, weighted, two-mode network of 376 cases and 9 justices and 2300
#> majority vote ties
#>
#> -- Nodes
#> # A tibble: 385 x 3
#>   label      mode   term
#>   <chr>      <chr> <int>
#> 1 E001_Ada95 cases  1995
#> 2 E002_All95 cases  1995
#> 3 E003_Ari95 cases  1995
#> 4 E004_Asg95 cases  1995
#> 5 E005_Bab95 cases  1995
#> 6 E006_Cal95 cases  1995
#> # i 379 more rows
#>
#> -- Ties
#> # A tibble: 2,300 x 3
```

```
#>    from    to weight
#>   <int> <int>  <dbl>
#> 1     1   377      1
#> 2     2   377      1
#> 3     3   377      1
#> 4     4   377      1
#> 5     6   377      1
#> 6     7   377      1
#> # i 2,294 more rows
#>
```

Details

Cases are labelled by an index and an abbreviation of the case name followed by the two-digit term, so that `E001_Ada95` is the first case of the 1995 term. Justice names have been corrected to their standard spelling.

Source

The UCINET standard dataset collection derives it from the Supreme Court Database.

References

Spaeth, Harold J. 1990. *United States Supreme Court Judicial Database*. Ann Arbor MI: Inter-university Consortium for Political and Social Research. doi:10.3886/icpsr09422

Doreian, Patrick, Paulette Lloyd, and Andrej Mrvar. 2013. "Partitioning large signed two-mode networks: Problems and prospects". *Social Networks* 35(2): 212-230. doi:10.1016/j.socnet.2012.01.002

`irps_tribes` *One-mode signed network of Gahuku-Gama sub-tribes (Read 1954)*

Description

Political relations among 16 Gahuku-Gama sub-tribes of the central highlands of New Guinea, as reported in Read's (1954) ethnography. This is one of the canonical datasets for the study of structural balance and of networks containing negative ties, since the sub-tribes stand in one of two mutually exclusive relations:

- `weight` > `0` records *rova*, a relation of political alliance
- `weight` < `0` records *hina*, a relation of political opposition

Each relation holds between 29 pairs of sub-tribes. No pair of sub-tribes stands in both relations, and pairs with no recorded political relation are simply absent.

Usage

```
data(irps_tribes)
```

Format

```
#> -- # Gahuku-Gama Sub-tribes ----------------------------------------------------
#> # A labelled, signed, undirected network of 16 sub-tribes and 58 political ties
#>
#> -- Nodes
#> # A tibble: 16 x 1
#>   label
#>   <chr>
#> 1 Gavev
#> 2 Kotun
#> 3 Ove
#> 4 Alika
#> 5 Nagam
#> 6 Gahuk
#> # i 10 more rows
#>
#> -- Ties
#> # A tibble: 58 x 3
#>    from    to weight
#>   <int> <int>  <dbl>
#> 1     1     2      1
#> 2     3     4      1
#> 3     3     6      1
#> 4     3     7      1
#> 5     5     7      1
#> 6     6     7      1
#> # i 52 more rows
#>
```

Details

The network is held as a 'stocnet' object, so that the metadata reported in the GRAND statement can be recorded in its info component, including where and when the relations were observed and by what method. As is the convention for 'stocnet' objects, the sign of each relation is held as a negative or positive weight rather than in a separate 'sign' column.

Source

Sub-tribe names have been title-cased. The date recorded is that of Read's publication; his fieldwork in the Asaro valley preceded it by some years.

References

Read, Kenneth E. 1954. "Cultures of the Central Highlands, New Guinea". *Southwestern Journal of Anthropology* 10(1): 1-43. doi:10.1086/soutjanth.10.1.3629074

Everett, Martin G., and Stephen P. Borgatti. 2014. "Networks containing negative ties". *Social Networks* 38: 111-120. doi:10.1016/j.socnet.2014.03.005

irps_usgeo *One-mode undirected network of US state contiguity (Meghanathan 2017)*

Description

This network is of contiguity between US states. States that share a border are connected by a tie in the network. The data is a network of 107 ties among 50 US states (nodes). States are named by their two-letter ISO-3166 code. This data includes also the names of the capitol cities of each state, which are listed in the node attribute 'capitol'.

Usage

```
data(irps_usgeo)
```

Format

```
#> -- # US State Contiguity -------------------------------------------------------
#> # A labelled, undirected network of 50 states and 107 contiguity ties
#>
#> -- Nodes
#> # A tibble: 50 x 3
#>   name  capitol     population
#>   <chr> <chr>            <int>
#> 1 AK    Juneau              NA
#> 2 AL    Montgomery     4780127
#> 3 AR    Little Rock    2915958
#> 4 AZ    Phoenix        6392307
#> 5 CA    Sacramento    37252895
#> 6 CO    Denver         5029324
#> # i 44 more rows
#>
#> -- Ties
#> # A tibble: 107 x 2
#>    from    to
#>   <int> <int>
#> 1     2     9
#> 2     2    10
#> 3     2    25
#> 4     2    42
#> 5     3    18
#> 6     3    24
#> # i 101 more rows
#>
```

References

Meghanathan, Natarajan. 2017. "Complex network analysis of the contiguous United States graph." *Computer and Information Science*, 10(1): 54-76. doi:10.5539/cis.v10n1p54

`irps_wwi` *One-mode signed network of relationships between European major powers (Antal et al. 2006)*

Description

This network records the evolution of the major relationship changes between the protagonists of World War I (WWI) from 1872 to 1907. Each tie carries the year in which a relationship began and the year in which it ended, and the relationships that still held in 1907 are run on to the end of the war in 1918. A friendly relationship has a weight of 1 and an unfriendly one of -1. It is incomplete both in terms of (eventual) parties to the war as well as some other relations, but gives a good overview of the main alliances and enmities.

The data series begins with the Three Emperors' League (1872, revived in 1881) between Germany, Austria-Hungary, and Russia. The Triple Alliance in 1882 joined Germany, Austria-Hungary, and Italy into a bloc that lasted until WWI. A bilateral alliance between Germany and Russia lapsed in 1890, and a French-Russian alliance developed between 1891-1894. The Entente Cordiale thawed and then fostered relations between Great Britain and France in 1904, and a British-Russian agreement in 1907 bound Great Britain, France, and Russia into the Triple Entente.

Usage

```
data(irps_wwi)
```

Format

```
#> -- # World War I Protagonists ---------------------------------------------------
#> # A dynamic, labelled, signed, undirected network of 6 European major powers
#> and 20 relationship ties from 1872 to 1918
#>
#> -- Nodes
#> # A tibble: 6 x 1
#>   label
#>   <chr>
#> 1 GBR
#> 2 FRA
#> 3 RUS
#> 4 AUH
#> 5 DEU
#> 6 ITA
#>
#> -- Ties
#> # A tibble: 20 x 5
```

```
#>    from    to weight begin   end
#>   <int> <int>  <dbl> <dbl> <dbl>
#> 1     1     2     -1  1872  1904
#> 2     1     3     -1  1872  1907
#> 3     1     4     -1  1872  1918
#> 4     2     3     -1  1872  1890
#> 5     2     4     -1  1872  1918
#> 6     2     5     -1  1872  1918
#> # i 14 more rows
#>
```

References

Antal, Tibor, Pavel Krapivsky, and Sidney Redner. 2006. "Social balance on networks: The dynamics of friendship and enmity". *Physica D* 224: 130-136. doi:10.1016/j.physd.2006.09.028

`ison_adolescents` *One-mode subset of the adolescent society network (Coleman 1961)*

Description

One-mode subset of Coleman's adolescent society network (Coleman 1961), as used in Feld's (1991) "Why your friends have more friends than you do". Coleman collected data on friendships among students in 12 U.S. high schools. Feld explored a subset of 8 girls from one of these schools, "Marketville", and gave them fictitious names, which are retained here.

Usage

```
data(ison_adolescents)
```

Format

```
#> -- # The Adolescent Society ----------------------------------------------------
#> # A labelled, undirected network of 8 adolescents and 10 friendship ties
#>
#> -- Nodes
#> # A tibble: 8 x 1
#>   name
#>   <chr>
#> 1 Betty
#> 2 Sue
#> 3 Alice
#> 4 Jane
#> 5 Dale
#> 6 Pam
#> # i 2 more rows
#>
```

```
#> -- Ties
#> # A tibble: 10 x 2
#>    from    to
#>   <int> <int>
#> 1     1     2
#> 2     2     3
#> 3     3     4
#> 4     2     5
#> 5     3     5
#> 6     4     5
#> # i 4 more rows
#>
```

References

Coleman, James S. 1961. *The Adolescent Society*. New York: Free Press.

Feld, Scott. 1991. "Why your friends have more friends than you do" *American Journal of Sociology* 96(6): 1464-1477. doi:10.1086/229693.

`ison_algebra` *Multiplex graph object of friends, social, and task ties (McFarland 2001)*

Description

Multiplex graph object of friends, social, and task ties between 16 anonymous students in an honors algebra class (M182). Each type of tie is weighted: the `friends` ties are weighted `2` = best friends, `1` = friend, and `0` is not a friend; `social` consists of social interactions per hour; and `tasks` consists of task interactions per hour.

Usage

```
data(ison_algebra)
```

Format

```
#> -- # M182 Algebra Class -------------------------------------------------------
#> # A multiplex, weighted, directed network of 16 nodes and 129 social arcs, 88
#> task arcs, and 62 friendship arcs
#>
#> -- Ties
#> # A tibble: 279 x 4
#>    from    to type   weight
#>   <int> <int> <chr>   <dbl>
#> 1     1     5 social   1.2
#> 2     1     5 tasks    0.3
#> 3     1     8 social   0.15
```

```
#> 4     1     9 social   2.85
#> 5     1     9 tasks    0.3
#> 6     1    10 social   6.45
#> # i 273 more rows
#>
```

Source

See also `data(studentnets.M182, package = "NetData")`

Larger comprehensive data set publicly available, contact Daniel A. McFarland for details.

References

McFarland, Daniel A. (2001) "Student Resistance." *American Journal of Sociology* 107(3): 612-78. doi:10.1086/338779.

`ison_bankwiring` *One-mode multiplex, signed network of the bank wiring room (Roethlisberger and Dickson 1939)*

Description

Observational data on 14 Western Electric employees in the bank wiring room of the Hawthorne plant, first reported by Roethlisberger and Dickson (1939) and better known through Homans' (1950) scrutiny of the interactions and the blockmodel analysis in Breiger, Boorman and Arabie (1975). The men worked in a single room, and their interaction was recorded by an observer stationed there over several months.

Six layers of tie were recorded, four positive and two negative. As is the convention for 'stocnet' objects, the sign of a tie is the sign of its `weight`:

- *friendship* (`weight` > 0): positive ties of friendship
- *games* (`weight` > 0): participation in horseplay
- *help* (`weight` > 0): helping another with their work
- *trades* (`weight` > 0): trading job assignments, weighted by the number of times observed
- *antagonism* (`weight` < 0): antagonistic behaviour
- *conflict* (`weight` < 0): participation in arguments about open windows

One nodal attribute is included:

- *role*: one of `inspector` (I1, I3), `wireman` (W1 to W9), or `solderer` (S1, S2, S4)

Usage

```
data(ison_bankwiring)
```

Format

```
#> -- # Bank Wiring Room ----------------------------------------------------------
#> # A labelled, multiplex, signed, weighted, directed network of 14 employees and
#> 13 friendship ties, 28 game ties, 24 help arcs, 7 trade arcs, 19 antagonism
#> ties, and 19 conflict ties
#>
#> -- Nodes
#> # A tibble: 14 x 2
#>   label role
#>   <chr> <chr>
#> 1 I1    inspector
#> 2 I3    inspector
#> 3 W1    wireman
#> 4 W2    wireman
#> 5 W3    wireman
#> 6 W4    wireman
#> # i 8 more rows
#>
#> -- Ties
#> # A tibble: 110 x 4
#>    from    to layer      weight
#>   <int> <int> <chr>       <dbl>
#> 1     1     5 friendship      1
#> 2     3     5 friendship      1
#> 3     3     6 friendship      1
#> 4     5     6 friendship      1
#> 5     9    10 friendship      1
#> 6     9    11 friendship      1
#> # i 104 more rows
#>
```

Details

The network is directed as a whole because the `help` and `trades` relations are asymmetric. The four symmetric relations are held once per dyad, and the info component records the directedness of each layer. Coercion to another class reciprocates those layers again, because an 'igraph' or 'network' object is directed or undirected as a whole. `to_uniplex()` returns an undirected network for a symmetric layer.

All ties are weighted `1` or `-1` except `trades`, which carries the observed count.

Source

The UCINET standard dataset collection, as distributed in `{xUCINET}`. The DOI given for Roethlisberger and Dickson resolves to the 2004 Routledge reissue; the 1939 original predates DOI registration.

References

Roethlisberger, Fritz J., and William J. Dickson. 1939. *Management and the Worker*. Cambridge MA: Harvard University Press. doi:10.4324/9780203503010

Homans, George C. 1950. *The Human Group*. New York: Harcourt-Brace.

Breiger, Ronald L., Scott A. Boorman, and Phipps Arabie. 1975. "An algorithm for clustering relational data with applications to social network analysis and comparison with multidimensional scaling". *Journal of Mathematical Psychology* 12(3): 328-383. doi:10.1016/00222496(75)900280

`ison_brandes` *One-mode and two-mode centrality demonstration networks*

Description

This network should solely be used for demonstration purposes as it does not describe a real network. To convert into the two-mode version, assign `ison_brandes |> rename(type = twomode_type)`.

Usage

```
data(ison_brandes)
```

Format

```
#> # A undirected network of 11 nodes and 12 ties
#>
#> -- Nodes
#> # A tibble: 11 x 1
#>   twomode_type
#>   <lgl>
#> 1 FALSE
#> 2 FALSE
#> 3 TRUE
#> 4 FALSE
#> 5 TRUE
#> 6 TRUE
#> # i 5 more rows
#>
#> -- Ties
#> # A tibble: 12 x 2
#>    from    to
#>   <int> <int>
#> 1     1     3
#> 2     2     3
#> 3     3     4
#> 4     4     5
#> 5     4     6
#> 6     5     7
```

```
#> # i 6 more rows
#>
```

`ison_classmates` *One-mode longitudinal network of Dutch classmates (Knecht 2008)*

Description

Friendship among 26 pupils aged 11 to 13 in a Dutch school class, measured at four time points over one school year, collected by Andrea Knecht for her dissertation. The data are widely used to illustrate the joint dynamics of network selection and behavioural influence.

Two layers of tie are recorded:

- *friends*: friendship nominations, at each of the four waves
- *primary*: whether the pupils knew each other from primary school. This is a time-invariant baseline relation and recorded at wave 1 only.

Four fixed nodal attributes are included:

- *sex*: `female` or `male`
- *age*: age at wave 1, in years
- *ethnicity*: `Dutch` or `non-Dutch`
- *religion*: religion of the father, one of `Christian`, `none`, `other`

Two further attributes change over the waves, and are held as nodal changes for waves 2 to 4:

- *delinquency*: self-reported delinquent behaviour, measured at all four waves
- *alcohol*: self-reported alcohol use, not measured at wave 1 and so `NA` there

Usage

```
data(ison_classmates)
```

Format

```
#> -- # Knecht's Classmates -------------------------------------------------------
#> # A longitudinal, labelled, multiplex, directed network of 26 pupils and 460
#> friendship arcs and 86 primary arcs over 4 waves
#>
#> -- Nodes
#> # A tibble: 26 x 7
#>   label sex      age ethnicity religion delinquency alcohol
#>   <chr> <chr>  <int> <chr>     <chr>          <int>   <dbl>
#> 1 a01   female    12 Dutch     none               2      NA
#> 2 a02   female    12 Dutch     none               1      NA
#> 3 a03   female    12 non-Dutch other              2      NA
#> 4 a04   male      12 Dutch     none               2      NA
```

```
#> # i 22 more rows
#>
#> -- Changes
#> # A tibble: 150 x 4
#>    time  node var             value
#>   <int> <int> <chr>          <list>
#> 1     2     1 alcohol        1<int>
#> 2     2     1 delinquency    2<int>
#> 3     2     2 na          TRUE<lgl>
#> 4     2     3 alcohol        3<int>
#> # i 146 more rows
#>
#> -- Ties
#> # A tibble: 546 x 4
#>    from    to layer    time
#>   <int> <int> <chr>   <int>
#> 1     3     1 friends     1
#> 2     9     1 friends     1
#> 3    12     1 friends     1
#> 4     3     2 friends     1
#> # i 542 more rows
#>
```

Details

Missing attribute values, coded `-99` elsewhere, are recorded as `NA` here.

The source data also distinguish nominations that are missing (`9`) from those that are structurally missing (`10`), where the pupil was not in the class at that wave. The two are recorded differently here.

A missing nomination says that a pupil did not answer. One pupil did not answer at wave 2 and two did not at wave 3, which is logged as a change of their `na` status, and a change back at the wave they answer again. Every nomination they would have given is then missing: 25 at wave 2 and 48 at wave 3, which `as_missinglist()` returns. The nominations they received are not missing, since the other pupils still answered. Recording them this way, rather than as the absence of a tie, keeps a nomination nobody gave from being read as one somebody withheld. `impute_ties()` treats them as absent ties, with `rule = "zero"`, or imputes them, with one of its other rules.

A structurally missing nomination says only that one pupil was not in the class, from wave 3 onwards. That pupil leaving is logged as a change of their `active` status at wave 3, which `as_siena()` renders as a composition change, and no ties are recorded for them at either wave. Nothing they might have been nominated for is missing at those waves either, since there was nothing there to miss. This is why the two pupils who did not answer at wave 3 have 24 missing nominations each and not 25.

Nobody was asked to nominate themselves, so the diagonal of each matrix is dropped. This drops a self-tie in each of the wave 2 friendship and primary-school matrices, and a self-nomination coded as missing in the wave 2 and wave 3 friendship matrices.

References

Knecht, Andrea. 2008. *Friendship Selection and Friends' Influence. Dynamics of Networks and Actor Attributes in Early Adolescence*. PhD dissertation, University of Utrecht.

Knecht, Andrea, Tom A. B. Snijders, Chris Baerveldt, Christian E. G. Steglich, and Werner Raub. 2010. "Friendship and Delinquency: Selection and Influence Processes in Early Adolescence". *Social Development* 19(3): 494-514. doi:10.1111/j.14679507.2009.00564.x

---

`ison_dolphins` *One-mode, undirected network of frequent associations in a dolphin pod (Lusseau et al. 2003)*

Description

These data contain the frequent associations between the 62 dolphins of a community of bottlenose dolphins living in Doubtful Sound, New Zealand. The network was constructed from systematic boat surveys of the fjord over seven years, from November 1994 to November 2001. Each time a school of dolphins was encountered, its adult members were photographed and identified by the natural markings on their dorsal fins, which gave the composition of 1,292 schools. How closely two dolphins associated was measured by a half-weight index, and compared to the index expected when the members of the schools are permuted at random. A tie joins a pair of 'preferred companions': two dolphins that were seen in the same school more often than expected by chance. Lusseau (2003) reports 64 adult individuals; the 62 here are those of the network that Lusseau and Newman (2004) share, with the same 159 ties. Additional information can be found in the literature cited below.

Usage

```
data(ison_dolphins)
```

Format

```
#> -- # Doubtful Sound dolphins ---------------------------------------------------
#> # A labelled, undirected network of 62 dolphins and 159 frequent association
#> ties
#>
#> -- Nodes
#> # A tibble: 62 x 1
#>   label
#>   <chr>
#> 1 Beak
#> 2 Beescratch
#> 3 Bumper
#> 4 CCL
#> 5 Cross
#> 6 DN16
#> # i 56 more rows
#>
```

```
#> -- Ties
#> # A tibble: 159 x 2
#>    from    to
#>   <int> <int>
#> 1     4     9
#> 2     6    10
#> 3     7    10
#> 4     1    11
#> 5     3    11
#> 6     6    14
#> # i 153 more rows
#>
```

References

Lusseau, David, K. Schneider, O. J. Boisseau, P. Haase, E. Slooten, and S. M. Dawson. 2003. "The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations", *Behavioral Ecology and Sociobiology* 54, 396-405.

Lusseau, David. 2003. "The emergent properties of a dolphin social network", *Proc. R. Soc. London B* 270(S): S186-S188. doi:10.1098/rsbl.2003.0057

Lusseau, David, and Mark E. J. Newman. 2004. "Identifying the role that animals play in their social networks". *Proc. R. Soc. London B* 271(S6). doi:10.1098/rsbl.2004.0225 Lusseau, David. 2007. "Evidence for social role in a dolphin social network". *Evolutionary Ecology* 21: 357–366. doi:10.1007/s1068200691050

`ison_emotions` *One-mode, weighted network of emotional transitions (Trampe et al. 2015)*

Description

Emotions are highly interconnected, and one emotion often follows another. This network describes the transitions between 18 different emotions as experienced in everyday life. The data is collected from more than 11,000 participants, who used a francophone smartphone application to complete short questionnaires on the emotions they felt at a given moment. They were asked about nine positive emotions (alertness, amusement, awe, gratitude, hope, joy, love, pride, and satisfaction) and nine negative emotions. While Trampe et al. (2015) created and analysed an undirected network in their paper, the directed network constructed by Will Hipson is shared here.

Usage

```
data(ison_emotions)
```

Format

```
#> -- # Emotional transitions ----------------------------------------------------
#> # A labelled, complex, weighted, directed network of 18 emotions and 315
#> transition arcs
#>
#> -- Nodes
#> # A tibble: 18 x 1
#>   label
#>   <chr>
#> 1 Alertness
#> 2 Amusement
#> 3 Anger
#> 4 Anxiety
#> 5 Awe
#> 6 Disdain
#> # i 12 more rows
#>
#> -- Ties
#> # A tibble: 315 x 3
#>    from    to weight
#>   <int> <int>  <int>
#> 1     1     1   6191
#> 2     1     2     97
#> 3     1     3    214
#> 4     1     4   4784
#> 5     1     5     14
#> 6     1     6     82
#> # i 309 more rows
#>
```

References

Trampe, Debra, Jordi Quoidbach, and Maxime Taquet. 2015. "Emotions in everyday life". *PLOS ONE*. doi:10.1371/journal.pone.0145450

Hipson, Will. 2019. https://www.r-bloggers.com/2019/03/network-analysis-of-emotions/

---

ison_florentine — *One-mode multiplex network of Florentine families (Padgett and Ansell 1993)*

Description

Marriage alliances and business dealings among 16 Renaissance Florentine families, coded by John Padgett from historical documents and analysed by Padgett and Ansell (1993) as an account of how the Medici came to control the city around 1430. Two factions dominated that struggle, one around the Medici and one around the Strozzi.

Two layers of tie are recorded:

- *business*: recorded financial ties such as loans, credits, and joint partnerships
- *marriage*: marriage alliances between the families

Three nodal attributes are included:

- *wealth*: each family's net wealth in 1427, in thousands of lira
- *priorates*: the number of seats on the civic council held between 1282 and 1344
- *totalties*: the number of business or marriage ties the family has in the larger dataset of 116 families used by Breiger and Pattison (1986)

Usage

```
data(ison_florentine)
```

Format

```
#> -- # Florentine Families -------------------------------------------------------
#> # A labelled, multiplex, undirected network of 16 families and 15 business ties
#> and 20 marriage ties
#>
#> -- Nodes
#> # A tibble: 16 x 4
#>   label      wealth priorates totalties
#>   <chr>       <int>     <int>     <int>
#> 1 Acciaiuoli     10        53         2
#> 2 Albizzi        36        65         3
#> 3 Barbadori      55         0        14
#> 4 Bischeri       44        12         9
#> 5 Castellani     20        22        18
#> 6 Ginori         32         0         9
#> # i 10 more rows
#>
#> -- Ties
#> # A tibble: 35 x 3
#>    from    to layer
#>   <int> <int> <chr>
#> 1     3     5 business
#> 2     3     6 business
#> 3     4     7 business
#> 4     4     8 business
#> 5     5     8 business
#> 6     7     8 business
#> # i 29 more rows
#>
```

Details

Both relations are recorded symmetrically here, as in the original coding. As Breiger & Pattison (1986) point out, this is defensible for marriage ties but unfortunate for the financial ties, which

were almost certainly directed. Because both layers are symmetric, each tie is stored once per dyad rather than as a pair of reciprocated arcs. The Pucci family is an isolate in both relations.

References

Padgett, John F., and Christopher K. Ansell. 1993. "Robust Action and the Rise of the Medici, 1400-1434". *American Journal of Sociology* 98(6): 1259-1319. doi:10.1086/230190

Breiger, Ron, and Philippa Pattison. 1986. "Cumulated social roles: The duality of persons and their algebras." *Social Networks*, 8: 215-256. doi:10.1016/03788733(86)900067

`ison_fraternity` *One-mode longitudinal network of fraternity preferences (Newcomb 1961)*

Description

Weekly sociometric preference rankings among 17 men attending the University of Michigan in the autumn of 1956. The men were recruited to live in off-campus fraternity housing rented for them as part of the Michigan Group Study Project, supervised by Theodore Newcomb from 1953 to 1956. All were incoming transfer students with no prior acquaintance of one another, which makes this one of the earliest records of a social structure forming from scratch.

Each week, every man ranked all 16 others from most to least preferred. No ties in rank were allowed; each wave is a complete directed network. Two tie attributes record this:

- *rank*: the raw ranking given, where `1` is the respondent's first preference and `16` their last
- *weight*: the ranking reversed (`17 - rank`), so that larger values indicate stronger preference, as expected by weighted measures

Usage

```
data(ison_fraternity)
```

Format

```
#> -- # Newcomb's Fraternity -------------------------------------------------------
#> # A longitudinal, labelled, weighted, directed network of 17 mens and 4080
#> preference arcs over 15 waves
#>
#> -- Nodes
#> # A tibble: 17 x 1
#>   label
#>   <chr>
#> 1 A01
#> 2 A02
#> 3 A03
#> 4 A04
#> 5 A05
```

```
#> 6 A06
#> # i 11 more rows
#>
#> -- Ties
#> # A tibble: 4,080 x 5
#>    from    to weight  rank  time
#>   <int> <int>  <dbl> <dbl> <int>
#> 1     2     1      9     8     1
#> 2     3     1      4    13     1
#> 3     4     1      4    13     1
#> 4     5     1      3    14     1
#> 5     6     1     10     7     1
#> 6     7     1      2    15     1
#> # i 4,074 more rows
#>
```

Details

The men are anonymised as `A01` to `A17`. Waves are numbered by the week of observation. Data were collected in weeks 1 to 9 and 11 to 16; no data were collected in week 10. The network therefore has 15 waves but a maximum wave number of 16.

References

Newcomb, Theodore M. 1961. *The Acquaintance Process*. New York: Holt, Rinehart and Winston. doi:10.1037/13156000

Nordlie, Peter. 1958. *A longitudinal study of interpersonal attraction in a natural group setting*. Unpublished doctoral dissertation, University of Michigan.

White, Harrison C., Scott A. Boorman, and Ronald L. Breiger. 1976. "Social Structure from Multiple Networks. I. Blockmodels of Roles and Positions". *American Journal of Sociology* 81(4): 730-780. doi:10.1086/226141

`ison_hightech` *One-mode multiplex, directed network of managers of a high-tech company (Krackhardt 1987)*

Description

21 managers of a company of just over 100 employees manufactured high-tech equipment on the west coast of the United States. Three types of ties were collected:

- *advice*: every manager's answers to the question "Who would this person go to for help or advice at work?", asked about each of the 21 managers in turn
- *friends*: every manager's answers to the question "Who would this person consider to be a friend?", again asked about each of the 21 managers
- *reports*: "To whom do you report?" based on company reports

The advice and friendship layers are therefore cognitive social structures: each manager reports on every tie among all 21 managers, and each tie names the manager who reported it in its 'by' column. The reports layer is taken from company records, so its ties name no reporter.

The data is anonymised, but four nodal attributes are included:

- *age*: the manager's age in years
- *tenure*: the manager's length of service
- *level*: the manager's level in the corporate hierarchy, where 3 = CEO, 2 = Vice President, and 1 = manager
- *dept*: one of four departments, B, C, D, E, with the CEO alone in A

Usage

```
data(ison_hightech)
```

Format

```
#> -- # High-tech company managers ------------------------------------------------
#> # A multiplex, directed cognitive social structure of 21 managers and 2735
#> advice arcs, 790 friendship arcs, and 20 report arcs from 21 reporters
#>
#> -- Nodes
#> # A tibble: 21 x 4
#>     age tenure level dept
#>   <dbl>  <dbl> <dbl> <chr>
#> 1    33      9     1 E
#> 2    42     20     2 E
#> 3    40     13     1 C
#> 4    33      8     1 E
#> 5    32      3     1 C
#> 6    59     28     1 B
#> # i 15 more rows
#>
#> -- Ties
#> # A tibble: 3,545 x 4
#>    from    to    by layer
#>   <int> <int> <int> <chr>
#> 1     2     1     1 advice
#> 2     3     1     1 advice
#> 3     4     1     1 advice
#> 4     5     1     1 advice
#> 5     6     1     1 advice
#> 6     7     1     1 advice
#> # i 3,539 more rows
#>
```

Details

Most measures pool every report as a tie of its own, so a tie reported by all 21 managers counts 21 times. Use `to_reporter()` for one manager's view of the network, or `to_aggregated(over = "by")` to combine the views. Each manager's own report on their own ties, `to_aggregated(ison_hightech, over = "by", reporters = "sender")`, gives the network that earlier versions of this dataset held, and that is often distributed as the high-tech managers network.

The friendship reports come from `cssTools::highTechManagers` (Yenigun, Ertan, and Siciliano 2016). That dataset replaces manager 17's report on their own friendships, which it describes as naming every other manager, with the reports of the other managers about manager 17. Here that row is instead taken from the friendship network of the earlier versions, in which manager 17 names 18 of the other 20 managers. The advice reports come from the transcription of the appendix of Krackhardt (1987) by Izabel Aguiar.

Source

Aguiar, Izabel. 2022. "Krackhardt's Cognitive Social Structures". https://github.com/izabelaguiar/krackhardt

Yenigun, Deniz, Gunes Ertan, and Michael Siciliano. 2016. *cssTools: Cognitive Social Structure Tools*. R package version 1.0. doi:10.32614/CRAN.package.cssTools

References

Krackhardt, David. 1987. "Cognitive social structures". *Social Networks* 9(2): 109-134. doi:10.1016/03788733(87)900098

`ison_judo_moves` *One-mode judo moves network (Bastazini 2025)*

Description

Judo is a martial art with a long history and many different techniques. It involves a dynamic 'chess match' of throws, holds, locks, submission techniques, and other maneuvers. The techniques are often combined in sequences to create fluid and effective combinations to score points or achieve victory. As the author of this network describes, "While individual techniques (called waza) are foundational, the real artistry lies in how they are chained together – through renraku-waza (combination techniques) and renzoku-waza (continuous combination techniques)" This network describes the relationships between 33 individual judo moves, as recognised by the Kodokan (the official international governing body of judo), where an arc indicates that one move can be followed by another.

Usage

```
data(ison_judo_moves)
```

Format

```
#> -- # Judo attack combinations ---------------------------------------------------
#> # A labelled, complex, directed network of 33 attacks and 81 sequence arcs
#>
#> -- Nodes
#> # A tibble: 33 x 1
#>   label
#>   <chr>
#> 1 Seoi.nage
#> 2 Seoi.otoshi
#> 3 O.uchi.gari
#> 4 Ko.uchi.gari
#> 5 Ippon.seoi.nage
#> 6 Osoto.gari
#> # i 27 more rows
#>
#> -- Ties
#> # A tibble: 81 x 2
#>    from    to
#>   <int> <int>
#> 1     1     2
#> 2     1     3
#> 3     1     4
#> 4     3     4
#> 5     3     6
#> 6     3     7
#> # i 75 more rows
#>
```

References

Bastazini, Vinicius. 2025. "The Dynamics of the “Gentle Way”: Exploring Judo Attack Combinations as Networks in R", https://geekcologist.wordpress.com/2025/05/27/the-dynamics-of-the-gentle-way-exploring-judo-attack-combinations-as-networks-in-r/

Kashiwazaki, Katsuhiko, and Hidetoshi Nakanishi. 1995. *Attacking Judo: A Guide to Combinations and Counters*. Ippon Books.

Kawaishi, Mikinosuke. 1963. *Standing judo: The combinations and counter-attacks*. Budoworks.

van Haesendonck, F.M. 1968. *Judo: Ecyclopédie par l’Image*. Éditions Erasme: Anvers-Bruxelles.

`ison_karateka` *One-mode karateka network (Zachary 1977)*

Description

The network was observed in a university karate club over three years, from 1970 to 1972, and published in 1977. The network describes association patterns among 34 members and maps out allegiance patterns between members and either Mr. Hi, the instructor, or John A., the club president, after an argument about hiking the price for lessons. The club had near 60 members at the time. The 34 in the network are those who met other members outside the club's classes and meetings, and the 'weight' of a tie is the number of contexts outside the club in which the two members met. The allegiance of each node is listed in the 'allegiance' node attribute, which takes the value 1 if the individual sided with Mr. Hi after the fight and 2 if the individual sided with John A.

Usage

```
data(ison_karateka)
```

Format

```
#> -- # Zachary's karate club ----------------------------------------------------
#> # A labelled, weighted, undirected network of 34 club members and 78
#> association ties
#>
#> -- Nodes
#> # A tibble: 34 x 2
#>   label allegiance
#>   <chr>      <dbl>
#> 1 Mr Hi          1
#> 2 2              1
#> 3 3              1
#> 4 4              1
#> 5 5              1
#> 6 6              1
#> # i 28 more rows
#>
#> -- Ties
#> # A tibble: 78 x 3
#>    from    to weight
#>   <int> <int>  <dbl>
#> 1     1     2      4
#> 2     1     3      5
#> 3     2     3      6
#> 4     1     4      3
#> 5     2     4      3
#> 6     3     4      3
#> # i 72 more rows
#>
```

References

Zachary, Wayne W. 1977. "An Information Flow Model for Conflict and Fission in Small Groups." *Journal of Anthropological Research* 33(4):452–73. doi:10.1086/jar.33.4.3629752.

ison_koenigsberg *One-mode Seven Bridges of Koenigsberg network (Euler 1741)*

Description

The Seven Bridges of Koenigsberg is a notable historical problem in mathematics and laid the foundations of graph theory. The city of Koenigsberg in Prussia (now Kaliningrad, Russia) was set on both sides of the Pregel River, and included two large islands which were connected to each other and the mainland by seven bridges. A weekend diversion for inhabitants was to find a walk through the city that would cross each bridge once and only once. The islands could not be reached by any route other than the bridges, and every bridge must have been crossed completely every time (one could not walk half way onto the bridge and then turn around and later cross the other half from the other side). In 1735, Leonard Euler proved that the problem has no solution.

Two pairs of the seven bridges run parallel, joining the same two landmasses: the Kraemer and Schmiede bridges join the Altstadt to the Kneiphof, and the Gruene and Koettel bridges join the Kneiphof to the Vorstadt. These parallel bridges are the problem and not an artefact of the data, so `tie_is_parallel()` marks four of the seven ties, and `as_matrix()` reports two rather than one in those cells. Each tie is named for the bridge it stands for in a 'name' column.

Usage

```
data(ison_koenigsberg)
```

Format

```
#> -- # Seven Bridges of Koenigsberg ----------------------------------------------
#> # A labelled, undirected network of 4 landmasses and 7 bridge ties (4 parallel)
#>
#> -- Nodes
#> # A tibble: 4 x 3
#>   label      lat   lon
#>   <chr>    <dbl> <dbl>
#> 1 Altstadt  54.7  20.5
#> 2 Kneiphof  54.7  20.5
#> 3 Lomse     54.7  20.5
#> 4 Vorstadt  54.7  20.5
#>
#> -- Ties
#> # A tibble: 7 x 3
#>    from    to name
#>   <int> <int> <chr>
#> 1     1     2 Kraemer Bruecke
#> 2     1     2 Schmiedebruecke
#> 3     1     3 Holzbruecke
#> 4     2     3 Honigbruecke
#> 5     2     4 Gruene Bruecke
#> 6     2     4 Koettelbruecke
```

```
#> # i 1 more row
#>
```

Source

{igraphdata}. Euler presented the solution to the St Petersburg Academy on 26 August 1735; it was published in 1741 and is catalogued as Eneström 53.

References

Euler, Leonard. 1741. "Solutio problematis ad geometriam situs pertinentis." *Commentarii academiae scientiarum Petropolitanae*.

---

ison_laterals *Two-mode projection examples (Hollway 2021)*

Description

These networks are for demonstration purposes and do not describe any real world network. All examples contain named nodes. The networks are gathered together as a list and can be retrieved simply by plucking the desired network.

Usage

```
data(ison_laterals)
```

Format

```
#> $ison_bb
#> # A labelled, two-mode network of 4 nodes and 6 nodes and 12 ties
#>
#> -- Nodes
#> # A tibble: 10 x 2
#>   name  type
#>   <chr> <lgl>
#> 1 A     FALSE
#> 2 B     FALSE
#> 3 C     FALSE
#> 4 D     FALSE
#> 5 U     TRUE
#> 6 V     TRUE
#> # i 4 more rows
#>
#> -- Ties
#> # A tibble: 12 x 2
#>    from    to
#>   <int> <int>
#> 1     1     5
```

```
#> 2     1     6
#> 3     2     5
#> 4     2     7
#> 5     2     8
#> 6     2     9
#> # i 6 more rows
#>
#>
#> $ison_bm
#> # A labelled, two-mode network of 4 nodes and 4 nodes and 9 ties
#>
#> -- Nodes
#> # A tibble: 8 x 2
#>   name  type
#>   <chr> <lgl>
#> 1 A     FALSE
#> 2 B     FALSE
#> 3 C     FALSE
#> 4 D     FALSE
#> 5 U     TRUE
#> 6 V     TRUE
#> # i 2 more rows
#>
#> -- Ties
#> # A tibble: 9 x 2
#>    from    to
#>   <int> <int>
#> 1     1     5
#> 2     1     6
#> 3     2     5
#> 4     2     7
#> 5     2     8
#> 6     3     6
#> # i 3 more rows
#>
#>
#> $ison_mb
#> # A labelled, two-mode network of 4 nodes and 4 nodes and 9 ties
#>
#> -- Nodes
#> # A tibble: 8 x 2
#>   name  type
#>   <chr> <lgl>
#> 1 A     FALSE
#> 2 B     FALSE
#> 3 C     FALSE
#> 4 D     FALSE
#> 5 M     TRUE
```

```
#> 6 X     TRUE
#> # i 2 more rows
#>
#> -- Ties
#> # A tibble: 9 x 2
#>    from    to
#>   <int> <int>
#> 1     1     5
#> 2     2     5
#> 3     2     6
#> 4     2     7
#> 5     3     5
#> 6     3     6
#> # i 3 more rows
#>
#>
#> $ison_mm
#> # A labelled, two-mode network of 4 nodes and 2 nodes and 6 ties
#>
#> -- Nodes
#> # A tibble: 6 x 2
#>   name  type
#>   <chr> <lgl>
#> 1 A     FALSE
#> 2 B     FALSE
#> 3 C     FALSE
#> 4 D     FALSE
#> 5 M     TRUE
#> 6 N     TRUE
#>
#> -- Ties
#> # A tibble: 6 x 2
#>    from    to
#>   <int> <int>
#> 1     1     5
#> 2     2     5
#> 3     2     6
#> 4     3     5
#> 5     3     6
#> 6     4     6
#>
```

`ison_lawfirm` *One-mode lawfirm (Lazega 2001)*

Description

One-mode network dataset collected by Lazega (2001) on the relations between partners in a corporate law firm called SG&R in New England 1988-1991. This particular subset includes the 36 partners among the 71 attorneys of this firm. Nodal attributes include seniority, formal status, office in which they work, gender, lawschool they attended, their age, and how many years they had been at the firm.

Usage

```
data(ison_lawfirm)
```

Format

```
#> -- # Lazega's Lawyers ---------------------------------------------------------
#> # A multiplex, directed network of 71 attorneys and 575 friendship arcs, 892
#> advice arcs, and 1104 cowork arcs
#>
#> -- Nodes
#> # A tibble: 71 x 7
#>   status  gender office   seniority   age practice   school
#>   <chr>   <chr>  <chr>        <dbl> <dbl> <chr>      <chr>
#> 1 partner man    Boston          31    64 litigation Harvard/Yale
#> 2 partner man    Boston          32    62 corporate  Harvard/Yale
#> 3 partner man    Hartford        13    67 litigation Harvard/Yale
#> 4 partner man    Boston          31    59 corporate  Other
#> 5 partner man    Hartford        31    59 litigation UConn
#> 6 partner man    Hartford        29    55 litigation Harvard/Yale
#> # i 65 more rows
#>
#> -- Ties
#> # A tibble: 2,571 x 3
#>    from    to type
#>   <int> <int> <chr>
#> 1     1     2 friends
#> 2     1     2 advice
#> 3     1     4 friends
#> 4     1     8 friends
#> 5     1    17 friends
#> 6     1    17 advice
#> # i 2,565 more rows
#>
```

Details

The larger data from which this subset comes includes also individual performance measurements (hours worked, fees brought in) and attitudes concerning various management policy options (see also {sand}), their strong-coworker network, advice network, friendship network, and indirect control network.

Source

`{networkdata}`

References

Lazega, Emmanuel. 2001. *The Collegial Phenomenon: The Social Mechanisms of Cooperation Among Peers in a Corporate Law Partnership*. Oxford: Oxford University Press.

---

ison_monks — *Multiplex network of three one-mode signed, weighted networks and a three-wave longitudinal network of monks (Sampson 1969)*

Description

The data were collected for an ethnographic study of community structure in a New England monastery. Various sociometric data was collected of the novices attending the minor seminary of 'Cloisterville' preparing to join the monastic order.

- `type = "like"` records whom novices said they liked most at three time points/waves
- `type = "esteem"` records whom novices said they held in esteem (weight > 0) and disesteem (weight < 0)
- `type = "praise"` records whom novices said they praised (weight > 0) and blamed (weight < 0)
- `type = "influence"` records whom novices said were a positive influence (weight > 0) and negative influence (weight < 0)

All networks are weighted. Novices' first choices are weighted 3, the second 2, and third choices 1. Some subjects offered tied ranks for their top four choices. The sign of each tie is held as the sign of its weight, so that the weights run from -3 to 3 and neither the valence nor the rank is lost when the network is coerced to another class.

In addition to node names, a 'groups' variable records the four groups that Sampson observed during his time there:

- The *Loyal* Opposition consists of novices who entered the monastery first and defended existing practices
- The *Young Turks* arrived later during a period of change and questioned practices in the monastery
- The *Interstitial* did not take sides in the debate
- The *Outcasts* were novices that were not accepted in the group

Information about senior monks was not included. While `type = "like"` is observed over three waves, the rest of the data was recorded retrospectively from the end of the study, after the network fragmented. The waves in which the novitiates were expelled (1), voluntarily departed (2 and 3), or remained (4) are given in the nodal attribute "left".

Usage

```
data(ison_monks)
```

Format

```
#> -- # Sampson's Monks ------------------------------------------------------------
#> # A longitudinal, labelled, multiplex, signed, weighted, directed network of 18
#> nodes and 168 like arcs, 112 esteem arcs, 103 influence arcs, and 80 praise
#> arcs over 3 waves
#>
#> -- Nodes
#> # A tibble: 18 x 3
#>   label       groups        left
#>   <chr>       <chr>        <dbl>
#> 1 Romuald     Interstitial     3
#> 2 Bonaventure Loyal            4
#> 3 Ambrose     Loyal            4
#> 4 Berthold    Loyal            4
#> 5 Peter       Loyal            3
#> 6 Louis       Loyal            4
#> # i 12 more rows
#>
#> -- Ties
#> # A tibble: 463 x 5
#>    from    to weight  time layer
#>   <int> <int>  <dbl> <dbl> <chr>
#> 1     1     2      1     2 like
#> 2     1     2      1     3 like
#> 3     1     3      1     3 like
#> 4     1     5      3     1 like
#> 5     1     5      3     2 like
#> 6     1     5      3     3 like
#> # i 457 more rows
#>
```

References

Sampson, Samuel F. 1969. *Crisis in a cloister*. Unpublished doctoral dissertation, Cornell University.

Breiger R., Boorman S. and Arabie P. 1975. "An algorithm for clustering relational data with applications to social network analysis and comparison with multidimensional scaling". *Journal of Mathematical Psychology*, 12: 328-383.

ison_networkers *One-mode EIES dataset (Freeman and Freeman 1979)*

Description

A directed, simple, named, weighted graph with 32 nodes and 440 edges. Nodes are academics and edges illustrate the communication patterns on an Electronic Information Exchange System among them. Node attributes include the number of citations (`Citations`) and the discipline of the researchers (`Discipline`). Edge weights illustrate the number of emails sent from one academic to another over the studied time period.

Usage

```
data(ison_networkers)
```

Format

```
#> -- # EIES Networkers ------------------------------------------------------------
#> # A labelled, weighted, directed network of 32 nodes and 440 arcs
#>
#> -- Nodes
#> # A tibble: 32 x 3
#>   name               Discipline   Citations
#>   <chr>              <chr>            <dbl>
#> 1 Lin Freeman        Sociology           19
#> 2 Doug White         Anthropology         3
#> 3 Ev Rogers          Other              170
#> 4 Richard Alba       Sociology           23
#> 5 Phipps Arabie      Other               16
#> 6 Carol Barner-Barry Other                6
#> # i 26 more rows
#>
#> -- Ties
#> # A tibble: 440 x 3
#>    from    to weight
#>   <int> <int>  <dbl>
#> 1     1     2    488
#> 2     1     3     28
#> 3     1     4     65
#> 4     1     5     20
#> 5     1     6     65
#> 6     1     7     45
#> # i 434 more rows
#>
```

Source

networkdata package

References

Freeman, Sue C. and Linton C. Freeman. 1979. *The networkers network: A study of the impact of a new communications medium on sociometric structure*. Social Science Research Reports No 46. Irvine CA, University of California.

Wasserman Stanley and Katherine Faust. 1994. *Social Network Analysis: Methods and Applications*. Cambridge University Press, Cambridge.

---

ison_physicians — *Four multiplex one-mode physician diffusion data (Coleman, Katz, and Menzel, 1966)*

Description

Ron Burt prepared this data from Coleman, Katz and Menzel's 1966 study on medical innovation. They had collected data from physicians in four towns in Illinois: Peoria, Bloomington, Quincy and Galesburg. These four networks are held as separate networks in a list.

Coleman, Katz and Menzel were concerned with the impact of network ties on the physicians' adoption of a new drug, tetracycline. Data on three types of ties were collected in response to three questions:

- advice: "When you need information or advice about questions of therapy where do you usually turn?"
- discussion: "And who are the three or four physicians with whom you most often find yourself discussing cases or therapy in the course of an ordinary week – last week for instance?"
- friendship: "Would you tell me the first names of your three friends whom you see most often socially?"

Additional questions and records of prescriptions provided additional information:

- recorded date of tetracycline `adoption` date
- years in `practice` (note that these are `{messydates}`-compatible dates)
- `conferences` attended (those that attended "Specialty" conferences presumably also attended "General" conferences)
- regular subscriptions to medical `journals`
- `free_time` spent associating with doctors
- `discussions` on medical matters when with other doctors sociallyy
- memberships in `clubs` with other doctores
- number of top 3 `friends` that are doctors
- time practicing in current `community`
- `patients` load (ordinal)
- physical `proximity` to other physicians (in building/sharing office)
- medical `specialty` (GP/Internist/Pediatrician/Other)

Usage

```
data(ison_physicians)
```

Format

```
#> $Peoria
#> # A multiplex, directed network of 117 nodes and 147 friendship arcs, 242
#> advice arcs, and 154 discussion arcs
#>
#> -- Nodes
#> # A tibble: 117 x 12
#>   adoption specialty    conferences journals practice   community patients
#>      <dbl> <chr>        <chr>          <dbl> <chr>      <chr>     <chr>
#> 1        1 Pediatrician Specialty          9 1920..1929 20+yrs    101-150
#> 2       12 GP           None               5 1945..     -1yr      76-100
#> 3        8 Internist    General            7 1935..1939 10-20yrs  76-100
#> 4        9 GP           General            6 1940..1944 5-10yrs   51-75
#> 5        9 GP           General            4 1935..1939 10-20yrs  51-75
#> 6       10 Internist    None               7 1930..1934 10-20yrs  101-150
#> # i 111 more rows
#> # i 5 more variables: doc_freetime <dbl>, doc_discuss <dbl>, doc_friends <dbl>,
#> #   doc_club <dbl>, doc_proximity <chr>
#>
#> -- Ties
#> # A tibble: 543 x 3
#>    from    to type
#>   <int> <int> <chr>
#> 1     1     8 friendship
#> 2     1    58 friendship
#> 3     1    87 advice
#> 4     1    90 advice
#> 5     1   110 advice
#> 6     1   112 friendship
#> # i 537 more rows
#>
#>
#> $Bloomington
#> # A multiplex, directed network of 50 nodes and 60 friendship arcs, 57
#> discussion arcs, and 94 advice arcs
#>
#> -- Nodes
#> # A tibble: 50 x 12
#>   adoption specialty conferences journals practice   community patients
#>      <dbl> <chr>     <chr>          <dbl> <chr>      <chr>     <chr>
#> 1       98 Internist Specialty          8 1930..1934 10-20yrs  101-150
#> 2        1 GP        General            3 1945..     5-10yrs   76-100
#> 3       98 GP        Specialty          4 1930..1934 10-20yrs  101-150
#> 4        7 Internist None               3 1945..     -1yr      26-50
```

```
#> 5        6 Internist General            9 1935..1939 5-10yrs   76-100
#> 6        1 GP        Specialty          5 1935..1939 10-20yrs  101-150
#> # i 44 more rows
#> # i 5 more variables: doc_freetime <dbl>, doc_discuss <dbl>, doc_friends <dbl>,
#> #   doc_club <dbl>, doc_proximity <chr>
#>
#> -- Ties
#> # A tibble: 211 x 3
#>    from    to type
#>   <int> <int> <chr>
#> 1     1     3 friendship
#> 2     1    10 discussion
#> 3     1    24 advice
#> 4     1    44 advice
#> 5     2     4 advice
#> 6     2     6 advice
#> # i 205 more rows
#>
#>
#> $Quincy
#> # A multiplex, directed network of 44 nodes and 70 advice arcs, 52 discussion
#> arcs, and 52 friendship arcs
#>
#> -- Nodes
#> # A tibble: 44 x 12
#>   adoption specialty conferences journals practice   community patients
#>      <dbl> <chr>     <chr>          <dbl> <chr>      <chr>     <chr>
#> 1        2 Internist None               6 1935..1939 10-20yrs  151+
#> 2       18 GP        General            3 1920..1929 20+yrs    151+
#> 3       18 Internist None               5 1945..     -1yr      -25
#> 4        4 GP        General            3 1930..1934 20+yrs    151+
#> 5       18 GP        Specialty          4 1935..1939 10-20yrs  151+
#> 6        5 Internist General            5 ..1919     20+yrs    51-75
#> # i 38 more rows
#> # i 5 more variables: doc_freetime <dbl>, doc_discuss <dbl>, doc_friends <dbl>,
#> #   doc_club <dbl>, doc_proximity <chr>
#>
#> -- Ties
#> # A tibble: 174 x 3
#>    from    to type
#>   <int> <int> <chr>
#> 1     1     8 advice
#> 2     1     9 advice
#> 3     1    10 discussion
#> 4     1    13 friendship
#> 5     1    15 advice
#> 6     1    22 discussion
#> # i 168 more rows
```

```
#>
#>
#> $Galesburg
#> # A multiplex, directed network of 35 nodes and 74 advice arcs, 46 discussion
#> arcs, and 51 friendship arcs
#>
#> -- Nodes
#> # A tibble: 35 x 12
#>   adoption specialty conferences journals practice   community patients
#>      <dbl> <chr>     <chr>          <dbl> <chr>      <chr>     <chr>
#> 1       18 GP        General            4 1935..1939 5-10yrs   101-150
#> 2       18 GP        None               4 1935..1939 -1yr      151+
#> 3        4 GP        General            6 1945..     2-5yrs    51-75
#> 4        5 GP        None               4 1935..1939 10-20yrs  101-150
#> 5        8 Internist General            6 1935..1939 5-10yrs   151+
#> 6        4 Internist Specialty          8 ..1919     20+yrs    76-100
#> # i 29 more rows
#> # i 5 more variables: doc_freetime <dbl>, doc_discuss <dbl>, doc_friends <dbl>,
#> #   doc_club <dbl>, doc_proximity <chr>
#>
#> -- Ties
#> # A tibble: 171 x 3
#>    from    to type
#>   <int> <int> <chr>
#> 1     1     5 advice
#> 2     1     6 advice
#> 3     1    20 discussion
#> 4     1    23 discussion
#> 5     1    30 friendship
#> 6     1    31 friendship
#> # i 165 more rows
#>
```

Source

{networkdata}

References

Coleman, James, Elihu Katz, and Herbert Menzel. 1966. *Medical innovation: A diffusion study*. Indianapolis: The Bobbs-Merrill Company.

---

ison_southern_women *Two-mode southern women (Davis, Gardner and Gardner 1941)*

Description

Two-mode network dataset collected by Davis, Gardner and Gardner (1941) about the pattern of a group of women's participation at informal social events in Old City during a 9 month period, as reported in the *Old City Herald* in 1936. By convention, the nodes are named by the women's first names and the code numbers of the events, but the women's surnames and titles (Miss, Mrs.) are recorded here too. The events' dates are recorded in place of the Surname. A date describes the event and not the attendance, so each event enters the network on the date it is held, which the changes component records. `is_changing()` therefore marks this network TRUE, and `to_time()` returns the events held up to a given moment, without the women who attended no event by then.

Usage

```
data(ison_southern_women)
```

Format

```
#> -- # Southern Women Data -------------------------------------------------------
#> # A labelled, two-mode network of 18 womens and 14 social events and 89
#> participation ties
#>
#> -- Nodes
#> # A tibble: 32 x 5
#>   label   mode  Surname    Title active
#>   <chr>   <chr> <chr>      <chr> <lgl>
#> 1 Evelyn  women Jefferson  Mrs   TRUE
#> 2 Laura   women Mandeville Miss  TRUE
#> 3 Theresa women Anderson   Miss  TRUE
#> 4 Brenda  women Rogers     Miss  TRUE
#> # i 28 more rows
#>
#> -- Changes
#> # A tibble: 14 x 4
#>   time        node var        value
#>   <date[1d]> <int> <chr>     <list>
#> 1 1936-02-23    29 active TRUE<lgl>
#> 2 1936-02-25    23 active TRUE<lgl>
#> 3 1936-03-02    20 active TRUE<lgl>
#> 4 1936-03-15    25 active TRUE<lgl>
#> # i 10 more rows
#>
#> -- Ties
#> # A tibble: 89 x 2
#>    from    to
#>   <int> <int>
#> 1    14    29
#> 2    15    29
#> 3    17    29
#> 4    18    29
```

```
#> # i 85 more rows
#>
```

References

Davis, Allison, Burleigh B. Gardner, and Mary R. Gardner. 1941. *Deep South*. Chicago: University of Chicago Press.

`ison_tailorshop` *One-mode multiplex, longitudinal network of a Zambian tailor shop (Kapferer 1972)*

Description

Bruce Kapferer observed interactions in a tailor shop in Zambia, then Northern Rhodesia, over a period of ten months. His interest was in the changing patterns of alliance among the 39 workers during extended negotiations for higher wages.

Two layers of tie were recorded, each at two times seven months apart:

- *instrumental*: work- and assistance-related interactions
- *sociational*: friendship and other socioemotional interactions

The data are of particular interest because an abortive strike occurred after the first set of observations, and a successful strike took place after the second, so the two waves bracket a change in the workers' collective capacity.

Usage

```
data(ison_tailorshop)
```

Format

```
#> -- # Kapferer's Tailor Shop ----------------------------------------------------
#> # A longitudinal, labelled, multiplex, directed network of 39 workers and 256
#> instrumental arcs and 381 sociational ties over 2 waves
#>
#> -- Nodes
#> # A tibble: 39 x 1
#>   label
#>   <chr>
#> 1 Kamwefu
#> 2 Nkumbula
#> 3 Abraham
#> 4 Seams
#> 5 Chipata
#> 6 Donald
#> # i 33 more rows
```

```
#>
#> -- Ties
#> # A tibble: 637 x 4
#>    from    to layer         time
#>   <int> <int> <chr>        <int>
#> 1     3     1 instrumental     1
#> 2     4     1 instrumental     1
#> 3    14     1 instrumental     1
#> 4    11     2 instrumental     1
#> 5    12     2 instrumental     1
#> 6     1     3 instrumental     1
#> # i 631 more rows
#>
```

Details

Worker names have been title-cased. While the network as a whole is listed as directed, only the `instrumental` layer is directed. The `sociational` layer is symmetric. The info component records the directedness of each layer. Coercion to another class reciprocates that layer again, because 'igraph' or 'network' objects can only be directed or undirected.

References

Kapferer, Bruce. 1972. *Strategy and Transaction in an African Factory*. Manchester: Manchester University Press.

Dosdall, Henrik. 2019. "Kapferer (1972): Strategy and Transaction in an African Factory". In *Schlüsselwerke der Netzwerkforschung*, 249-252. Wiesbaden: Springer. doi:10.1007/9783658-217426_66

`make_collect` *Making networks of inter- and intra-package dependencies*

Description

These functions create networks of the dependencies between or within R packages:

- `collect_cran()` creates a network of the dependencies among the packages available on CRAN. It reads the `Depends`, `Imports`, `LinkingTo`, `Suggests`, and `Enhances` fields of each package's DESCRIPTION file, and creates a network in which the nodes are packages and the ties are dependencies of a given type.
- `collect_pkg()` creates a network of the dependencies among the functions defined in a directory of R scripts. It uses R's own parser to establish where each function is defined and which functions it calls, and creates a network in which the nodes are functions and the ties are calls.

Usage

```
collect_cran(
  pkg = "all",
  dependencies = c("Depends", "Imports", "LinkingTo"),
  max_dist = Inf,
  direction = c("out", "in", "all")
)

collect_pkg(dir = getwd(), external = FALSE)
```

Arguments

pkg	A character vector of one or more package names, from which dependencies are collected. By default "all", which collects the dependencies among all the packages currently available on CRAN.
dependencies	A character vector naming the dependency fields to collect, from "Depends", "Imports", "LinkingTo", "Suggests", and "Enhances". By default `c("Depends", "Imports", "LinkingTo")`, the dependencies that must be installed alongside a package.
max_dist	The maximum number of steps from pkg to collect. By default infinite, i.e. the whole dependency closure.
direction	Whether to collect the packages that pkg depends upon, "out" by default, the packages that depend upon pkg, "in", or both, "all".
dir	Character string with the path of the directory in which to look for R scripts. By default the current working directory. Where `dir` holds a DESCRIPTION file and an R folder, as a package does, the R folder is searched.
external	Logical. Where TRUE, calls to functions that are not defined in `dir`, such as those from other packages, are included as nodes too. By default FALSE, since these are numerous and rarely of interest.

Details

Dependency networks grow quickly, and are most useful once scoped. `collect_cran()` therefore collects only the `Depends`, `Imports`, and `LinkingTo` fields by default, since these are the dependencies that must be installed alongside a package, as in `utils::install.packages()`. Adding `Suggests` grows the dependency closure of a package by two orders of magnitude. For the same reason, `collect_pkg()` collects only calls to the functions defined in the directory by default.

Both return networks that can be scoped further using, for example, `to_ego()`, `to_uniplex()`, `to_giant()`, `delete_isolates()`, `to_blockmodel()`, or `to_subgraph()`.

`collect_cran()` relies on `utils::available.packages()`, which caches the repository index for an hour by default. Set `options(max.repo.cache.age = )` for a fresher or staler snapshot.

Note that these functions are not as actively maintained as others in the package, so please let us know if any are not currently working for you or if there are missing import routines by raising an issue on Github.

Value

A `tidygraph` object representing the network of package dependencies or function dependencies in a package.

Source

https://www.r-bloggers.com/2016/01/r-graph-objects-igraph-vs-network/

Inspired by Jakob Gepp's `helfRlein::get_network()`, https://github.com/STATWORX/helfRlein/blob/master/R/get_netw

See Also

`to_ego()`, `to_uniplex()`, `delete_isolates()`

Other makes: `make_create`, `make_ego`, `make_emails`, `make_explicit`, `make_groups`, `make_growth`, `make_learning`, `make_play`, `make_random`, `make_read`, `make_write`

Examples

```
## Not run:
# The packages {manynet} depends upon, directly and indirectly:
collect_cran("manynet")
# The packages that depend directly upon {manynet}:
collect_cran("manynet", direction = "in", max_dist = 1)

## End(Not run)
## Not run:
# The network of calls among the functions in the working directory:
collect_pkg()
# Collapsed onto generics, where the directory is a package:
# to_blockmodel(collect_pkg(), node_attribute(collect_pkg(), "generic"))

## End(Not run)
```

`make_create` *Making networks with defined structures*

Description

These functions create networks with particular structural properties.

- `create_empty()` creates an empty network without any ties.
- `create_filled()` creates a filled network with every possible tie realised.
- `create_ring()` creates a ring or chord network where each nodes' neighbours form a clique.
- `create_star()` creates a network with a maximally central node.
- `create_tree()` creates a network with successive branches.
- `create_lattice()` creates a network that forms a regular tiling.
- `create_components()` creates a network that clusters nodes into separate components.

- `create_core()` creates a network in which a certain proportion of 'core' nodes are densely tied to each other, and the rest peripheral, tied only to the core.
- `create_degree()` creates a network with a given (out/in)degree sequence, which can also be used to create k-regular networks.
- `create_cycle()` creates a network in which all the nodes form a single closed chain.
- `create_wheel()` creates a network in which a single dominant node is tied to all the nodes in a cycle.
- `create_windmill()` creates a network in which several cliques, or blades, share a single hub node. Unless a blade `width` is given, the number of blades is balanced against their size.

Some of these structures are constrained in two-mode networks. Since ties in two-mode networks can only run between the modes, a two-mode cycle must alternate between them, and so can only be as long as twice the number of nodes in the smaller mode. Similarly, the rim of a two-mode wheel alternates between the modes, and its hub, drawn from the first mode, can only be tied to the second mode's rim nodes. The hub of a two-mode windmill is also drawn from the first mode, and is tied to every blade node in the second mode, while each other first-mode node is tied to its own blade. Where `n` is larger than such a structure can accommodate, the largest such structure is created and the surplus nodes are added as isolates, with a message. This is also the case for windmills whose blades cannot be equally sized.

These functions can create either one-mode or two-mode networks. To create a one-mode network, pass the main argument `n` a single integer, indicating the number of nodes in the network. To create a two-mode network, pass `n` a vector of *two* integers, where the first integer indicates the number of nodes in the first mode, and the second integer indicates the number of nodes in the second mode. As an alternative, an existing network can be provided to `n` and the number of modes, nodes, and directedness will be inferred.

Usage

```
create_empty(n, directed = FALSE)

create_filled(n, directed = FALSE)

create_ring(n, directed = FALSE, width = 1, ...)

create_star(n, directed = FALSE)

create_tree(n, directed = FALSE, width = 2)

create_lattice(n, directed = FALSE, width = 8)

create_components(n, directed = FALSE, membership = NULL)

create_degree(n, outdegree = NULL, indegree = NULL)

create_core(n, directed = FALSE, mark = NULL)

create_windmill(n, directed = FALSE, width = NULL)
```

```
create_cycle(n, directed = FALSE)

create_wheel(n, directed = FALSE)
```

Arguments

`n` Given:

- A single integer, e.g. `n = 10`, a one-mode network will be created.
- A vector of two integers, e.g. `n = c(5,10)`, a two-mode network will be created.
- A manynet-compatible object, a network of the same dimensions will be created.

`directed` Logical whether the network should be directed. By default `directed = FALSE`, but for the functions that the usage above shows another default for. A network given as `n` sets the direction instead. If the opposite direction is desired, use `to_redirected()` on the output of these functions. In two-mode networks, the directed argument is ignored.

`width` Integer specifying the width of the ring, breadth of the branches, number of nodes in each blade of a windmill, maximum extent of the neighbourbood, or number of ties that bridge each pair of islands.

`...` Additional arguments passed on to `igraph::make_ring()`.

`membership` A vector of partition membership as integers. If left as `NULL` (the default), nodes in each mode will be assigned to two, equally sized partitions.

`outdegree` Numeric scalar or vector indicating the desired outdegree distribution. Note that a scalar (single number) will result in a k-regular graph. By default NULL. If `n` is an existing network object and the outdegree is not specified, then the outdegree distribution will be inferred from that of the network. If only the indegree is specified, then in one-mode networks the outdegree will mirror it, and in two-mode networks the same number of ties will be spread as evenly as possible across the nodes in the first mode. If neither is specified, the sparsest connected structure is created: a cycle in one-mode networks, and in two-mode networks one in which the larger mode is 1-regular.

`indegree` Numeric vector indicating the desired indegree distribution. By default NULL but not required unless a directed network is desired. If `n` is an existing directed network object and the indegree is not specified, then the indegree distribution will be inferred from that of the network. Otherwise it is filled in from the outdegree as described above.

`mark` A logical vector the length of the nodes in the network. This can be created by, among other things, any `node_is_*()` function.

Value

A `stocnet` object is returned, which can be coerced into other types of objects using `as_edgelist()`, `as_matrix()`, `as_tidygraph()`, `as_igraph()`, or `as_network()`. `create_ring()`, `create_lattice()`, `generate_random()`, `generate_smallworld()`, and `generate_scalefree()` return a `tbl_graph` or `igraph` object for now, since the released versions of the packages built on this one expect one from them.

By default, most networks are created as undirected. This can be overruled with the argument `directed = TRUE`. This will return a directed network in which the arcs are out-facing or equivalent. This direction can be swapped using `to_redirected()`. In two-mode networks, the directed argument is ignored.

Lattice graphs

`create_lattice()` creates two-dimensional lattices with as even dimensions as possible. The `width` parameter sets the maximum (in or out) degree of any node:

- `width = 4` creates a square grid lattice. Such a network is bipartite, that is partitionable into two types that are not adjacent to any of their own type.
- `width = 6` creates a triangular grid lattice, a square grid with one diagonal across each square.
- `width = 8` creates a King's move lattice, a square grid with both diagonals across each square.
- `width = 12` creates a lattice where each node is tied to every node within two steps on the square grid.

If the number of nodes is a prime number, it will only return a chain (a single dimensional lattice).

When `n` is a vector of two equal integers, `create_lattice()` creates a two-mode honeycomb (hexagonal) lattice, in which each node is tied to at most three nodes of the other mode. Where the number of nodes allows, no node hangs from a single tie.

See Also

as

Other makes: `make_collect`, `make_ego`, `make_emails`, `make_explicit`, `make_groups`, `make_growth`, `make_learning`, `make_play`, `make_random`, `make_read`, `make_write`

Examples

```
create_empty(10)
create_filled(10)
create_ring(8, width = 2)
create_star(12)
create_tree(c(7,8))
create_lattice(12, width = 4)
create_lattice(12, width = 6)
create_lattice(c(6,6))
create_components(10, membership = c(1,1,1,2,2,2,3,3,3,3))
create_degree(10, outdegree = rep(1:5, 2))
create_degree(10)
create_degree(c(6,4))
create_core(6)
  create_windmill(7)
  create_windmill(7, width = 2)
  create_windmill(c(4,6))
  create_cycle(6)
  create_cycle(c(4,6))
  create_wheel(6)
  create_wheel(c(4,6))
```

make_ego *Making ego networks through interviewing*

Description

This function creates an ego network through interactive interview questions. It currently only supports a simplex, directed network of one or two modes. These directed networks can be reformatted as undirected using `to_undirected()`. Multiplex networks can be collected separately and then joined together afterwards.

The function supports the use of rosters or a maximum number of alters to collect. If a roster is provided it will offer ego all names. The function can also prompt ego to interpret each node's attributes, or about how ego considers their alters to be related.

Usage

```
collect_ego(
  ego = NULL,
  max_alters = Inf,
  roster = NULL,
  interpreter = FALSE,
  interrelater = FALSE,
  twomode = FALSE
)
```

Arguments

`ego`	A character string. If desired, the name of ego can be declared as an argument. Otherwise the first prompt of the function will be to enter a name for ego.
`max_alters`	The maximum number of alters to collect. By default infinity, but many name generators will expect a maximum of e.g. 5 alters to be named.
`roster`	A vector of node names to offer as potential alters for ego.
`interpreter`	Logical. If TRUE, then it will ask for which attributes to collect and give prompts for each attribute for each node in the network. By default FALSE.
`interrelater`	Logical. If TRUE, then it will ask for the contacts from each of the alters perspectives too.
`twomode`	Logical. If TRUE, then it will assign ego to the first mode and all alters to a second mode.

See Also

Other makes: `make_collect`, `make_create`, `make_emails`, `make_explicit`, `make_groups`, `make_growth`, `make_learning`, `make_play`, `make_random`, `make_read`, `make_write`

make_emails *Making networks from email metadata*

Description

`collect_emails()` creates a network of who writes to whom from the headers of a mailbox that has been exported to file. It reads the `From`, `To`, `Cc`, `Bcc`, `Date`, and `Message-ID` fields of each message, and creates a network in which the nodes are email addresses and each tie runs from the sender of a message to one of its recipients. Only the headers are read: neither the subject nor the body of any message is collected.

Usage

```
collect_emails(file = file.choose(), ego = NULL, twomode = FALSE)
```

Arguments

`file`	A character string with the path to an 'mbox' file, an '.eml' file, or a directory holding any number of either, including 'Maildir' folders. If left unspecified, an OS-specific file picker is opened to help users select a file. Note that a file picker cannot select a directory, so the path to a directory must be given.
`ego`	A character vector of the email address(es) of the owner of the mailbox. By default NULL, in which case ego is inferred from the messages. Use FALSE where the messages are not from one mailbox.
`twomode`	Logical. If TRUE, a two-mode network of addresses and the messages they sent or received is returned, in which the ties are layered by the role each address had in each message. By default FALSE, which returns a directed one-mode network of addresses.

Details

Most email clients and services can export a mailbox to file. Google Takeout, Thunderbird (via "ImportExportTools NG"), and Apple Mail ("Mailbox > Export Mailbox...") write the 'mbox' format, in which the messages follow one another in a single file. Outlook and others can save messages as individual '.eml' files instead. `collect_emails()` reads either, as well as 'Maildir' folders, and needs neither a password nor a connection to the mail server.

Emails are events, and so the network returned is dynamic. By default each tie is stamped with the time its message was sent, and several messages between the same two addresses are several ties. Use `to_time()` or `to_aggregated()` to scope or aggregate them. Where `twomode = TRUE` the time belongs to the message rather than to any one of its ties, and so each message instead enters the network at the time it was sent.

An address is counted once for each message, as its sender if it sent it, and otherwise under the first of `To`, `Cc`, and `Bcc` that names it. Messages repeated in the export, as where a message carries several labels, are collected once, and messages without both a sender and at least one other recipient are dropped.

Value

A `stocnet` object. Nodes hold the address as their label, the name most often displayed alongside it, and whether the address is ego's.

Ego

A mailbox holds the messages that its owner sent or received, and so its boundary is that of an ego network: two alters are only tied where ego was party to the message. It is not an egocentric network in the sense of `is_egocentric()`, though, since the ties are records of events rather than ego's reports of them.

Not every export is one person's mailbox. By default, the address party to the most messages is taken to be ego where it is party to at least half of them, and the network is otherwise taken to have no ego, as in a corpus of several mailboxes. Note that the address of a mailing list will be found this way too. Name ego's address(es) in `ego` to override this, or use `ego = FALSE`.

Source

Inspired by Nathan Yau's "Downloading Your Email Metadata", https://flowingdata.com/2014/05/07/downloading-your-email-metadata/

See Also

`collect_ego()`, `to_time()`, `to_aggregated()`, `to_unlabelled()`

Other makes: `make_collect`, `make_create`, `make_ego`, `make_explicit`, `make_groups`, `make_growth`, `make_learning`, `make_play`, `make_random`, `make_read`, `make_write`

Examples

```
mbox <- tempfile(fileext = ".mbox")
writeLines(c("From - Mon Jan  6 09:15:00 2025",
             "Message-ID: <1@example.org>",
             "Date: Mon, 6 Jan 2025 09:15:00 +0100",
             "From: Ada <ada@example.org>",
             "To: Ben <ben@example.org>, cy@example.org",
             "",
             "Shall we meet?",
             "",
             "From - Mon Jan  6 10:02:00 2025",
             "Message-ID: <2@example.org>",
             "Date: Mon, 6 Jan 2025 10:02:00 +0100",
             "From: Ben <ben@example.org>",
             "To: Ada <ada@example.org>",
             "Cc: cy@example.org",
             "",
             "Yes, let's."), mbox)
collect_emails(mbox)
collect_emails(mbox, twomode = TRUE)
```

make_explicit *Making networks with explicit ties*

Description

This function creates a network from a vector of explicitly named nodes and ties between them. `create_explicit()` largely wraps `igraph::graph_from_literal()`, but will also accept character input and not just a formula, and will never simplify the result.

Ties are indicated by `-`, and directed ties (arcs) require `+` at either or both ends. Ties are separated by commas, and isolates can be added as an additional, unlinked node after the comma within the formula. Sets of nodes can be linked to other sets of nodes through use of a semi-colon. See the example for a demonstration.

Usage

```
create_explicit(...)
```

Arguments

`...`	Arguments passed on to `{igraph}`.

See Also

Other makes: `make_collect`, `make_create`, `make_ego`, `make_emails`, `make_groups`, `make_growth`, `make_learning`, `make_play`, `make_random`, `make_read`, `make_write`

Examples

```
create_explicit(A -+ B, B -+ C, A +-+ C, D, E:F:G-+A, E:F+-+G:H)
```

make_groups *Making networks with groups*

Description

These functions are similar to the `create_*` functions, but include some element of randomisation. They are particularly useful for creating a distribution of networks for exploring or testing network properties. The networks here start from a planted structure, to which some random noise is then added.

- `generate_smallworld()` generates a small-world structure via ring rewiring at some probability.
- `generate_islands()` generates an islands model.
- `generate_communities()` generates communities of unequal size among nodes of unequal degree.

- `generate_core()` generates a core and a periphery, as the random counterpart of `create_core()`.

`generate_islands()` and `generate_communities()` plant discrete groups, and record the group of each node in the node attribute `community`. `generate_core()` plants two, and records whether each node is in the core in the node attribute `core`. `generate_smallworld()` plants local clusters around a ring instead, so its nodes do not belong to discrete groups.

These functions can create either one-mode or two-mode networks. To create a one-mode network, pass the main argument `n` a single integer, indicating the number of nodes in the network. To create a two-mode network, pass `n` a vector of *two* integers, where the first integer indicates the number of nodes in the first mode, and the second integer indicates the number of nodes in the second mode. As an alternative, an existing network can be provided to `n` and the number of modes, nodes, and directedness will be inferred.

Usage

```
generate_smallworld(n, p = 0.05, directed = FALSE, width = 2)

generate_islands(n, p = 0.5, directed = FALSE, width = 1, groups = 2, ...)

generate_communities(
  n,
  p = 0.1,
  directed = FALSE,
  groups = NULL,
  degree = NULL,
  max_degree = NULL,
  degree_exp = 2,
  community_exp = 1
)

generate_core(n, p = 0.5, directed = FALSE, mark = NULL)
```

Arguments

`n` Given:

- A single integer, e.g. `n = 10`, a one-mode network will be created.
- A vector of two integers, e.g. `n = c(5,10)`, a two-mode network will be created.
- A manynet-compatible object, a network of the same dimensions will be created.

`p` The main probability of the model, which is what makes a network of it more or less random. It always follows `n`, and what it is the probability of depends on the model. For `generate_smallworld()`, it is the probability that each tie of the ring is rewired, by default 0.05. For `generate_islands()`, it is the probability of a tie between two nodes of the same island, by default 0.5. For `generate_communities()`, it is the probability that a tie of a node leads out of its community, known as the mixing parameter, by default 0.1: a low value gives

communities that are easy to tell apart, and a value above 0.5 gives communities with more ties out than in. For `generate_core()`, it is the probability of a tie between a node of the core and a node of the periphery, by default 0.5, or a vector of three: the probabilities of a tie within the core, between the two, and within the periphery.

`directed` Logical whether the network should be directed. By default `directed = FALSE`, but for the functions that the usage above shows another default for. A network given as `n` sets the direction instead. If the opposite direction is desired, use `to_redirected()` on the output of these functions. In two-mode networks, the directed argument is ignored.

`width` Integer specifying the width of the ring, breadth of the branches, number of nodes in each blade of a windmill, maximum extent of the neighbourhood, or number of ties that bridge each pair of islands.

`groups` How the nodes are divided into groups, in one of three ways:

- A single integer, e.g. `groups = 3`, is the number of groups, which are then as equal in size as they can be.
- A vector of integers that sum to the number of nodes, e.g. `groups = c(10, 20, 30)` for 60 nodes, is the size of each group.
- Any other two integers, e.g. `groups = c(10, 30)`, are the smallest and the largest that a group can be, and the sizes of the groups are drawn at random from between them.

Two integers that sum to the number of nodes are read as two sizes, and not as the smallest and the largest. In a two-mode network, a group has nodes of both modes, in the proportion of the two modes in the network as a whole, and its size is the size of the two together. The group of each node is recorded in the node attribute `community`. For `generate_islands()`, by default 2, and sizes between a smallest and a largest are all as likely as one another. For `generate_communities()`, by default the smallest and the largest are `degree + 1` and `max_degree + 1`, so that a node of any degree can find all the ties it keeps within its community, and sizes between them are drawn from a power law.

`...` Arguments under the names they had in earlier versions, which still work: `islands` for `groups` and `bridges` for `width` in `generate_islands()`.

`degree` The number of ties that a node sends. For `generate_communities()`, this is the mean over the nodes, by default 10, or less in a network too small for that. In a directed network, it is the mean number of ties that a node sends, which is also the mean number that a node receives. In a two-mode network, it is the mean degree of the nodes of the first mode, and the mean degree of the second mode follows from it, since both modes have the same ties: it is `degree * n[1] / n[2]`.

`max_degree` The maximum degree of the nodes. By default three times `degree`, or `n - 1` if that is smaller. In a two-mode network, this can be one number for each mode, and is by default three times the mean degree of the mode, or the number of nodes in the other mode if that is smaller.

`degree_exp` The exponent of the power law from which the degrees are drawn. By default 2. In a two-mode or directed network, this can be one number for each mode, or for the ties sent and the ties received.

community_exp	The exponent of the power law from which the community sizes are drawn. By default 1.
mark	A logical vector the length of the nodes in the network. This can be created by, among other things, any node_is_*() function.

Details

In generate_islands(), the nodes of an island are tied to one another with probability p, and width further ties bridge each pair of islands. In a two-mode network, a node of each mode that share an island are tied with probability p. Where a network is given as n, p is instead inferred from it, so that as many ties are expected as that network has.

generate_communities() follows the benchmark of Lancichinetti, Fortunato, and Radicchi (2008). Degrees and community sizes are each drawn from a power law, so that there are a few large hubs and a few large communities, as in many observed networks. Each node then keeps a share 1 - p of its ties within its community and sends the remaining share to nodes in other communities. Ties within and between communities are drawn at random from all the simple networks with those degrees. Where a degree sequence cannot be realised as a simple network, or a node has more ties to keep than its community has other members, ties are dropped or sent elsewhere, so the realised degrees and mixing can differ a little from those asked for. They differ most in small networks and where there are few communities.

The benchmark is defined for one-mode networks. For a two-mode network, generate_communities() extends it by analogy, in the same way that generate_islands() does for islands. Each community has nodes of both modes, in the proportion of the two modes in the network as a whole, and each mode draws its degrees from a power law of its own. A node then keeps a share 1 - p of its ties for the nodes of the other mode in its community. The two modes must keep the same number of ties in each community, and the ties that one mode has too many of there are dropped, so the realised degrees are a little lower than those asked for where p is low.

A directed network is treated in the same way, with the ties that a node sends and the ties that it receives in place of the two modes. Each node draws both from a power law, and keeps a share 1 - p of each within its one community. Lancichinetti and Fortunato (2009) define a directed benchmark too, from which this differs in how the two are drawn and reconciled, so it should not be taken for theirs.

generate_islands() is a simpler model of the same kind: the ties in each island are random, rather than following from degrees that differ, and each pair of islands is joined by a fixed number of bridges, rather than by a share of each node's ties.

In generate_core(), with a single p every pair of nodes in the core is tied and no pair in the periphery is, as in create_core(), and what is random is which nodes of the periphery each node of the core is tied to. With three, the core can be less than complete and the periphery less than empty, as Borgatti and Everett (2000) describe of observed networks. mark says which nodes are in the core, with TRUE for a node of the core, and by default they are the first half of the nodes, or of each mode. Note that create_core() for now reads a logical mark the other way round, with TRUE for a node of the periphery.

Value

A stocnet object is returned, which can be coerced into other types of objects using as_edgelist(), as_matrix(), as_tidygraph(), as_igraph(), or as_network(). create_ring(), create_lattice(),

generate_random(), generate_smallworld(), and generate_scalefree() return a tbl_graph or igraph object for now, since the released versions of the packages built on this one expect one from them.

By default, most networks are created as undirected. This can be overruled with the argument directed = TRUE. This will return a directed network in which the arcs are out-facing or equivalent. This direction can be swapped using to_redirected(). In two-mode networks, the directed argument is ignored.

References

On small-world networks:

Watts, Duncan J., and Steven H. Strogatz. 1998. "Collective Dynamics of 'Small-World' Networks." *Nature* 393(6684):440–42. doi:10.1038/30918.

On communities:

Lancichinetti, Andrea, Santo Fortunato, and Filippo Radicchi. 2008. "Benchmark Graphs for Testing Community Detection Algorithms." *Physical Review E* 78(4):046110. doi:10.1103/PhysRevE.78.046110

Lancichinetti, Andrea, and Santo Fortunato. 2009. "Benchmarks for Testing Community Detection Algorithms on Directed and Weighted Graphs with Overlapping Communities." *Physical Review E* 80(1):016118. doi:10.1103/PhysRevE.80.016118

On cores and peripheries:

Borgatti, Stephen P., and Martin G. Everett. 2000. "Models of Core/Periphery Structures." *Social Networks* 21(4):375–95. doi:10.1016/S03788733(99)000192

See Also

Other makes: make_collect, make_create, make_ego, make_emails, make_explicit, make_growth, make_learning, make_play, make_random, make_read, make_write

Examples

```
generate_smallworld(12, 0.025)
generate_smallworld(12, 0.25)
generate_islands(10)
generate_islands(12, groups = c(3, 4, 5), width = 2)
generate_islands(c(10, 6))
generate_communities(50)
generate_communities(50, p = 0.4)
generate_communities(50, directed = TRUE, groups = 3)
generate_communities(c(60, 40))
generate_core(12)
generate_core(12, p = c(0.8, 0.3, 0.1))
```

make_growth *Making networks that grow*

Description

These functions are similar to the `create_*` functions, but include some element of randomisation. They are particularly useful for creating a distribution of networks for exploring or testing network properties. The networks here grow: nodes arrive one at a time, and each new node ties to nodes that arrived before it. The functions differ in how a new node chooses whom to tie to.

- `generate_scalefree()` generates a scale-free structure via preferential attachment at some probability.
- `generate_fire()` generates a forest fire model.
- `generate_citations()` generates a citations model, in which each new node cites some of the nodes before it.

These functions can create either one-mode or two-mode networks. To create a one-mode network, pass the main argument `n` a single integer, indicating the number of nodes in the network. To create a two-mode network, pass `n` a vector of *two* integers, where the first integer indicates the number of nodes in the first mode, and the second integer indicates the number of nodes in the second mode. As an alternative, an existing network can be provided to `n` and the number of modes, nodes, and directedness will be inferred.

Usage

```
generate_scalefree(n, p = 1, directed = FALSE)

generate_fire(n, p = 0, directed = FALSE, contacts = 1, their_in = 1, ...)

generate_citations(
  n,
  p = 1,
  directed = FALSE,
  degree = sample(1:4, 1),
  agebins = max(1, n/10),
  ...
)
```

Arguments

`n` Given:

- A single integer, e.g. `n = 10`, a one-mode network will be created.
- A vector of two integers, e.g. `n = c(5,10)`, a two-mode network will be created.
- A manynet-compatible object, a network of the same dimensions will be created.

p — The main probability of the model, which is what makes a network of it more or less random. It always follows n, and what it is the probability of depends on the model. For generate_scalefree(), it is the power of the preferential attachment, by default 1. For generate_fire(), it is the probability that the fire burns along each tie that a contact sends, and so that the new node ties to the node at its other end, by default 0. In a two-mode network, it is instead the probability of burning across each two-path, and so of closing a four-cycle. For generate_citations(), it is the probability that a node chooses each node that it cites at random, by default 1, where it otherwise cites the node that came most recently, so that 0 returns the same network every time.

directed — Logical whether the network should be directed. By default directed = FALSE, but for the functions that the usage above shows another default for. A network given as n sets the direction instead. If the opposite direction is desired, use to_redirected() on the output of these functions. In two-mode networks, the directed argument is ignored.

contacts — Number of contacts or ambassadors chosen from among existing nodes in the network. By default 1. See igraph::sample_forestfire().

their_in — Probability of tieing to a contact's incoming ties. By default 1. This is a factor on p rather than a probability in its own right, so p = 0 gives a tree whatever their_in is set to. In a two-mode network, p * their_in is instead the probability that a newly burned node re-ignites and spreads the fire further.

... — Arguments under the names they had in earlier versions, which still work: their_out for p in generate_fire(), and ties for degree in generate_citations().

degree — The number of ties that a node sends. For generate_citations(), this is how many of the nodes before it each new node cites, by default a random number from 1 to 4. Inf has each node cite every node before it. In a two-mode network, each new node of the first mode ties to this many nodes of the second mode.

agebins — Number of aging bins. By default either $\frac{n}{10}$ or 1, whichever is the larger. See igraphr::sample_last_cit() for more.

Details

In a one-mode network, each burn step is a single hop, so each tie the fire creates closes a triangle. A tie in a two-mode network crosses modes, so the shortest closure there is the four-cycle rather than the triangle. In a two-mode network each burn step is therefore a two-path hop across the other mode, and each tie the fire creates closes a four-cycle. Both modes grow over the course of the simulation.

In generate_citations(), a node that chooses at random whom to cite prefers the nodes that were cited most recently. With p = 0 nothing is left to chance, and each node cites the degree nodes that arrived just before it. With degree = Inf as well, each node cites every node before it, which in a directed network is the complete citation network that tidygraph::create_citation() returns. With p = 1 a node can cite another more than once.

Value

A stocnet object is returned, which can be coerced into other types of objects using as_edgelist(), as_matrix(), as_tidygraph(), as_igraph(), or as_network(). create_ring(), create_lattice(),

generate_random(), generate_smallworld(), and generate_scalefree() return a tbl_graph or igraph object for now, since the released versions of the packages built on this one expect one from them.

By default, most networks are created as undirected. This can be overruled with the argument directed = TRUE. This will return a directed network in which the arcs are out-facing or equivalent. This direction can be swapped using to_redirected(). In two-mode networks, the directed argument is ignored.

References

On scale-free networks:

Barabasi, Albert-Laszlo, and Reka Albert. 1999. "Emergence of Scaling in Random Networks." *Science* 286(5439):509–12. doi:10.1126/science.286.5439.509

On the forest-fire model:

Leskovec, Jure, Jon Kleinberg, and Christos Faloutsos. 2007. "Graph evolution: Densification and shrinking diameters". *ACM transactions on Knowledge Discovery from Data*, 1(1): 2-es. doi:10.1145/1217299.1217301

See Also

Other makes: make_collect, make_create, make_ego, make_emails, make_explicit, make_groups, make_learning, make_play, make_random, make_read, make_write

Examples

```
generate_scalefree(12, 0.25)
generate_scalefree(12, 1.25)
generate_fire(10)
generate_fire(10, p = 0.3)
generate_fire(c(10, 6))
generate_citations(10)
generate_citations(10, p = 0.5, directed = TRUE, degree = 3)
generate_citations(10, p = 0, directed = TRUE, degree = Inf)
generate_citations(c(10, 6))
```

---

make_learning *Making learning models on networks*

Description

These functions allow learning games to be played upon networks.

- play_learning() plays a learning model upon a network.
- play_segregation() plays a Schelling segregation model upon a network.

Usage

```
play_learning(.data, beliefs, closeness = Inf, steps, epsilon = 5e-04)

play_segregation(
  .data,
  attribute,
  heterophily = 0,
  who_moves = c("ordered", "random", "most_dissatisfied"),
  choice_function = c("satisficing", "optimising", "minimising"),
  steps
)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`beliefs` A vector indicating the probabilities nodes put on some outcome being 'true'.

`closeness` A threshold at which beliefs are too different to influence each other. By default `Inf`, i.e. there is no threshold.

`steps` The number of steps forward in learning. By default the number of nodes in the network.

`epsilon` The maximum difference in beliefs accepted for convergence to a consensus.

`attribute` A string naming some nodal attribute in the network. Currently only tested for binary attributes.

`heterophily` A score ranging between -1 and 1 as a threshold for how heterophilous nodes will accept their neighbours to be. A single proportion means this threshold is shared by all nodes, but it can also be a vector the same length of the nodes in the network for issuing different thresholds to different nodes. By default this is 0, meaning nodes will be dissatisfied if more than half of their neighbours differ on the given attribute.

`who_moves` One of the following options: "ordered" (the default) checks each node in turn for whether they are dissatisfied and there is an available space that they can move to, "random" will check a node at random, and "most_dissatisfied" will check (one of) the most dissatisfied nodes first.

`choice_function` One of the following options: "satisficing" (the default) will move the node to any coordinates that satisfy their heterophily threshold, "optimising" will move the node to coordinates that are most homophilous, and "minimising" distance will move the node to the next nearest unoccupied coordinates.

Learning models

The default is a Degroot learning model, but if `closeness` is defined as anything less than infinity, this becomes a Deffuant model. A Deffuant model is similar to a Degroot model, however nodes only learn from other nodes whose beliefs are not too dissimilar from their own.

References

DeGroot, Morris H. 1974. "Reaching a consensus", *Journal of the American Statistical Association*, 69(345): 118–21. doi:10.1080/01621459.1974.10480137

Deffuant, Guillaume, David Neau, Frederic Amblard, and Gérard Weisbuch. 2000. "Mixing beliefs among interacting agents", *Advances in Complex Systems*, 3(1): 87-98. doi:10.1142/S0219525900000078

Golub, Benjamin, and Matthew O. Jackson 2010. "Naive learning in social networks and the wisdom of crowds", *American Economic Journal*, 2(1): 112-49. doi:10.1257/mic.2.1.112

See Also

Other makes: `make_collect`, `make_create`, `make_ego`, `make_emails`, `make_explicit`, `make_groups`, `make_growth`, `make_play`, `make_random`, `make_read`, `make_write`

Other models: `make_play`

Examples

```
play_learning(ison_networkers,
    rbinom(net_nodes(ison_networkers),1,prob = 0.25))
startValues <- rbinom(100,1,prob = 0.5)
startValues[sample(seq_len(100), round(100*0.2))] <- NA
latticeEg <- create_lattice(100)
latticeEg <- add_node_attribute(latticeEg, "startValues", startValues)
latticeEg
play_segregation(latticeEg, "startValues", 0.5)
```

---

make_mnet — *Multilevel, multiplex, multimodal, signed, dynamic or longitudinal changing networks*

Description

The 'mnet' class of network object is an additional class layered on top of the 'igraph' and 'tbl_graph' classes. Under the hood it is an 'igraph' object, which enables all the igraph functions to operate. It is also a 'tbl_graph' object, which enables it to be used with `{ggraph}`. However, 'mnet' objects offer prettier printing and a consistent structure that enables more complex forms of networks to be contained in a single object.

Usage

```
## S3 method for class 'mnet'
print(x, ..., n = 12)

print_all(x, ...)
```

Arguments

x	An object of class "mnet" or "tbl_graph".
...	Other arguments passed to or from other methods.
n	Number of observations to print across all network components, i.e. nodes, changes, and ties. By default 12.

Nodes

Nodes are held as vertices and vertex attributes in the 'igraph' object, but printed as a nodelist. Here the convention is for the first column of the nodelist to be called 'name' and records the labels of the nodes. Additional reserved columns include 'active' for changing networks and 'type' for multimodal networks.

Changes

Changes, that is a list of changes to the nodes in the network, are held internally as a graph attribute in the 'igraph' object, but printed as a changelist. Here the convention is for the 'wave' or 'time' column to appear first, followed by 'node' indicating to which node the change applies, 'var' for the variable to which the change applies, and 'value' for the new value to be applied.

Ties

Ties are held as edges and edge attributes in the 'igraph' object, but printed as an edgelist. Here the convention is for the first column of the edgelist to be called 'from' and the second column 'to', even if the network is not directed. Additional reserved columns include 'weight' for weighted networks, 'wave' for longitudinal networks, 'type' for multiplex networks, and 'sign' for signed networks.

Printing

When printed, 'mnet' objects will print to the console any information stored about the network's name, or its types of nodes or ties. It will also describe key features of the network, such as whether the network is multiplex, weighted, directed, etc.

It will then print tibbles for the nodes, changes, and ties in the network, as appropriate. That is, if there is no nodal data (e.g. it is an unlabelled network without any other nodal attributes), then this will be skipped. Similarly, if no nodal changes are logged, this information will be skipped too.

make_play *Making diffusion models on networks*

Description

These functions simulate diffusion or compartment models upon a network.

- play_diffusion() runs a single simulation of a compartment model, allowing the results to be visualised and examined.

These functions allow both a full range of compartment models, as well as simplex and complex diffusion to be simulated upon a network.

Usage

```
play_diffusion(
  .data,
  seeds = 1,
  contact = NULL,
  prevalence = 0,
  thresholds = 1,
  transmissibility = 1,
  latency = 0,
  recovery = 0,
  waning = 0,
  fatality = 0,
  immune = NULL,
  steps,
  old_version = FALSE
)
```

Arguments

.data	An object of a {manynet}-consistent class: • adjacency or incidence matrix from {base} R • edgelist data.frame from {base} R or tbl/tbl_df from {tibble} • stocnet stocnet, from the {manynet} package • igraph igraph, from the {igraph} package • network network, from the {network} package • tidygraph tbl_graph, from the {tidygraph} package
seeds	A valid mark vector the length of the number of nodes in the network.
contact	A matrix or network that replaces ".data" with some other explicit contact network, e.g. create_components(.data, membership = node_in_structural(.data)). Can be of arbitrary complexity, but must of the same dimensions as .data.

`prevalence` The proportion that global prevalence contributes to diffusion. That is, if prevalence is 0.5, then the current number of infections is multiplied by 0.5 and added "prevalence" is 0 by default, i.e. there is no global mechanism. Note that this is endogenously defined and is updated at the outset of each step.

`thresholds` A numeric vector indicating the thresholds each node has. By default 1. A single number means a generic threshold; for thresholds that vary among the population please use a vector the length of the number of nodes in the network. If 1 or larger, the threshold is interpreted as a simple count of the number of contacts/exposures sufficient for infection. If less than 1, the threshold is interpreted as complex, where the threshold concerns the proportion of contacts.

`transmissibility` The transmission rate probability, β. By default 1, which means any node for which the threshold is met or exceeded will become infected. Anything lower means a correspondingly lower probability of adoption, even when the threshold is met or exceeded.

`latency` The inverse probability those who have been exposed become infectious (infected), σ or κ. For example, if exposed individuals take, on average, four days to become infectious, then $\sigma = 0.75$ (1/1-0.75 = 1/0.25 = 4). By default 0, which means those exposed become immediately infectious (i.e. an SI model). Anything higher results in e.g. a SEI model.

`recovery` The probability those who are infected recover, γ. For example, if infected individuals take, on average, four days to recover, then $\gamma = 0.25$. By default 0, which means there is no recovery (i.e. an SI model). Anything higher results in an SIR model.

`waning` The probability those who are recovered become susceptible again, ξ. For example, if recovered individuals take, on average, four days to lose their immunity, then $\xi = 0.25$. By default 0, which means any recovered individuals retain lifelong immunity (i.e. an SIR model). Anything higher results in e.g. a SIRS model. $\xi = 1$ would mean there is no period of immunity, e.g. an SIS model.

`fatality` The probability those who are infected are removed from the network, α. Note that fatality is distinct from a natural mortality rate. By default $\alpha = 0$, which means that there is no fatality. Where $\alpha > 0$, the nodal attribute 'active' will be added if it is not already present.

`immune` A logical or numeric vector identifying nodes that begin the diffusion process as already recovered. This could be interpreted as those who are vaccinated or equivalent. Note however that a waning parameter will affect these nodes too. By default NULL, indicating that no nodes begin immune.

`steps` The number of steps forward in the diffusion to play. By default the number of nodes in the network. If `steps = Inf` then the diffusion process will continue until there are no new infections or all nodes are infected.

`old_version` This is included to maintain backward compatibility with the old version of this function, that would return a special object. The new version adds the diffusion event record as changes to the original network.

Simple and complex diffusion

By default, the function will simulate a simple diffusion process in which some infectious disease or idea diffuses from seeds through contacts at some constant rate (`transmissibility`).

These `seeds` can be specified by a vector index (the number of the position of each node in the network that should serve as a seed) or as a logical vector where TRUE is interpreted as already infected.

`thresholds` can be set such that adoption/infection requires more than one (the default) contact already being infected. This parameter also accepts a vector so that thresholds can vary.

Complex diffusion is where the `thresholds` are defined less than one. In this case, the thresholds are interpreted as proportional. That is, the threshold to adoption/infection is defined by the proportion of the node's contacts infected.

Nodes that cannot be infected can be indicated as `immune` with a logical vector or index, similar to how `seeds` are identified. Note that `immune` nodes are interpreted internally as Recovered (R) and are thus subject to `waning` (see below).

Compartment models

Compartment models are flexible models of diffusion or contagion, where nodes are compartmentalised into one of two or more categories.

The most basic model is the SI model. The SI model is the default in `play_diffusion()`/`play_diffusions()`, where nodes can only move from the Susceptible (S) category to the Infected (I) category. Whether nodes move from S to I depends on whether they are exposed to the infection, for instance through a contact, the `transmissibility` of the disease, and their `thresholds` to the disease.

Another common model is the SIR model. Here nodes move from S to I, as above, but additionally they can move from I to a Recovered (R) status. The probability that an infected node recovers at a timepoint is controlled by the `recovery` parameter.

The next most common models are the SIS and SIRS models. Here nodes move from S to I or additionally to R, as above, but additionally they can move from I or R back to a Susceptible (S) state. This probability is governed by the `waning` parameter. Where `recover > 0` and `waning = 1`, the Recovery (R) state will be skipped and the node will return immediately to the Susceptible (S) compartment.

Lastly, these functions also offer the possibility of specifying a latency period in which nodes have been infected but are not yet infectious. Where `latency > 0`, an additional Exposed (E) compartment is introduced that governs the probability that a node moves from this E compartment to infectiousness (I). This can be used in in SEI, SEIS, SEIR, and SEIRS models.

See Also

Other makes: `make_collect`, `make_create`, `make_ego`, `make_emails`, `make_explicit`, `make_groups`, `make_growth`, `make_learning`, `make_random`, `make_read`, `make_write`

Other models: `make_learning`

Examples

```
smeg <- generate_smallworld(15, 0.025)
```

make_random *Making unconditional and conditional random networks*

Description

These functions are similar to the `create_*` functions, but include some element of randomisation. They are particularly useful for creating a distribution of networks for exploring or testing network properties.

- `generate_random()` generates a random network with ties appearing at some probability.
- `generate_degree()` generates a random network consistent with a given degree distribution, as the random counterpart of `create_degree()`. `generate_configuration()` is an alias, since this is the configuration model.
- `generate_man()` generates a random network conditional on the dyad census of Mutual, Asymmetric, and Null dyads, respectively.
- `generate_utilities()` generates the utility that each node finds in each other node, as a signed and weighted network.

These functions can create either one-mode or two-mode networks. To create a one-mode network, pass the main argument `n` a single integer, indicating the number of nodes in the network. To create a two-mode network, pass `n` a vector of *two* integers, where the first integer indicates the number of nodes in the first mode, and the second integer indicates the number of nodes in the second mode. As an alternative, an existing network can be provided to `n` and the number of modes, nodes, and directedness will be inferred.

Usage

```
generate_random(n, p = 0.5, directed = FALSE, with_attr = TRUE)

generate_degree(n, p = 1, directed = FALSE, outdegree = NULL, indegree = NULL)

generate_configuration(
  n,
  p = 1,
  directed = FALSE,
  outdegree = NULL,
  indegree = NULL
)

generate_man(n, p = NULL, directed = NULL, ...)

generate_utilities(
  n,
  p = 1,
  directed = TRUE,
  form = c("normal", "uniform", "relative"),
```

```
  steps = 1,
  volatility = 0.1,
  inertia = 0
)
```

Arguments

`n` Given:

- A single integer, e.g. `n = 10`, a one-mode network will be created.
- A vector of two integers, e.g. `n = c(5,10)`, a two-mode network will be created.
- A manynet-compatible object, a network of the same dimensions will be created.

`p` The main probability of the model, which is what makes a network of it more or less random. It always follows `n`, and what it is the probability of depends on the model. For `generate_random()`, it is the proportion of possible ties in the network that are realised or, if an integer greater than 1, the number of ties in the network, by default 0.5. For `generate_degree()`, it is the share of the ties that are switched with one another, by default 1, so that 0 returns the network as it was. For `generate_man()`, it is a vector of three: the proportions of Mutual, Asymmetric, and Null dyads, which is inferred from `n` if that is a network and is otherwise `c(0.25, 0.5, 0.25)`. For `generate_utilities()`, it is the share of the utilities that are ties, by default 1.

`directed` Logical whether the network should be directed. By default `directed = FALSE`, but for the functions that the usage above shows another default for. A network given as `n` sets the direction instead. If the opposite direction is desired, use `to_redirected()` on the output of these functions. In two-mode networks, the directed argument is ignored.

`with_attr` Logical whether any attributes of the object should be retained. By default TRUE.

`outdegree` Numeric scalar or vector indicating the desired outdegree distribution. Note that a scalar (single number) will result in a k-regular graph. By default NULL. If `n` is an existing network object and the outdegree is not specified, then the outdegree distribution will be inferred from that of the network. If only the indegree is specified, then in one-mode networks the outdegree will mirror it, and in two-mode networks the same number of ties will be spread as evenly as possible across the nodes in the first mode. If neither is specified, the sparsest connected structure is created: a cycle in one-mode networks, and in two-mode networks one in which the larger mode is 1-regular.

`indegree` Numeric vector indicating the desired indegree distribution. By default NULL but not required unless a directed network is desired. If `n` is an existing directed network object and the indegree is not specified, then the indegree distribution will be inferred from that of the network. Otherwise it is filled in from the outdegree as described above.

`...` Arguments under the names they had in earlier versions, which still work: `man` for `p` in `generate_man()`.

`form` How the utilities are distributed between -1 and 1.

- "normal" (the default) draws them from a normal distribution around 0, so that most nodes are close to indifferent about most others and only a few are much liked or much disliked.
- "uniform" draws every utility between -1 and 1 as likely as any other.
- "relative" draws them from a normal distribution and then divides each node's utilities by the largest of them, whether liked or disliked, so that each node's utilities are relative to the node it feels most strongly about.

`steps` Number of moments at which to return the utilities. By default 1. If more than 1, the utilities change from each moment to the next by as much as `volatility` allows, and a longitudinal network with a wave for each moment is returned.

`volatility` How much the utilities change between moments. At each step a new set of utilities is drawn as `form` describes, multiplied by `volatility`, and added to those there are. By default 0.1, so that a utility changes by a tenth of a new draw.

`inertia` The share of utilities that keep their value at each step, from 0 to 1. These are the utilities that would have changed the least, so that with some inertia utilities stay as they are until a change large enough comes along, rather than all drifting a little at every step. By default 0, so that every utility changes at every step.

Details

`generate_degree()` keeps the degree of every node and is random in which nodes are tied. The degrees are those of the network given as `n`, or those given as `outdegree` and `indegree`, as for `create_degree()`. It then switches the ends of pairs of ties, which changes who is tied to whom but not how many ties anyone has. With `p = 1`, the default, a network is drawn anew from those degrees. With a lower `p` only that share of the ties are switched, so that the result is in part the network it started from.

For `generate_man()`, counts such as `p = c(10, 0, 20)` are read as relative weights and normalised, so the dyad census is reproduced in expectation rather than exactly. The default is the dyad distribution of a random (Erdős-Rényi) digraph in which each arc is present with probability 0.5. The network is directed unless there are no asymmetric dyads, as with `p = c(0.1, 0, 0.1)`, since it then has no direction to speak of. `directed` overrules this either way, and where an undirected network is asked for from asymmetric dyads, a dyad is tied where it would have been mutual or asymmetric. The same holds for two-mode networks, whose ties are undirected, so only the sum of the mutual and asymmetric dyads is consequential there.

A utility is how much a node gains or loses from a tie to another node, from -1 for the most it could lose to 1 for the most it could gain. `generate_utilities()` returns these as the weights of a signed network. In a one-mode network every node has a utility for every other node. In a two-mode network each node of the first mode has a utility for each node of the second mode.

Utilities say what ties nodes would like, not what ties there are, so they can serve as the basis for a model of how ties come about. The examples show two such models. Where a node can tie to another without its consent, the ties are the positive utilities. Where a tie needs the consent of both nodes, the ties are those where the lower of the two utilities is positive.

With `p` below 1, a node is indifferent to the utilities closest to 0, and only the share `p` of them that are furthest from 0 are ties, whether positive or negative. With `directed = FALSE`, the two nodes of a pair find the same in each other, and the network is undirected.

Where there is more than one step, how far from 0 a utility must be to be a tie is set at the first moment and kept, so a tie can come and go as a utility crosses it. A utility cannot rise above 1 or fall below -1, and one that a change would take further stays at that limit until a change brings it back. Over many steps the utilities therefore spread out towards the limits, and more of them are far enough from 0 to be ties than at the start.

Value

A `stocnet` object is returned, which can be coerced into other types of objects using `as_edgelist()`, `as_matrix()`, `as_tidygraph()`, `as_igraph()`, or `as_network()`. `create_ring()`, `create_lattice()`, `generate_random()`, `generate_smallworld()`, and `generate_scalefree()` return a `tbl_graph` or `igraph` object for now, since the released versions of the packages built on this one expect one from them.

By default, most networks are created as undirected. This can be overruled with the argument `directed = TRUE`. This will return a directed network in which the arcs are out-facing or equivalent. This direction can be swapped using `to_redirected()`. In two-mode networks, the directed argument is ignored.

References

On random networks:

Erdos, Paul, and Alfred Renyi. 1959. "On Random Graphs I" *Publicationes Mathematicae*. 6: 290–297.

On configuration models:

Bollobas, Bela. 1980. "A Probabilistic Proof of an Asymptotic Formula for the Number of Labelled Regular Graphs". *European Journal of Combinatorics* 1: 311-316.

On dyad-census conditioned networks:

Holland, Paul W., and Samuel Leinhardt. 1976. “Local Structure in Social Networks.” In D. Heise (Ed.), *Sociological Methodology*, pp 1-45. San Francisco: Jossey-Bass.

See Also

Other makes: `make_collect`, `make_create`, `make_ego`, `make_emails`, `make_explicit`, `make_groups`, `make_growth`, `make_learning`, `make_play`, `make_read`, `make_write`

Examples

```
generate_random(12, 0.4)
# generate_random(c(6, 6), 0.4)
generate_degree(12, outdegree = 3)
generate_degree(create_core(12), p = 0.25)
generate_man(6)
generate_man(6, p = c(0.3, 0, 0.7))
generate_man(c(4, 6))
(utils <- generate_utilities(6))
generate_utilities(6, p = 0.25)
generate_utilities(6, directed = FALSE)
generate_utilities(c(4, 6), form = "uniform")
```

```
generate_utilities(6, steps = 3)
generate_utilities(6, steps = 3, volatility = 0.5, inertia = 0.8)
# the ties that one node wants
to_unsigned(utils, keep = "positive")
# the ties that both nodes want
to_unsigned(to_undirected(utils, rule = "min"), keep = "positive")
```

`make_read` *Making networks from external files*

Description

Researchers regularly need to work with a variety of external data formats. The following functions enable importing from some common external file formats into objects that {manynet} and other graph/network packages in R can work with:

- `read_matrix()` imports adjacency matrices from Excel/csv files.
- `read_edgelist()` imports edgelists from Excel/csv files.
- `read_nodelist()` imports nodelists from Excel/csv files.
- `read_pajek()` imports Pajek (.net or .paj) files.
- `read_ucinet()` imports UCINET files from the header (.##h).
- `read_dynetml()` imports DyNetML interchange format for rich social network data.
- `read_graphml()` imports GraphML files, including those exported by Network Canvas.
- `read_gml()` imports GML files.
- `read_gdf()` imports GDF files.
- `read_gexf()` imports GEXF files, such as those exported by Gephi.
- `read_hif()` imports Hypergraph Interchange Format (HIF) files.
- `read_geojson()` imports the lines of a GeoJSON file, such as roads, rivers, or routes, as a spatial network.

Usage

```
read_matrix(file = file.choose(), sv = c("comma", "semi-colon"), ...)

read_edgelist(file = file.choose(), sv = c("comma", "semi-colon"), ...)

read_nodelist(file = file.choose(), sv = c("comma", "semi-colon"), ...)

read_pajek(file = file.choose(), ties = NULL, ...)

read_ucinet(file = file.choose())

read_dynetml(file = file.choose())
```

```
read_graphml(file = file.choose(), ego = TRUE)

read_gml(file = file.choose())

read_gdf(file = file.choose())

read_gexf(file = file.choose())

read_geojson(file = file.choose(), junctions = TRUE, tolerance = 0)

read_hif(file = file.choose())
```

Arguments

`file`	A character string with the system path to the file to import. If left unspecified, an OS-specific file picker is opened to help users select it. Note that in `read_ucinet()` the file path should be to the header file (.##h), if it exists and that it is currently not possible to import multiple networks from a single UCINET file. Please convert these one by one.
`sv`	Allows users to specify whether their csv file is `"comma"` (English, the default) or `"semi-colon"` (European) separated.
`...`	Additional parameters passed to the read/write function.
`ties`	A character string indicating the ties/network, where the data contains several.
`ego`	Logical, whether to add ego as a node in ego-centric formats such as Network Canvas, where ego is otherwise only recorded at the level of the network. By default TRUE. Where ego is added, ties from ego to each of its alters are also added, and every tie gains a 'by' column identifying the ego that reported it, which makes the network an egocentric network (see `is_egocentric()`).
`junctions`	Whether to keep only the junctions and ends of the lines as nodes, joining the lines that meet at a node where nothing else does. By default TRUE. Use FALSE to keep every line in the file as a tie of its own.
`tolerance`	How far apart two ends of lines can be and still be read as the same node, in the units of the file's coordinates. By default 0, so that only ends at the very same position are one node. Surveyed or digitised lines seldom meet at the very same position, so a small tolerance, say a metre, often recovers a better connected network.

Details

There are a number of repositories for network data that hold various datasets in different formats. See for example:

- networkdata
- GML datasets
- SNAP Stanford Large Network Dataset Collection
- Colorado Index of Complex Networks

Please let us know if you identify any further repositories of social or political networks and we would be happy to add them here.

The `_ucinet` functions only work with relatively recent UCINET file formats, e.g. type 6406 files. To import earlier UCINET file types, you will need to update them first. To import multiple matrices packed into a single UCINET file, you will need to unpack them and convert them one by one.

`read_graphml()` reads the file itself rather than relying on igraph, so that keys declared `for="all"` and files holding more than one graph are read rather than quietly discarded. Where several graphs are present, they are combined into a single network and distinguished by a 'graph' node attribute.

Network Canvas exports are recognised by their namespace and read accordingly. Each interview session is exported as a separate graph, so the sessions are combined into one network in which each session is a component. Since ego is recorded at the level of the graph rather than as a node, and ego-alter ties are left implicit, ego is added as a node with ties to each of its alters, and every tie records the ego that reported it in a 'by' column, making the result an egocentric network (see `is_egocentric()`). It is not a cognitive social structure, since each ego reports on alters of its own rather than on one roster of nodes (see `is_cognitive()`). Use `ego = FALSE` to keep just the alters and the ties between them. Note that node types are reported in a character 'nodeset' column rather than a logical 'type' column, because ego networks are not two-mode: ties within a nodeset are exactly what is collected. Note too that, since alters are particular to a session, the sessions share no nodes, so `as_matrix()` on such a network returns a large and very sparse three-dimensional array.

`read_gexf()` reads the node and tie attributes declared in the file, as well as the visualisation elements Gephi adds, so that positions are available as 'x' and 'y' node attributes, and sizes and colours as 'size' and 'color'. Nodes are named from their labels where the file gives them, since node ids are required by the format and so are an export artefact. Dynamic files are read with their 'start' and 'end' times, which makes the result a dynamic network (see `is_dynamic()`). Where a directed network contains ties declared undirected or mutual, those ties are reciprocated, since a network is directed or not as a whole.

`read_hif()` reads a file in the Hypergraph Interchange Format (Coll et al. 2025), a JSON format that hypergraph libraries such as HyperNetX, XGI, Hypergraphx, and SimpleHypergraphs.jl share. It returns a 'stocnet' object that holds each hyperedge as a tie among its nodes (see the Hypergraphs section of `make_stocnet()`). The weights and attributes of the nodes and of the hyperedges become their attributes, and those of each node's membership of a hyperedge are kept with it, as is the side of a directed hyperedge that the node is on: its 'tail' is read as 'from' and its 'head' as 'to'. The metadata become the information about the network, and a file that says it holds an abstract simplicial complex is marked as one. An attribute that holds more than a single value is kept as the JSON that records it.

`read_geojson()` reads each line in the file as a tie between its two ends, and each place where the ends of lines meet as a node. The positions of the nodes are held as 'x' and 'y' node attributes, and 'z' where the file gives elevations, in whatever coordinate reference system the file declares, which makes the result a spatial network (see `is_spatial()`). The properties of a line become the attributes of its tie, together with a 'length' that is measured along the line, unless the file gives the lines a 'length' of its own: in metres where the coordinates are longitudes and latitudes, or those of a Mercator projection (EPSG:3395 or EPSG:3857), and otherwise in the units of the coordinates. The shape of the line itself is not retained. Any points in the file describe the nodes at their positions, so that their properties, such as names, become node attributes, and a point that no line ends at is read as an isolate. Lines are often divided where nothing meets them, say where the surface or the name of a road changes, and so by default the ties that meet at such a node are joined, unless a point describes that node. A joined tie records the number of 'segments' it joins, and its 'length' is the

sum of theirs. Its other numeric attributes are their means, weighted by their lengths, which is not what is wanted of an attribute that is itself a total. Where its segments differ on any other attribute, each of their values is given, separated by semi-colons. Lines that share the same two ends are kept as separate ties, since they are different routes between the same places.

Value

`read_edgelist()` and `read_nodelist()` import a list rather than a network, so they return a tibble, which can then be coerced or combined into a network from there.

Every other `read_*()` function returns a stocnet object (see `make_stocnet()`). This is the class that holds the most of what a file can contain, such as the network's metadata, more than two modes, several layers, and the times at which nodes and ties are present. Note that a network can be coerced into any other format with `{manynet}`'s `as_` methods.

Source

`read_ucinet()` kindly supplied by Christian Steglich, constructed on 18 June 2015.

References

Coll, Martín, Cliff A. Joslyn, Nicholas W. Landry, Quintino Francesco Lotito, Audun Myers, Joshua Pickard, Brenda Praggastis, and Przemysław Szufel. 2025. "HIF: The hypergraph interchange format for higher-order networks". *Network Science* 13: e21. doi:10.1017/nws.2025.10018

See Also

as

Other makes: `make_collect`, `make_create`, `make_ego`, `make_emails`, `make_explicit`, `make_groups`, `make_growth`, `make_learning`, `make_play`, `make_random`, `make_write`

`make_stocnet` *Multilevel, multiplex, multimodal, signed, dynamic or longitudinal changing networks*

Description

The 'stocnet' class of network object is a list of six elements: nodes, ties, (nodal) changes, globals, missings, and info metadata about the network as a whole. Every element but info is a table, which is what its plural name signifies, and any of them may be NULL. This offers a consistent and flexible structure that enables more complex forms of networks to be contained in a single object.

Unlike 'mnet' objects, 'stocnet' objects are not layered on top of 'igraph' or 'tbl_graph' objects, but instead are a list of tibbles and metadata. Unlike 'igraph' or 'tbl_graph' objects, 'stocnet' objects typically include more complex multimodal, longitudinal, or dynamic networks. They also typically include more metadata about the network, such as the names of the types of nodes and ties in the network. In other words, they are made not just for network analysis, but also network modelling.

Usage

```
make_stocnet(
  info = NULL,
  nodes = NULL,
  ties = NULL,
  changes = NULL,
  globals = NULL,
  missings = NULL
)

## S3 method for class 'stocnet'
print(x, ..., n = 12)

validate_stocnet(.data)
```

Arguments

`info`	A list of metadata about the network as a whole. This can include the name of the network, as well as the names of the types of nodes and ties in the network. For example, the info component could include a 'name' element with the name of the network, a 'modes' element with a character vector of the names of the types of nodes in the network, and a 'layers' element with a character vector of the names of the types of ties in the network. By default NULL.
`nodes`	A tibble of nodes in the network, with one row per node and one column for the node labels, which should be called 'label'. Additional columns can be included for node attributes, such as 'active' for changing networks and 'mode' for multimodal networks. By default NULL.
`ties`	A tibble of ties in the network, with one row per tie and at least two columns for the node labels of the tie endpoints, which should be called 'from' and 'to', even if the network is not directed. Additional columns can be included for tie attributes, such as 'weight' for weighted networks and 'layer' for multiplex networks. By default NULL.
`changes`	A tibble of nodal changes in the network, with one row per change and at least three columns for the node label of the change, which should be called 'node', the variable to which the change applies, which should be called 'var', and the new value to be applied, which should be called 'value'. Additional columns can be included for the time of the change, such as 'wave' or 'time'. By default NULL.
`globals`	A tibble of global variables in the network, with one row per change and at least three columns for the variable to which the change applies, which should be called 'var', the time of the change, which should be called 'time', and the new value to be applied, which should be called 'value'. By default NULL.
`missings`	A tibble of the ties the network could have observed and did not, with one row per tie and at least two columns for its endpoints, which should be called 'from' and 'to'. Only the ties that no node's non-response implies need be listed here; see the Missingness section. By default NULL.
`x`	An object of class "mnet" or "tbl_graph".

...	Other arguments passed to or from other methods.
n	Number of observations to print across all network components, i.e. nodes, changes, and ties. By default 12.
.data	An object of a {manynet}-consistent class: • adjacency or incidence matrix from {base} R • edgelist data.frame from {base} R or tbl/tbl_df from {tibble} • stocnet stocnet, from the {manynet} package • igraph igraph, from the {igraph} package • network network, from the {network} package • tidygraph tbl_graph, from the {tidygraph} package

Details

The package includes a validation function for stocnet objects, validate_stocnet(). This checks that the object has the correct structure and required components, and suggests improvements to the structure (e.g. correcting or adding reserved names) where possible. The required and reserved names for the components of a stocnet object are described below.

Info

There are several reserved names for the elements of the info component of a stocnet object.

- 'name' should be a single character string with the name of the network.
- 'modes' should be a character vector of the names of the modes of the nodes in a multimodal network.
- 'layers' should be a character vector of the names of the layers of the ties in a multiplex or multilayer network.
- 'directed' should be a logical indicating whether each layer is directed or undirected. If there are multiple layers, this can be a named logical vector with the directedness of each layer, where the names correspond to the layer names. An undirected layer holds one row per dyad in the ties component, whether or not the other layers are directed. Since an 'igraph' or 'network' object is directed or undirected as a whole, coercion reciprocates the undirected layers of a network that has any directed layer, and coercion back collapses them again. This keeps the degree of a node in an undirected layer the same in every class, and makes the round trip lossless.
- 'signed' should be a logical indicating whether the network is signed. If there are multiple layers, this can be a named logical vector that says which layers are signed, where the names correspond to the layer names. Since the sign of a tie is held as the sign of its weight, this records that a network is signed where none of its ties is negative. A network with any negative weight is signed whether or not this is given.
- 'focal' should be a character vector indicating which variables are dependent (endogenous). These may be tie layers (dependent networks) and/or nodal variables recorded in the changes component (dependent behaviours), as used when modelling with e.g. 'RSiena'.
- 'centered' should be a named logical vector indicating, for each covariate (nodal or dyadic), whether it should be centered when modelling. The names correspond to the covariate variable or layer names.

- 'siena' can hold a list of metadata specific to 'RSiena' 'sienadata' objects that has no native meaning elsewhere (e.g. the 'allowOnly', 'uponly', and 'downonly' estimation constraints of dependent variables, the composition change option, node set relations, and the originating 'RSiena' version). This is populated by `as_stocnet()` and consumed by `as_siena()` so that coercion to and from 'sienadata' objects is lossless.
- 'doi' can be a character string with the DOI of the network, if it is from a published source.
- 'date' can be an integer of the year or the date the network represents.
- 'location' can be a character string with the location of the network.
- 'source' can be a character string indicating whether the network is observed or synthetic. If it is observed, the 'method' of data collection can be further specified. For example, the source could be 'observed', 'synthetic', 'survey', 'archival', 'digital trace', etc. If it is synthetic, then more details about how the network was generated can be included.

Many of these elements are drawn from the GRAND project's metadata standards for networks, which are designed to be consistent with the FAIR principles for data management.

In addition to these reserved names, the info component can include metadata relating to each layer of the network, such as the names of the types of nodes and ties in each layer, as well as the names of the dependent and independent layers in a multiplex network. These must be named as one of the layer names. There are some reserved names for these elements too:

- 'sender' should be a character string naming the type of node that sends ties in this layer.
- 'recipient' should be a character string naming the type of node that receives ties in this layer.
- 'observation' should be a character string naming how the network was observed: "cross-sectional" for a single observation, "panel" for a few complete re-observations, "event" for a stream of many records, and "egocentric" or "cognitive" for the two designs that ask each respondent about a network of their own. Where the layers were observed differently, this can be a layer-named character vector. A layer named "cross-sectional" in an otherwise longitudinal or dynamic network states something that holds throughout, a constant dyadic covariate, and is carried into every moment the network is scoped to.
- 'update' should be a character string indicating how each record of a tie relates to the record before it: "replace", where each states the tie's value afresh, or "increment", where each adds to it. Where the layers are updated differently, this can be a layer-named character vector. This says nothing about a tie recorded as an interval, in 'begin' and 'end' columns, since such a tie carries its own lifespan. See the Time section of `to_time()` for how the two are scoped.

Nodes

There are several reserved names for the columns of the nodes component of a stocnet object.

- 'label' should be a character vector of the labels of the nodes in the network.
- 'mode' should be a character vector of the modes of the nodes in a multimodal network.
- 'active' should be a logical vector indicating the initial status of nodes in changing networks. Inactive nodes are not in the network, so cannot hold incoming nor outgoing ties.
- 'na' should be a logical vector indicating which nodes were non-responsive, despite being in the network. Non-responsive nodes are in the network, so can hold incoming ties, but cannot report outgoing ties. Where a layer's ties name who reported them, a non-responsive node instead missed its whole report of that layer. See `as_missinglist()`.

Changes

There are several required names for the columns of the changes component of a stocnet object (if one is included).

- 'time' can be an integer (e.g. for a wave) or date (e.g. POSIXct or mdate) vector of the times at which changes occur.
- 'node' must be an index (or names) of the node to which the change applies.
- 'var' must be a string vector naming the variable to which the change applies, such as 'active' for changing networks.
- 'value' must be the new value that should be applied at that change (or incremented, as appropriate). Note that the value column can be of any class, such as logical for changes to active status, or numeric for changes to a nodal attribute. These values are held internally as a list within the tibble, so that they can be of any class and length, but printed as a tibble with a 'value' column that shows the first value and a type label for the class of the value.

A change states what a variable becomes from that moment on, so it is carried forward until another change states otherwise. A node that does not report at one wave and reports again at the next therefore holds two changes of the 'na' variable, one each way. Unlike the ties, then, changes are always recorded at a moment and always carried forward: an interval over which a nodal variable holds is stated as two changes, one at each end, and there is no 'update' to declare. `as_changelist()` takes a `time` and returns the changes in force at it.

There is one reserved name for a further column.

- 'layer' should be a character vector naming the layer a change applies to, where it applies to one layer and not to others. A change that names no layer applies to every layer.

Ties

There are several required names for the columns of the ties component of a stocnet object (if one is included).

- 'from' must be an integer vector of the nodes sending each tie
- 'to' must be an integer vector of the nodes receiving each tie

There are also several reserved names for the columns of the ties component of a stocnet object.

- 'layer' should be a character vector of the layer of each tie in a multiplex or multilayer network
- 'by' should be an integer vector of the node that reported each tie, as in a cognitive social structure or an egocentric network.
- 'about' should be an integer vector of the node each tie is about, as where a sender tells a receiver about a target. Like 'from' and 'to', these two are always held as indices into the nodes, never as labels. `make_stocnet()` converts labels given for any of the four into those indices, and coercion to a labelled class converts them back. Either may be `NA`, for a layer that no node reported or that is about no node. See the Third nodes section.
- 'weight' should be a numeric vector of the weights of the ties in a weighted network If the weight vector includes also negative values, then the network is a signed network, and the sign of the tie can be determined from the weight. Missing values, `NA`, in the weight vector indicate that the tie value is missing, not the tie itself, which is still present in the network. For

individual missing ties beyond those implied by inactive or non-responsive nodes, please add them to the missings component as an edgelist.

- 'time' should be a numeric, character, or date vector of the moment at which each tie was recorded.
- 'begin' and 'end' should instead give the interval over which each tie lasts, where the network records that rather than a moment.
- 'increment' and 'replace' should give the change each record makes to a tie's value, where the network records a stream of such changes. These are renamed to 'weight' on coercion, and what they said is kept in `info$update`.

How a network records time in its ties, and what each of these columns therefore means, is set out in the Time section of `to_time()`.

Hypergraphs

A hyperedge is a tie among any number of nodes, such as the people at a meeting, the authors of a paper, or the reactants and products of a reaction. A 'stocnet' object holds each hyperedge as one row of its ties, so that a hyperedge has a weight, a layer, a moment, and any other attribute in the same columns as a tie between two nodes has.

- 'with' should be a list of integer vectors, each the nodes that are the members of a hyperedge. A hyperedge names no node in 'from' or 'to', which are `NA` on its row, and a tie between two nodes holds `NULL` in 'with'. `make_stocnet()` converts labels given in 'with' into indices, and adds 'from' and 'to' where the ties give only 'with'. A hyperedge may have a single member, or none.
- 'label' should be a character vector naming each hyperedge.
- 'weight' is the weight of the hyperedge as a whole.
- Any column whose name begins 'with_' should be a list that holds a value for each member of each hyperedge, in the order 'with' lists them. 'with_weight' is the weight of each node's membership, and 'with_direction' says whether each member is on the 'from' or on the 'to' side of a hyperedge that has two sides, as the senders and the receivers of a message have. The hyperedges are directed where the network is.

Two elements of the info describe the hyperedges.

- 'hyperedges' should be a character string naming what the hyperedges are, as 'modes' names what the nodes are.
- 'simplicial' should be `TRUE` where the hyperedges are the largest faces of a simplicial complex, in which every subset of a hyperedge is a hyperedge too. Only the largest faces are held.

A network may hold ties between two nodes in some layers and hyperedges in others, so long as each layer holds one or the other.

Other classes have no hyperedges of their own. `as_igraph()` and `as_tidygraph()` return a two-mode network in which each hyperedge is a node of the second mode, marked so that `is_hypergraph()` recognises it and `as_stocnet()` returns the hyperedges. `as_network()` returns a 'network' object that holds the hyperedges as `{network}` does, and `as_matrix()` the incidence matrix of nodes by hyperedges. See `to_hypergraph()`, `to_simplicial()`, and `read_hif()` for how to make or read a hypergraph, and `to_twomode()` and `to_mode()` for how to hold or project one as a network of ties between two nodes.

Globals

There are several required names for the columns of the globals component of a stocnet object (if one is included).

- 'var' must be a string vector naming the global variable
- 'value' must be the value the variable takes from that moment on. There may be an additional column:
- 'time' should be a numeric, character, or date vector of the moment at which each global attribute is updated. Globals are carried forward the way changes are: a value holds from the moment it is recorded at until another value states otherwise.

Third nodes

Some designs record a third node for each tie, as well as the two nodes it runs between. A cognitive social structure (Krackhardt 1987) asks each respondent about every pair of nodes in one roster, and so records each tie once for each node that reported it, in the 'by' column. An egocentric design also records the ego that reported each tie in 'by', but each ego names alters of its own. Gossip records the node that a tie is about in the 'about' column. `is_cognitive()`, `is_egocentric()`, and `is_gossip()` mark these, and the 'observation' information tells the first two apart.

Each report is a network of its own, so reports of one tie by two reporters are two records and not parallel ties. `as_matrix()` returns such a network as a three-dimensional array of senders, receivers, and reporters (or targets), and `as_stocnet()` reads one back, once told what the third dimension holds. Functions that renumber or delete nodes renumber these columns too, and drop the reports of a deleted reporter.

Missings

The missings component lists the ties the network could have observed and did not, one row each, where these are not already implied by an inactive or a non-responsive node. It takes the same columns as the ties component:

- 'from' and 'to' must be integer vectors of the nodes at each end
- 'layer' and 'time' should name the layer and moment the tie was missing from, where the network records them.

Since most missing tie data is a node that did not report, this component is usually NULL. `as_missinglist()` returns what it holds together with the ties that the non-responsive nodes imply, which is nearly always the larger part.

Size and speed

A 'stocnet' object is usually smaller in memory than the same network as an 'igraph' or 'tbl_graph' object. A network of a thousand nodes and four thousand ties takes about half the memory as a 'stocnet', whether it is sparse or dense, directed or not, and about four fifths where its nodes are named and carry attributes, since names and attributes take the same room in any class. Only a very small network, or one with no ties at all, is a little larger as a 'stocnet', since the list has a few kilobytes of its own. A matrix is by far the largest for any network that is not both small and dense, since it has a cell for every pair of nodes.

What a 'stocnet' saves in memory it can cost in time, where a function works on another class. Many functions here, and in the packages that build on this one, calculate on an 'igraph' object or on a matrix, and coerce a 'stocnet' to one first, which takes a few milliseconds for a network of a thousand nodes. That is not noticeable in a single call, but adds up where a function is called many thousands of times, as in a simulation. There it is quicker to coerce once, with `as_igraph()` or `as_matrix()`, and to pass the result to the functions inside the loop.

Printing

When printed, 'stocnet' objects will print to the console any information stored about the names of the network, its modes, or layers. It will also describe key features of the network, such as whether the network is multiplex, weighted, directed, etc.

It will then print tibbles for the nodes, changes, and ties in the network, as appropriate. That is, if there is no nodal data (e.g. it is an unlabelled network without any other nodal attributes), then this will be skipped. Similarly, if no nodal changes are logged, this information will be skipped too.

Examples

```
out <- make_stocnet(info = list(name = "Example Network",
                          modes = c("Person", "Organization"),
                          layers = c("Friendship", "Collaboration")),
  nodes = data.frame(label = c("A", "B", "C"),
                     mode = c("Person", "Person", "Organization"),
                     active = c(TRUE, FALSE, TRUE)),
  ties = data.frame(from = c("A", "B"),
                    to = c("B", "C"),
                    weight = c(1, 2),
                    layer = c("Friendship", "Collaboration")),
  changes = data.frame(time = c(1, 2),
                       node = c("A", "B"),
                       var = c("active", "active"),
                       value = c(FALSE, TRUE)))
```

`make_write` *Making networks to external files*

Description

Researchers may want to save or work with networks outside R. The following functions offer ways to export to some common external file formats:

- `write_matrix()` exports an adjacency matrix to a .csv file.
- `write_edgelist()` exports an edgelist to a .csv file.
- `write_nodelist()` exports a nodelist to a .csv file.
- `write_pajek()` exports Pajek .net files.
- `write_ucinet()` exports a pair of UCINET files in V6404 file format (.##h, .##d).
- `write_dynetml()` exports DyNetML interchange format files.

- `write_graphml()` exports GraphML files.
- `write_gml()` exports GML files.
- `write_gdf()` exports GDF files.
- `write_gexf()` exports GEXF files, for example for use in Gephi.
- `write_hif()` exports a hypergraph as a Hypergraph Interchange Format (HIF) file, which the main hypergraph libraries read.
- `write_geojson()` exports a spatial network as a GeoJSON file, for example for use in a geographic information system.

Usage

```
write_matrix(.data, filename, ...)

write_edgelist(.data, filename, ...)

write_nodelist(.data, filename, ...)

write_pajek(.data, filename, ...)

write_ucinet(.data, filename, name)

write_graphml(.data, filename, ...)

write_gml(.data, filename, ...)

write_gdf(.data, filename, ...)

write_gexf(.data, filename, ...)

write_dynetml(.data, filename, ...)

write_geojson(.data, filename, ...)

write_hif(.data, filename, ...)
```

Arguments

`.data` An object of a {manynet}-consistent class:
- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`filename` Character string filename. If missing, the files will have the same name as the object and be saved to the working directory. An appropriate extension will be added if not included.

...	Additional parameters passed to the write function.
name	Character string to name the network internally, e.g. in UCINET. By default the name will be the same as the object.

Details

Note that these functions are not as actively maintained as others in the package, so please let us know if any are not currently working for you or if there are missing import routines by raising an issue on Github.

write_gexf() writes the node and tie attributes as declared attributes, except that 'x', 'y', 'z', 'size', and 'color' are written as the visualisation elements Gephi reads, and 'start' and 'end' are written as times, which makes the file dynamic. Only a hexadecimal colour is written as a visualisation element. A colour named some other way, such as "red", is written as an ordinary attribute, which reads back unchanged. Node names are written as labels, since node ids are required by the format and so are written as positions in the network.

write_hif() writes a hypergraph in the Hypergraph Interchange Format. A network that is not a hypergraph is written as one, as to_hypergraph() reformats it. Nodes and hyperedges are identified by their labels where they have them, and otherwise by their positions.

write_geojson() writes a spatial network (see is_spatial()), with each node as a point and each tie as a line between two nodes, and with their attributes as the properties of those features. Since a network does not hold the shape of a tie, each line is straight. A height the nodes have, as 'z', 'elevation', 'altitude', or 'alt', is written as the third coordinate of their positions, and so is read back as 'z'. The direction of a tie is not written.

Value

The write_functions export to different file formats, depending on the function.

A pair of UCINET files in V6404 file format (.##h, .##d)

Source

write_ucinet() kindly supplied by Christian Steglich, constructed on 18 June 2015.

See Also

as

Other makes: make_collect, make_create, make_ego, make_emails, make_explicit, make_groups, make_growth, make_learning, make_play, make_random, make_read

manip_changes *Manipulating changes to nodes over time*

Description

These functions offer ways to modify data held about how nodes change over time. They include:

- `bind_changes()` adds a table of changes to the nodes of a network.
- `mutate_changes()` can be used to update network changes.
- `filter_changes()` is used to subset network changes.
- `gather_changes()` is similar to `filter_changes()`, but collects the cumulative changes up to a time point.
- `apply_changes()` applies the changes collected up to a time point to a network, removing the changes.

An example of when this might be useful is to track change in the composition of a network (when nodes are present or absent over time), though the function can flexibly accommodate changes in other nodal attributes.

Usage

```
delete_changes(.data)

mutate_changes(.data, ...)

bind_changes(.data, changes, var, ...)

filter_changes(.data, ..., .by = NULL)

select_changes(.data, ..., .by = NULL)

arrange_changes(.data, ...)

rename_changes(.data, ...)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`...` Additional parameters and arguments passed on internally.

changes — A data frame containing the changes to be added. This should have columns `time`, `node` (labelled), `var` (referencing an existing nodal attribute), and `value` (which can be of any class).

var — A character string specifying the nodal variable to which the changes apply, if not already specified in the `changes` data frame. This is only used when the `changes` data frame does not already have a `var` column, and is ignored otherwise.

.by — An attribute name to join objects by. By default, NULL.

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods for these functions:

	data.frame	default	igraph	stocnet	tbl_graph
arrange_changes		*	*	*	
bind_changes		*	*	*	*
delete_changes		*	*	*	
filter_changes		*	*	*	
mutate_changes		*		*	*
rename_changes	*	*	*	*	
select_changes	*	*	*	*	

If a method is not available for a particular class, but a default method is, the default method will attempt to coerce the object to a class for which a method is defined, and then coerce the output back to the original class. If no method is available for any class, an error will be thrown.

Value

A data object of the same class as the function was given.

See Also

`to_time()`

Other changes: `mark_format_change`

Other manipulations: `manip_globals`, `manip_info`, `manip_nodes_attr`, `manip_nodes_num`, `manip_ties_attr`, `manip_ties_num`

Examples

```
filter_changes(fict_starwars, node == "Anakin")
select_changes(fict_starwars, node)
```

manip_globals *Manipulating global attributes*

Description

These functions offer ways to manipulate network-level data constants or variables that are not tied to a particular node or tie. They include:

- `bind_globals()` adds rows of global variables to the network.
- `mutate_globals()` changes the columns of the global variables table. Where the network holds no global variables yet, it creates the table. Where it holds one already, it changes that table's columns rather than adding to it, so use `bind_globals()` to add another global variable.
- `rename_globals()` renames columns in the global variables table.
- `select_globals()` selects columns in the global variables table.
- `filter_globals()` subsets rows of the global variables table.
- `arrange_globals()` orders rows of the global variables table.
- `delete_globals()` drops the global variables table from the network.

It expects three columns for the variable to which the change applies, which should be called 'var', the time of the change, which should be called 'time', and the new value to be applied, which should be called 'value'.

Usage

```
mutate_globals(.data, ...)

rename_globals(.data, ...)

select_globals(.data, ...)

bind_globals(.data, globals)

filter_globals(.data, ...)

arrange_globals(.data, ...)

delete_globals(.data)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package

- igraph `igraph`, from the `{igraph}` package
- network `network`, from the `{network}` package
- tidygraph `tbl_graph`, from the `{tidygraph}` package

`...` Additional parameters and arguments passed on internally.

`globals` A data frame of global variables to bind on. It should hold a column `var` naming the variable, a column `value` holding its value, and, where the variable changes over time, a column `time`.

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods for these functions:

	data.frame	default	igraph	stocnet
arrange_globals	*	*		*
bind_globals	*	*		*
delete_globals		*	*	*
filter_globals	*	*		*
mutate_globals		*		*
rename_globals	*	*		*
select_globals	*	*		*

If a method is not available for a particular class, but a default method is, the default method will attempt to coerce the object to a class for which a method is defined, and then coerce the output back to the original class. If no method is available for any class, an error will be thrown.

Value

A data object of the same class as the function was given.

See Also

`to_time()`

Other manipulations: `manip_changes`, `manip_info`, `manip_nodes_attr`, `manip_nodes_num`, `manip_ties_attr`, `manip_ties_num`

Examples

```
as_stocnet(ison_algebra) |>
   mutate_globals(time = 2, var = "active", value = FALSE)
as_stocnet(ison_algebra) |>
   bind_globals(data.frame(time = 1:2, var = "budget", value = c(10, 20)))
as_stocnet(ison_algebra) |>
   bind_globals(data.frame(time = 1:2, var = "budget", value = c(10, 20))) |>
   filter_globals(time == 1)
```

manip_info *Manipulating network information*

Description

These functions allow users to add and edit information about the network itself. Where available, this information is printed for tidygraph-class objects, and can be used for printing a grand table in the {grand} package.

- add_info() adds information attributes to the network.
- mutate_info() updates information attributes of the network.
- net_attributes() lists the information attributes of the network.

Usage

```
add_info(.data, ...)

mutate_info(.data, ...)

net_attributes(.data)
```

Arguments

.data An object of a {manynet}-consistent class:
- adjacency or incidence matrix from {base} R
- edgelist data.frame from {base} R or tbl/tbl_df from {tibble}
- stocnet stocnet, from the {manynet} package
- igraph igraph, from the {igraph} package
- network network, from the {network} package
- tidygraph tbl_graph, from the {tidygraph} package

... Named attributes. The following are currently recognised:
- "name" is the name of the network
- "modes" is the name(s) of the nodeset(s)
- "layers" is the name(s) of the tie type(s)
- "directed" is a logical vector indicating whether each layer is directed
- "signed" is a logical vector indicating whether each layer is signed, which to_signed() sets and to_unsigned() removes
- "source" is the source of the network ("empirical" or "synthetic")
- "method" is the method of data collection or model used (e.g. "survey", "interview","sensor","observation","archival", or "simulation")
- "location" is the geographic, institutional, or digital location of the network
- "date" is the date of data collection or model run
- "boundary" is the boundary specification of the network ("ego", "roster", or "snowball")

- "observation" is the observation type of the network ("cross-sectional", "panel", or "event")
- "update" is the update type of the network ("increment" or "replace")
- "max_degree" is the maximum degree of the network
- "min_degree" is the minimum degree of the network
- "doi" is the DOI or URL of the network
- "transformations" records how the network has been transformed since it was collected or generated. See the Transformations section. Note that this records what has been done to the network, where "method" records how the network was collected or modelled.

If no arguments are used, the function will check for missing information and prompt the user to add it. If `optional = TRUE` is specified, the function will also prompt for optional information.

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods for these functions:

```
            data.frame default igraph matrix network stocnet tbl_graph
add_info                     *      *                      *
as_infolist          *              *      *       *       *         *
mutate_info                  *      *                      *
```

If a method is not available for a particular class, but a default method is, the default method will attempt to coerce the object to a class for which a method is defined, and then coerce the output back to the original class. If no method is available for any class, an error will be thrown.

Value

A data object of the same class as the function was given.

Transformations

The "transformations" field implements section 4 of the GRAND guidelines, which names six ways raw data is turned into analytic data. It holds a named list, one element for each of them:

- "symmetrisation" (GRAND 4.1), which `to_undirected()` sets.
- "dichotomisation" (4.2), which `to_unweighted()` sets.
- "projection" (4.3), which `to_mode1()`, `to_mode2()`, and `to_concepts()` set.
- "exclusion" (4.4), which the functions that drop nodes or ties set, such as `delete_isolates()`, `to_component()`, and `to_uniplex()`.
- "aggregation" (4.5), which `to_aggregated()`, `to_flat()`, `join_ties()`, and `to_blockmodel()` set.
- "imputation" (4.6), which `impute_ties()` and `impute_nodes()` set.

One further name, "normalisation", records what `to_normalised()` does. The guidelines do not name it, since rescaling tie values neither dichotomises them (4.2 ends in an unweighted network) nor aggregates them (4.5 combines what was separate), but it changes the analytic network and so is recorded too. Nor do they name "disaggregation", which `to_disaggregated()` sets where it turns tie weights back into that many parallel ties.

A name that is absent means that transformation was not applied, so `"symmetrisation" %in% names(as_infolist(.data)$transformations)` answers whether a network was symmetrised without reading past everything else done to it. `describe_transformations()` reports the same thing as a phrase.

Each element is a character vector naming the method first, and, where the guidelines ask for an amount too, its consequence in parentheses: `list(symmetrisation = "collapse", imputation = "reciprocity (73 missing ties)")`. An element accumulates rather than replaces, so a network imputed in more than one step reports each of them, and the order of the names is the order the transformations were applied.

The `to_*()` and `impute_*()` functions set this themselves, so it rarely needs to be set by hand. Where it does, `add_info()` takes a named list and merges it in, and refuses a name that is not one of those above.

See Also

GRAND statement for more information on the Guidelines for Reporting About Network Data (GRAND).

Other manipulations: `manip_changes`, `manip_globals`, `manip_nodes_attr`, `manip_nodes_num`, `manip_ties_attr`, `manip_ties_num`

Examples

```
add_info(ison_algebra, name = "Algebra")
```

`manip_nodes_attr` *Manipulating node attributes*

Description

These functions allow users to add nodes attributes:

- `add_node_attribute()` offers an {igraph}-style way to add a vector of values to a network as a nodal attribute.
- `delete_node_attribute()` offers an {igraph}-style way to remove one or more named nodal attributes from a network.
- `mutate_nodes()` offers a {tidyverse}-style way to add one or more vectors of values to a network as nodal attributes.
- `rename_nodes()` offers a {tidyverse}-style way to rename nodal attributes.
- `select_nodes()` offers a {tidyverse}-style way to select a subset of nodal attributes.
- `join_nodes()` merges all nodal attributes from one network to another.

Note that while add_*()/delete_*() functions operate similarly as comparable {igraph} functions, mutate*(), bind*(), etc work like {tidyverse} or {dplyr}-style functions. A nodal attribute can equally be deleted the {tidyverse} way by assigning it NULL in mutate_nodes().

Usage

```
add_node_attribute(.data, attr_name, vector)

delete_node_attribute(.data, attr_name)

mutate(.data, ...)

mutate_nodes(.data, ...)

rename_nodes(.data, ...)

select_nodes(.data, ...)

join_nodes(
  .data,
  object2,
  .by = NULL,
  join_type = c("full", "left", "right", "inner")
)

bind_node_attributes(.data, object2)
```

Arguments

.data	An object of a {manynet}-consistent class: • adjacency or incidence matrix from {base} R • edgelist data.frame from {base} R or tbl/tbl_df from {tibble} • stocnet stocnet, from the {manynet} package • igraph igraph, from the {igraph} package • network network, from the {network} package • tidygraph tbl_graph, from the {tidygraph} package
attr_name	Character string naming a nodal attribute. The attribute itself may be a logical mark, numeric measure, or character membership vector.
vector	A vector of values for the new attribute.
...	Additional parameters and arguments passed on internally.
object2	A second object to copy nodes or ties from.
.by	An attribute name to join objects by. By default, NULL.
join_type	A type of join to be used. Options are "full","left", "right", "inner".

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods for these functions:

	data.frame	default	igraph	network	stocnet	tbl_graph
add_node_attribute		*	*	*	*	*
delete_node_attribute		*	*	*		*
join_nodes		*	*		*	
mutate_nodes		*	*	*	*	*
select_nodes	*	*	*		*	*

If a method is not available for a particular class, but a default method is, the default method will attempt to coerce the object to a class for which a method is defined, and then coerce the output back to the original class. If no method is available for any class, an error will be thrown.

Value

A data object of the same class as the function was given.

See Also

Other nodes: manip_nodes_num

Other manipulations: manip_changes, manip_globals, manip_info, manip_nodes_num, manip_ties_attr, manip_ties_num

Examples

```
other <- create_filled(4) |> mutate(name = c("A", "B", "C", "D"))
add_nodes(other, 4, list(name = c("Matthew", "Mark", "Luke", "Tim")))
other <- create_filled(4) |> mutate(name = c("A", "B", "C", "D"))
another <- create_filled(3) |> mutate(name = c("E", "F", "G"))
join_nodes(another, other)
```

---

manip_nodes_num *Manipulating number of nodes*

Description

These functions allow users to add and delete nodes:

- add_nodes() adds an additional number of nodes to network data.
- delete_nodes() deletes nodes from network data.
- delete_isolates() deletes all nodes without ties.
- delete_incomplete() deletes nodes with any missing attribute values, retaining only the complete cases. A node attribute that is missing for every node is not read as missing data, since in a dynamic network such an attribute is a placeholder for one whose values are all recorded as changes.

- `bind_nodes()` adds two nodesets together.
- `filter_nodes()` subsets nodes based on some nodal attribute-related logical statement.
- `arrange_nodes()` reorders nodes based on some nodal attribute.

While `add_*()`/`delete_*()` functions operate similarly as comparable {igraph} functions, `bind_*()` and `filter_*()` works like a {tidyverse} or {dplyr}-style function.

Usage

```
add_nodes(.data, nodes, attribute = NULL)

delete_nodes(.data, nodes)

delete_isolates(.data)

delete_incomplete(.data)

bind_nodes(.data, object2)

filter_nodes(.data, ..., .by = NULL)

arrange_nodes(.data, ...)
```

Arguments

`.data`	An object of a {manynet}-consistent class: • adjacency or incidence `matrix` from {base} R • edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble} • stocnet `stocnet`, from the {manynet} package • igraph `igraph`, from the {igraph} package • network `network`, from the {network} package • tidygraph `tbl_graph`, from the {tidygraph} package
`nodes`	The number of nodes to be added.
`attribute`	A named list to be added as tie or node attributes.
`object2`	A second object to copy nodes or ties from.
`...`	Additional parameters and arguments passed on internally.
`.by`	An attribute name to join objects by. By default, NULL.

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods for these functions:

	data.frame	default	igraph	list	matrix	network	stocnet
add_nodes		*	*			*	
arrange_nodes		*					*

```
bind_nodes                          *                                   *
delete_incomplete                   *                                   *
delete_isolates            *        *      *    *      *       *       *
delete_nodes                        *      *                   *       *
filter_nodes                        *                                  *
                   tbl_graph
add_nodes                 *
arrange_nodes             *
bind_nodes
delete_incomplete         *
delete_isolates           *
delete_nodes              *
filter_nodes              *
```

If a method is not available for a particular class, but a default method is, the default method will attempt to coerce the object to a class for which a method is defined, and then coerce the output back to the original class. If no method is available for any class, an error will be thrown.

Value

A data object of the same class as the function was given.

See Also

Other nodes: manip_nodes_attr

Other manipulations: manip_changes, manip_globals, manip_info, manip_nodes_attr, manip_ties_attr, manip_ties_num

Examples

```
  other <- create_filled(4) |> mutate(name = c("A", "B", "C", "D"))
  add_nodes(other, 4, list(name = c("Matthew", "Mark", "Luke", "Tim")))
ison_adolescents |>
  mutate_ties(wave = sample(1995:1998, 10, replace = TRUE)) |>
  to_waves(attribute = "wave") |>
  delete_isolates()
```

---

manip_ties_attr *Manipulating tie attributes*

Description

These functions allow users to add and delete tie attributes:

- add_tie_attribute() and mutate_ties() offer ways to add a vector of values to a network as a tie attribute.
- delete_tie_attribute() offers an {igraph}-style way to remove one or more named tie attributes.

- `rename_ties()` renames tie attributes.
- `join_ties()` merges ties from two networks, adding a tie attribute identifying the newly added ties.

Note that while `add_*()`/`delete_*()` functions operate similarly as comparable {igraph} functions, `mutate*()`, `bind*()`, etc work like {tidyverse} or {dplyr}-style functions. A tie attribute can equally be deleted the {tidyverse} way by assigning it `NULL` in `mutate_ties()`.

Usage

```
add_tie_attribute(.data, attr_name, vector)

delete_tie_attribute(.data, attr_name)

mutate_ties(.data, ...)

rename_ties(.data, ...)

arrange_ties(.data, ...)

join_ties(.data, object2, attr_name)

select_ties(.data, ...)

summarise_ties(.data, ...)
```

Arguments

`.data`	An object of a {manynet}-consistent class: • adjacency or incidence `matrix` from {base} R • edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble} • stocnet `stocnet`, from the {manynet} package • igraph `igraph`, from the {igraph} package • network `network`, from the {network} package • tidygraph `tbl_graph`, from the {tidygraph} package
`attr_name`	Character string naming a nodal attribute. The attribute itself may be a logical mark, numeric measure, or character membership vector.
`vector`	A vector of values for the new attribute.
`...`	Additional parameters and arguments passed on internally.
`object2`	A second object to copy nodes or ties from.

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods for these functions:

	data.frame	default	igraph	stocnet	tbl_graph
add_tie_attribute	*	*	*	*	
arrange_ties		*		*	*
delete_tie_attribute	*	*	*		
mutate_ties		*		*	*
rename_ties	*	*		*	*
select_ties	*	*		*	*
summarise_ties		*			*

If a method is not available for a particular class, but a default method is, the default method will attempt to coerce the object to a class for which a method is defined, and then coerce the output back to the original class. If no method is available for any class, an error will be thrown.

Value

A data object of the same class as the function was given.

See Also

Other ties: manip_ties_num, modif_backbone, modif_direction, modif_weight

Other manipulations: manip_changes, manip_globals, manip_info, manip_nodes_attr, manip_nodes_num, manip_ties_num

Examples

```
other <- create_filled(4) |> mutate(name = c("A", "B", "C", "D"))
mutate_ties(other, form = 1:6) |> filter_ties(form < 4)
add_tie_attribute(other, "weight", c(1, 2, 2, 2, 1, 2))
```

---

manip_ties_num *Manipulating number of ties*

Description

These functions allow users to add and delete ties:

- add_ties() adds additional ties to network data
- delete_ties() deletes ties from network data
- bind_ties() appends the tie data from two networks
- filter_ties() subsets ties based on some tie attribute-related logical statement.

While add_*()/delete_*() functions operate similarly as comparable {igraph} functions, filter*(), etc work like {tidyverse} or {dplyr}-style functions.

Usage

```
add_ties(.data, ties, attr_list = NULL)

delete_ties(.data, ties)

bind_ties(.data, ...)

filter_ties(.data, ...)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`ties` The ties to add. Several forms are accepted:

- a single number, e.g. `3`, in which case that number of ties is added at random among those dyads not already tied (respecting whether the network is directed or two-mode)
- an even vector of node names or indices, e.g. `c("Betty","Tina")`, interpreted pairwise as sender and receiver
- a two-column matrix, edgelist, or data frame of node names or indices, each row of which is interpreted as a tie
- an explicit tie formula in the same syntax as `create_explicit()`, e.g. `Betty -+ Tina` or `1 -+ 3` for an arc from the first to the third node, `Betty +-+ Tina` or `1 ++ 3` for both arcs, and `Betty -- Tina` or `1 -- 3` for a tie between them. Note that both ends of the tie operator must be marked with `-` or `+`, so that `1-3` remains arithmetic while `1--3` is a tie. Several ties can be added at once by wrapping them in `c()`, e.g. `c(Betty -+ Tina, Sue -+ Pam)`, and node sets can be linked using `:`, e.g. `Betty:Sue -+ Tina`. Such formulae can also be passed as one-sided formulas, e.g. `~ Betty -+ Tina`, which is useful when passing them around programmatically.

Note that in a directed network an undirected tie operator, like an even vector, adds a single arc from the first to the second node.

`attr_list` A list of attributes to be added to the new ties. Where the network is weighted but no weight is given here, the new ties are given a weight of 1.

`...` Additional parameters and arguments passed on internally.

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods for these functions:

```
             default igraph network stocnet tbl_graph
add_ties           *      *       *                 *
bind_ties          *                      *         *
delete_ties        *      *       *       *         *
filter_ties        *      *               *         *
```

If a method is not available for a particular class, but a default method is, the default method will attempt to coerce the object to a class for which a method is defined, and then coerce the output back to the original class. If no method is available for any class, an error will be thrown.

Value

A data object of the same class as the function was given.

See Also

Other ties: manip_ties_attr, modif_backbone, modif_direction, modif_weight

Other manipulations: manip_changes, manip_globals, manip_info, manip_nodes_attr, manip_nodes_num, manip_ties_attr

Examples

```
  other <- create_filled(4) |> mutate(name = c("A", "B", "C", "D"))
  mutate_ties(other, form = 1:6) |> filter_ties(form < 4)
  add_tie_attribute(other, "weight", c(1, 2, 2, 2, 1, 2))
ison_adolescents |> add_ties(c("Betty","Tina"))
ison_adolescents |> add_ties(Betty -+ Tina)
ison_adolescents |> add_ties(c(Betty -+ Tina, Sue -+ Pam))
ison_adolescents |> add_ties(3)
delete_ties(ison_adolescents, 3)
delete_ties(ison_adolescents, "Alice|Sue")
```

---

mark_features *Marking networks features*

Description

These functions implement logical tests for various network features.

- `is_connected()` tests whether network is connected, by default in the strong sense where ties are directed. Pass `connectivity = "weak"` to ignore tie direction.
- `is_perfect_matching()` tests whether there is a matching for a network that covers every node in the network.
- `is_eulerian()` tests whether there is a Eulerian path for a network where that path passes through every tie exactly once.
- `is_acyclic()` tests whether network is a directed acyclic graph.
- `is_aperiodic()` tests whether network is aperiodic.

Usage

```
is_connected(.data, connectivity = c("strong", "weak"))

is_perfect_matching(.data, mark = "type")

is_eulerian(.data)

is_acyclic(.data)

is_aperiodic(.data, max_path_length = 4)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`connectivity` Character string, "weak" treats a directed network's components as if the network were undirected, and "strong" requires ties in both directions between members. This is ignored for undirected networks, where the two notions coincide. Note that the default differs by function: marks that assert connectedness default to "strong", while functions that scope or split a network into components default to "weak".

`mark` A logical vector marking two types or modes. By default "type".

`max_path_length` Maximum path length considered. If negative, paths of all lengths are considered. By default 4, to avoid potentially very long computation times.

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods for these functions:

```
                    default igraph
is_acyclic                *      *
is_aperiodic              *      *
is_connected              *      *
is_eulerian               *      *
is_perfect_matching       *      *
```

If a method is not available for a particular class, but a default method is, the default method will attempt to coerce the object to a class for which a method is defined, and then coerce the output back to the original class. If no method is available for any class, an error will be thrown.

Value

TRUE if the condition is met, or FALSE otherwise.

is_perfect_matching

For two-mode or bipartite networks, `to_matching()` is used to identify whether a perfect matching is possible. For one-mode networks, we use the Tutte theorem. Note that currently only subgraphs with cutpoints removed are tested, and not all possible subgraphs. This is to avoid computationally expensive combinatorial operations, but may come at the cost of some edge cases where a one-mode network cannot perfectly match as suggested.

Aperiodicity

Aperiodicity is a property of directed networks that can be interpreted as the absence of cycles of a common length. Aperiodicity is a necessary condition for the existence of a unique stationary distribution in Markov chains, and thus is an important property for the analysis of dynamic processes on networks. The function `is_aperiodic()` tests for aperiodicity by finding all simple paths from each node back to itself, and then calculating the greatest common divisor of the lengths of these paths. If the greatest common divisor is 1, the network is aperiodic.

Source

https://stackoverflow.com/questions/55091438/r-igraph-find-all-cycles

References

On perfect matching:

Tutte, William T. 1950. "The factorization of locally finite graphs". *Canadian Journal of Mathematics*. 2: 44–49. doi:10.4153/cjm19500052

On Eulerian cycles and paths:

Euler, Leonard. 1736. "Solutio problematis ad geometriam situs pertinentis". *Comment. Academiae Sci. I. Petropolitanae* 8: 128–140.

Hierholzer, Carl. 1873. "Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren". *Mathematische Annalen*, 6(1): 30–32. doi:10.1007/BF01442866

On aperiodicity:

Jarvis, J.P, and D.R. Shier. 1996. "Graph-theoretic analysis of finite Markov chains", in Shier, D.R., Wallenius, K.T. (eds) *Applied Mathematical Modeling: A Multidisciplinary Approach*. CRC Press.

See Also

Other marking: `mark_is`

Examples

```
is_connected(ison_southern_women)
is_connected(ison_algebra, connectivity = "weak")
is_perfect_matching(ison_southern_women)
is_eulerian(ison_brandes)
is_acyclic(ison_algebra)
is_aperiodic(ison_algebra)
```

`mark_format_change` *Marking networks change formats*

Description

These functions implement logical tests for various network properties. All `is_*()` functions return a logical scalar (TRUE or FALSE).

- `is_longitudinal()` marks networks TRUE if they are panels, i.e. if each moment they record re-observes the whole network.
- `is_dynamic()` marks networks TRUE if they record a stream of events, either as increments to the ties or as the interval each tie lasts over.

A network records time in one way or the other, so the two marks do not overlap. See the Time section of `to_time()` for what separates them.

- `is_changing()` marks networks TRUE if they contain any nodal changes.

Usage

```
is_longitudinal(.data)

is_dynamic(.data)

is_changing(.data)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods for these functions:

```
                default diff_model igraph list stocnet
is_changing           *          *      *            *
is_dynamic            *                 *            *
is_longitudinal       *                 *    *       *
```

If a method is not available for a particular class, but a default method is, the default method will attempt to coerce the object to a class for which a method is defined, and then coerce the output back to the original class. If no method is available for any class, an error will be thrown.

See Also

Other marks: `mark_format_cognitive`, `mark_format_node`, `mark_format_tie`

Other changes: `manip_changes`

Examples

```
is_longitudinal(create_tree(5, 3))
is_dynamic(create_tree(3))
is_changing(fict_starwars)
```

`mark_format_cognitive` *Marking networks cognitive formats*

Description

These functions implement logical tests for networks whose ties name a third node as well as the two nodes each tie runs between. All `is_*()` functions return a logical scalar (TRUE or FALSE).

- `is_cognitive()` marks networks TRUE if they are cognitive social structures, i.e. where the edgelist contains a 'by' column naming the node that reported each tie, and all of them report on one roster of nodes.
- `is_egocentric()` marks networks TRUE if the edgelist contains a 'by' column naming the ego that reported each tie, but each ego reports on alters of its own.
- `is_gossip()` marks networks TRUE if the edgelist contains an 'about' column naming the node each tie is about, as where a sender tells a receiver about a target.

Usage

```
is_cognitive(.data)

is_egocentric(.data)

is_gossip(.data)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

Details

A cognitive social structure (Krackhardt 1987) asks each respondent about every pair of nodes in one roster, so each report is a network of the same nodes. An egocentric design instead asks each ego to name alters of its own, as a name generator does, so the egos' reports share no nodes. Both record the reporter in a 'by' column. They are told apart by the network's 'observation' information, "cognitive" or "egocentric", and where that is not given, by whether any two reports share a node. Note that `is_egolist()` marks something else: a list of networks, each of one ego and its ties.

A 'by' or 'about' column that holds only `NA` names no node, so a network whose other layers hold one is not marked.

A three-dimensional array cannot say what its third dimension holds, which may be reporters, targets, waves, or layers. `is_cognitive()` marks an array TRUE where that dimension could hold one reporter for each node, and `is_egocentric()` and `is_gossip()` do not mark arrays. `as_stocnet()` asks which of these an array holds.

References

Krackhardt, David. 1987. "Cognitive social structures". *Social Networks* 9(2): 109-134. doi:10.1016/03788733(87)900098

See Also

Other marks: `mark_format_change`, `mark_format_node`, `mark_format_tie`

Examples

```
is_cognitive(create_filled(3))
is_egocentric(create_filled(3))
is_gossip(create_filled(3))
```

mark_format_node *Marking networks nodal formats*

Description

These functions implement logical tests for various network properties. All `is_*()` functions return a logical scalar (TRUE or FALSE).

- `is_twomode()` marks networks TRUE if they contain two sets of nodes.
- `is_multilevel()` marks networks TRUE if they contain two or more levels of nodes that are tied both within and between levels.
- `is_hypergraph()` marks networks TRUE if they hold hyperedges, which are ties among any number of nodes.
- `is_labelled()` marks networks TRUE if there is a 'names' attribute for the nodes.
- `is_attributed()` marks networks TRUE if there are other nodal attributes than 'names' or 'type'.
- `is_spatial()` marks networks TRUE if the nodes have positions, that is a pair of numeric nodal attributes called 'x' and 'y', or 'lat' and 'lon' ('long', 'lng'), or 'latitude' and 'longitude'. A position may also have a height, called 'z', 'elevation', 'altitude', or 'alt', though it takes no height to be spatial. Note that this is all a network records of a position, so a network read with the positions of a layout, such as one exported from Gephi, is marked spatial too.
- `is_egolist()` marks TRUE a list of networks, one for each node of a network, as `to_egos()` returns. Unlike the other marks here, it marks a list and not a network, so a single network is always FALSE.

Usage

```
is_twomode(.data)

is_multilevel(.data)

is_hypergraph(.data)

is_labelled(.data)

is_attributed(.data)

is_spatial(.data)

is_egolist(.data)
```

Arguments

`.data` An object of a `{manynet}`-consistent class:

- adjacency or incidence `matrix` from `{base}` R

- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods for these functions:

```
              data.frame default igraph list matrix network numeric stocnet
is_attributed                  *
is_egolist                     *
is_labelled            *       *      *    *      *       *              *
is_multilevel                  *      *    *                             *
is_spatial                     *
is_twomode             *       *      *    *      *       *       *      *
              tbl_graph
is_attributed
is_egolist
is_labelled           *
is_multilevel         *
is_spatial
is_twomode            *
```

If a method is not available for a particular class, but a default method is, the default method will attempt to coerce the object to a class for which a method is defined, and then coerce the output back to the original class. If no method is available for any class, an error will be thrown.

A multilevel network is one in which the nodes belong to two or more levels, or nodesets, that are tied not only to each other but also among themselves. `fict_marvel`, for instance, interlocks a one-mode layer of ties among its characters with a two-mode layer of affiliations between those characters and their teams. Such networks are distinguished from plain two-mode networks, such as `ison_southern_women`, in which ties run only between the two nodesets and never within them.

See Also

Other marks: `mark_format_change`, `mark_format_cognitive`, `mark_format_tie`

Examples

```
is_twomode(create_filled(c(2,2)))
is_multilevel(fict_marvel)
is_multilevel(ison_southern_women)
is_hypergraph(create_empty(3))
is_labelled(create_empty(3))
is_attributed(ison_algebra)
is_spatial(ison_algebra)
is_spatial(mutate_nodes(create_ring(4), x = c(0, 1, 1, 0), y = c(0, 0, 1, 1)))
is_egolist(to_egos(ison_adolescents))
```

mark_format_tie *Marking networks tie formats*

Description

These functions implement logical tests for various network properties. All `is_*()` functions return a logical scalar (TRUE or FALSE).

- `is_twomode()` marks networks TRUE if they contain two sets of nodes.
- `is_weighted()` marks networks TRUE if they contain tie weights. Note that signed networks often hold their signs as weights of -1 and 1, so that no sign is lost when coercing between formats; since such a 'weight' records only the sign of each tie, these networks are marked FALSE unless the weights vary in magnitude.
- `is_directed()` marks networks TRUE if the ties specify which node is the sender and which the receiver. A two-mode network can be directed too, for example where its ties run from speakers to the concepts they claim. A two-mode matrix cannot record direction, however, so it is always marked FALSE, and an edgelist whose two-mode ties all run one way is marked FALSE too, since it cannot say otherwise.
- `is_labelled()` marks networks TRUE if there is a 'names' attribute for the nodes.
- `is_attributed()` marks networks TRUE if there are other nodal attributes than 'names' or 'type'.
- `is_signed()` marks networks TRUE if the ties can be either positive or negative. This is the case where the ties have a 'sign' attribute, and also where they are weighted and any of those weights are negative.
- `is_complex()` marks networks TRUE if any ties are loops, with the sender and receiver being the same node.
- `is_multiplex()` marks networks TRUE if it contains multiple types of ties, such that there can be multiple ties between the same sender and receiver.
- `is_uniplex()` marks networks TRUE if it is neither complex nor multiplex.
- `is_disaggregated()` marks networks TRUE if any of their ties are parallel, as `tie_is_parallel()` marks them, so that `to_aggregated(over = NULL)` can combine them. A network marked FALSE holds no parallel ties, which does not mean that it was aggregated.

Usage

```
is_weighted(.data)

is_directed(.data)

is_signed(.data)

is_complex(.data)

is_multiplex(.data)
```

```
is_disaggregated(.data)

is_uniplex(.data)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods for these functions:

```
                  data.frame default igraph list matrix network stocnet
is_complex                 *       *      *    *      *       *       *
is_directed                *       *      *           *       *       *
is_disaggregated                   *
is_multiplex               *       *      *           *       *       *
is_signed                  *       *      *           *       *       *
is_uniplex                         *      *
is_weighted                *       *      *           *       *       *
layer_is_directed                  *      *                           *
                  tbl_graph
is_complex                *
is_directed               *
is_disaggregated
is_multiplex              *
is_signed                 *
is_uniplex
is_weighted               *
layer_is_directed
```

If a method is not available for a particular class, but a default method is, the default method will attempt to coerce the object to a class for which a method is defined, and then coerce the output back to the original class. If no method is available for any class, an error will be thrown.

See Also

Other marks: `mark_format_change`, `mark_format_cognitive`, `mark_format_node`

Examples

```
is_weighted(create_tree(3))
is_directed(create_tree(2))
is_directed(create_tree(2, directed = TRUE))
is_signed(create_lattice(3))
is_complex(create_lattice(4))
is_multiplex(create_filled(c(3,3)))
is_disaggregated(ison_koenigsberg)
is_uniplex(create_star(3))
```

`mark_is` *Marking networks classes*

Description

These functions implement logical tests for networks' classes.

- `is_manynet()` marks a network TRUE if it is compatible with {manynet} functions.
- `is_edgelist()` marks a network TRUE if it is an edgelist.
- `is_graph()` marks a network TRUE if it contains graph-level information.
- `is_list()` marks a network TRUE if it is a (non-igraph) list of networks, for example a set of ego networks or a dynamic or longitudinal set of networks.
- `is_longitudinal()` marks a network TRUE if it contains longitudinal, panel data.
- `is_dynamic()` marks a network TRUE if it contains dynamic, time-stamped data.
- `is_changing()` marks a network TRUE if it contains changes to nodal attributes.

All `is_*()` functions return a logical scalar (TRUE or FALSE).

Usage

```
is_manynet(.data)

is_graph(.data)

is_edgelist(.data)

is_list(.data)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods for these functions:

```
            data.frame default igraph list matrix network stocnet tbl_graph
is_edgelist          *       *
is_graph                     *      *                  *       *         *
is_list                      *           *
is_manynet           *       *      *           *       *       *         *
```

If a method is not available for a particular class, but a default method is, the default method will attempt to coerce the object to a class for which a method is defined, and then coerce the output back to the original class. If no method is available for any class, an error will be thrown.

Value

TRUE if the condition is met, or FALSE otherwise.

See Also

Other marking: `mark_features`

Examples

```
is_manynet(create_filled(2))
is_graph(create_star(2))
is_edgelist(matrix(c(2,2), 1, 2))
is_edgelist(as_edgelist(matrix(c(2,2), 1, 2)))
```

---

measure_attributes_nodes

Describing attributes of nodes in a network

Description

These functions extract certain attributes from network data:

- `node_attribute()` returns an attribute's values for the nodes in a network.
- `node_labels()` returns the names of the nodes in a network.
- `node_is_mode()` returns the mode of the nodes in a network.

These functions are also often used as helpers within other functions. `node_*()` always return vectors the same length as the number of nodes in the network.

Usage

```
node_attribute(.data, attr_name)

node_labels(.data)

node_names(.data)

node_is_mode(.data)
```

Arguments

`.data`	An object of a {manynet}-consistent class: • adjacency or incidence `matrix` from {base} R • edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble} • stocnet `stocnet`, from the {manynet} package • igraph `igraph`, from the {igraph} package • network `network`, from the {network} package • tidygraph `tbl_graph`, from the {tidygraph} package
`attr_name`	Character string naming a nodal attribute. The attribute itself may be a logical mark, numeric measure, or character membership vector.

See Also

Other measures: `measure_attributes_ties`, `measure_dims`

Examples

```
node_attribute(fict_lotr, "Race")
node_labels(ison_southern_women)
node_is_mode(ison_southern_women)
```

---

measure_attributes_ties

Describing attributes of ties in a network

Description

These functions extract certain attributes from network data:

- `tie_attribute()` returns an attribute's values for the ties in a network.
- `tie_weights()` returns the weights of the ties in a network.
- `tie_signs()` returns the signs of the ties in a network.
- `tie_is_twomode()` returns whether each tie in a network is a cross-mode tie.
- `tie_is_parallel()` returns whether each tie in a network runs parallel to another, i.e. whether another tie joins the same pair of nodes at the same moment.

- `tie_is_backbone()` returns whether each tie in a network is retained by a backbone filter, i.e. whether it carries more weight, or holds more structure, than a null model local to its endpoints expects.

These functions are also often used as helpers within other functions. `tie_*()` always return vectors the same length as the number of ties in the network, respectively.

Usage

```
tie_attribute(.data, attr_name)

tie_weights(.data)

tie_signs(.data)

tie_is_twomode(.data)

tie_is_parallel(.data)

tie_is_backbone(
  .data,
  filter = NULL,
  threshold = NULL,
  endpoints = c("either", "both")
)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`attr_name` Character string naming a nodal attribute. The attribute itself may be a logical mark, numeric measure, or character membership vector.

`filter` Which backbone filter to apply, one of:

- "disparity": the disparity filter of Serrano et al. (2009), which tests a tie's share of its node's strength against a null model in which that strength is divided at random. Assumes heavy-tailed weights.
- "lans": locally adaptive network sparsification, Foti et al. (2011), which replaces that null model with the empirical distribution of the node's own tie weights, and so assumes nothing about their shape.
- "noise": the noise-corrected filter of Coscia and Neffke (2017), which tests a tie's weight against the strengths of both its endpoints and allows for the uncertainty of the null itself. Corrects the disparity filter's bias towards hubs.

- "mlf": the marginal likelihood filter of Dianati (2016), which reads the weights as counts of independent events and tests each against a maximum-entropy null. Requires whole-number weights.
- "simmelian": the Simmelian backbone of Nick et al. (2013), which ranks each node's neighbours by how embedded the tie is and retains the ties whose endpoints rank each other similarly. Ignores tie weights, and so is the only filter available for an unweighted network. Reads a directed network as an undirected one, since it counts shared neighbours.

By default `NULL`, which uses "lans" where the network is weighted and "simmelian" where it is not. "lans" leads because it assumes nothing about the shape of the weights, and because it retains each node's strongest tie whatever that shape is. "disparity" is the better known filter, but its null model expects heavy-tailed weights: where weights are more even, a tie's share of its node's strength approaches 1/k for every tie, the p-value approaches 1/e, and the filter retains nothing at all.

`threshold` The cutoff below which a tie is retained. For the four statistical filters this is a significance level, by default 0.05. For "simmelian" it is instead one minus the similarity of the two endpoints' rankings, by default 0.5. By default `NULL`, which uses whichever of these the filter calls for.

`endpoints` Whether a tie is retained where it passes the filter at "either" of its endpoints, by default, or only where it passes at "both". "both" is the more demanding, since a tie must be locally salient to both the sender and the receiver. Applies to the "disparity" and "lans" filters only, since the others weigh both endpoints at once.

Parallel ties

Parallel ties, also called multi-edges, are two or more ties that join the same pair of nodes at the same moment. Ties that join the same pair of nodes at different moments follow one another rather than run alongside one another, and so are not parallel. How a network records time therefore decides which ties coexist:

- Where a network is a panel, ties are parallel where they share a wave.
- Where a network records a stream of events, ties are parallel where they share a moment.
- Where a network records the interval each tie lasts over, ties are parallel where those intervals overlap. Intervals that merely abut, one beginning as the other ends, do not.
- Where a network records no time at all, any two ties on a pair of nodes are parallel.

Ties of different types are likewise not parallel. Several types of tie between a pair of nodes is what `is_multiplex()` marks; `tie_is_parallel()` marks several ties of one type. Nor are reports of a tie by two reporters in a cognitive social structure, or gossip about two different targets, since each is a record of its own.

Every tie in such a bundle is marked, and not just the repetitions, so `sum(tie_is_parallel(ison_koenigsberg))` counts four of the seven bridges and not two.

Note that `as_matrix()` reports how many ties join each pair of nodes, so a network with parallel ties gives a matrix with cells greater than one even where it is neither weighted nor signed.

See Also

to_backbone(), which deletes the ties this does not mark, and which documents each filter and the works they come from.

Other measures: measure_attributes_nodes, measure_dims

Examples

```
tie_attribute(ison_algebra, "task_tie")
tie_weights(to_mode1(ison_southern_women))
tie_signs(to_uniplex(fict_marvel,"relationship"))
tie_is_twomode(fict_actually)
tie_is_parallel(ison_koenigsberg)
tie_is_backbone(ison_networkers)
```

---

measure_dims *Describing network dimensions*

Description

These functions extract certain attributes from given network data:

- net_nodes() returns the total number of nodes (of any mode) in a network.
- net_ties() returns the number of ties in a network.
- mode_nodes() returns the dimensions of a network in a vector as long as the number of modes in the network.
- net_layers() returns the number of layers in a multiplex network.
- layer_ties() returns the number of ties in a vector as long as the number of layers in the network.
- net_waves() returns the number of waves a panel network records, see is_longitudinal(). A network that is not a panel has one wave.
- net_times() returns the number of distinct moments a network records, however it records them: the waves of a panel, the events of a dynamic network, or the moments an interval network begins and ends a tie at. See the Time section of to_time().

These functions are also often used as helpers within other functions.

Usage

```
net_nodes(.data)

net_modes(.data)

net_ties(.data)

net_layers(.data)
```

```
layer_ties(.data)

net_waves(.data)

net_times(.data)

mode_nodes(.data)

net_dims(.data)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

Value

`net_*()` functions always relate to the overall graph or network, usually returning a scalar. `mode_nodes()` returns an integer of the number of nodes in a one-mode network, or one integer per mode (in `mode_names()` order) in the case of a two-mode, three-mode, or other multimodal network. `layer_ties()` returns an integer of the number of ties in a single-layer network, or one integer per layer (in `layer_names()` order) in the case of a multiplex network.

See Also

Other measures: `measure_attributes_nodes`, `measure_attributes_ties`

Examples

```
net_nodes(ison_southern_women)
net_modes(ison_southern_women)
net_ties(ison_southern_women)
net_layers(ison_southern_women)
layer_ties(fict_marvel)
net_waves(ison_monks)
net_times(irps_wwi)
mode_nodes(ison_southern_women)
mode_nodes(to_mode1(ison_southern_women))
```

measure_missingness *Describing network missingness*

Description

These functions describe the missingness in network data:

- `net_node_missing()` returns the proportion of nodes that are missing in a network.
- `net_node_incomplete()` returns the proportion of the network's node attribute values that are unknown.
- `net_tie_missing()` returns the proportion of ties that are missing in a network.
- `net_tie_incomplete()` returns the proportion of the network's ties whose value is unknown.

A network is *missing* a tie where the tie itself was not observed, so that whether it exists is not known. A tie or a node is *incomplete* where it is there and observed, but an attribute of it is not known. A weight of `NA` therefore marks an incomplete tie and not a missing one. `impute_ties()` and `impute_nodes()` impute each of these states.

A tie recorded as missing is one that could have been observed and was not. It is not a tie, so `net_ties()` does not count it, and it is not the absence of a tie either. See `as_missinglist()` for how each class records them, and `make_stocnet()` for how they differ from a node's absence and from a tie of unknown value.

For a multiplex or longitudinal network, `net_tie_missing()` counts the ties that could have been observed over each layer and each moment the network records. Coercing such a network to a matrix first gives a higher proportion, since a matrix holds only one cell for each dyad.

Usage

```
net_node_missing(.data)

net_tie_missing(.data)

net_node_incomplete(.data)

net_tie_incomplete(.data)
```

Arguments

`.data` An object of a {`manynet`}-consistent class:

- adjacency or incidence `matrix` from {`base`} R
- edgelist `data.frame` from {`base`} R or `tbl/tbl_df` from {`tibble`}
- stocnet `stocnet`, from the {`manynet`} package
- igraph `igraph`, from the {`igraph`} package
- network `network`, from the {`network`} package
- tidygraph `tbl_graph`, from the {`tidygraph`} package

Value

net_node_missing(), net_tie_missing(), net_node_incomplete(), and net_tie_incomplete() return a scalar.

Examples

```
net_node_incomplete(fict_lotr)
net_tie_incomplete(ison_adolescents)
```

---

member_names *Describing network names*

Description

These functions extract certain attributes from given network data:

- net_name() returns the name of the network, if it has one.
- mode_names() returns a vector of the names of the modes in a network, if they have been defined.
- net_node_attributes() returns a vector of nodal attributes in a network.
- layer_names() returns a vector of the names of the layers in a network, if they have been defined.
- layer_is_directed() returns whether each layer of a network is directed, named by layer.
- net_tie_attributes() returns a vector of tie attributes in a network.

These functions are also often used as helpers within other functions.

Usage

```
net_name(.data, prefix = NULL)

mode_names(.data)

net_node_attributes(.data)

layer_names(.data)

layer_is_directed(.data, layer = NULL)

net_tie_attributes(.data)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`prefix` An optional string to be added before the name of the network.

`layer` An optional character string naming one of the layers, one of those returned by `layer_names()`. Where a layer is named, a single value is returned for that layer; otherwise a value is returned for every layer, named by layer.

Details

A network can be directed in one layer and undirected in another, as a network of interstate trade and of state membership in intergovernmental organisations is. `is_directed()` marks such a network TRUE, since it holds arcs, and `layer_is_directed()` says which of its layers those arcs are in. Where a network records nothing about a layer, the network's own direction is reported for it.

Value

`net_*()` functions always relate to the overall graph or network, usually returning a scalar. `net_*_attributes()` returns a string vector with the names of all node or tie attributes in the network.

Examples

```
net_name(ison_southern_women)
  mode_names(ison_algebra)
  net_node_attributes(fict_lotr)
  layer_names(ison_algebra)
  layer_is_directed(ison_algebra)
  net_tie_attributes(ison_algebra)
```

`modif_backbone` *Modifying networks to their backbone*

Description

These functions reduce a network to its backbone, the ties that carry more weight, or hold more structure, than a null model expects:

- `to_backbone()` returns the network with only its backbone ties.
- `tie_is_backbone()` marks which ties are in the backbone.

A global cutoff such as `to_unweighted()` compares every tie against the same number, so it deletes a network's whole periphery before it touches a hub. A backbone filter compares each tie against a null model local to its endpoints, so a tie that is weak overall but strong for its own node is retained. This makes the multiscale structure of a weighted network visible, both for analysis and for drawing.

Usage

```
to_backbone(
  .data,
  filter = NULL,
  threshold = NULL,
  endpoints = c("either", "both")
)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`filter` Which backbone filter to apply, one of:

- "disparity": the disparity filter of Serrano et al. (2009), which tests a tie's share of its node's strength against a null model in which that strength is divided at random. Assumes heavy-tailed weights.
- "lans": locally adaptive network sparsification, Foti et al. (2011), which replaces that null model with the empirical distribution of the node's own tie weights, and so assumes nothing about their shape.
- "noise": the noise-corrected filter of Coscia and Neffke (2017), which tests a tie's weight against the strengths of both its endpoints and allows for the uncertainty of the null itself. Corrects the disparity filter's bias towards hubs.
- "mlf": the marginal likelihood filter of Dianati (2016), which reads the weights as counts of independent events and tests each against a maximum-entropy null. Requires whole-number weights.
- "simmelian": the Simmelian backbone of Nick et al. (2013), which ranks each node's neighbours by how embedded the tie is and retains the ties whose endpoints rank each other similarly. Ignores tie weights, and so is the only filter available for an unweighted network. Reads a directed network as an undirected one, since it counts shared neighbours.

By default `NULL`, which uses "lans" where the network is weighted and "simmelian" where it is not. "lans" leads because it assumes nothing about the shape of the weights, and because it retains each node's strongest tie whatever that

shape is. "disparity" is the better known filter, but its null model expects heavy-tailed weights: where weights are more even, a tie's share of its node's strength approaches 1/k for every tie, the p-value approaches 1/e, and the filter retains nothing at all.

`threshold` The cutoff below which a tie is retained. For the four statistical filters this is a significance level, by default 0.05. For "simmelian" it is instead one minus the similarity of the two endpoints' rankings, by default 0.5. By default `NULL`, which uses whichever of these the filter calls for.

`endpoints` Whether a tie is retained where it passes the filter at "either" of its endpoints, by default, or only where it passes at "both". "both" is the more demanding, since a tie must be locally salient to both the sender and the receiver. Applies to the "disparity" and "lans" filters only, since the others weigh both endpoints at once.

Details

The work is done by the `stocnet` method, since that class holds both the ties in the order the network records them and the metadata in which the transformation is recorded. Every other class is coerced to it and back again, so all of them are supported.

Each filter is applied to the ties as the network holds them. Where a network holds parallel ties, each is filtered on its own rather than as the bundle they make together.

Only closed-form filters are offered here. For the backbone models that resample a null distribution, such as the stochastic and fixed degree sequence models, or that extract the backbone of a two-mode projection, see the **backbone** package (Neal 2022).

Value

An object of the same class as the function was given, modified as explained in the function description, details, or section. Functions that split a network return a list of such objects.

References

On the disparity filter:

Serrano, M. Angeles, Marian Boguna, and Alessandro Vespignani. 2009. "Extracting the multiscale backbone of complex weighted networks". *Proceedings of the National Academy of Sciences* 106(16): 6483-6488. doi:10.1073/pnas.0808904106

On locally adaptive network sparsification:

Foti, Nicholas J., James M. Hughes, and Daniel N. Rockmore. 2011. "Nonparametric sparsification of complex multiscale networks". *PLoS ONE* 6(2): e16431. doi:10.1371/journal.pone.0016431

On the noise-corrected filter:

Coscia, Michele, and Frank M. H. Neffke. 2017. "Network backboning with noisy data". *IEEE International Conference on Data Engineering* 33: 425-436. doi:10.1109/ICDE.2017.100

On the marginal likelihood filter:

Dianati, Navid. 2016. "Unwinding the hairball graph: Pruning algorithms for weighted complex networks". *Physical Review E* 93(1): 012304. doi:10.1103/PhysRevE.93.012304

On Simmelian backbones:

Nick, Bobo, Conrad Lee, Padraig Cunningham, and Ulrik Brandes. 2013. "Simmelian backbones: Amplifying hidden homophily in Facebook networks". *Advances in Social Networks Analysis and Mining* 2013: 525-532. doi:10.1145/2492517.2492569

Nocaj, Arlind, Mark Ortmann, and Ulrik Brandes. 2015. "Untangling the hairballs of multi-centered, small-world online social media networks". *Journal of Graph Algorithms and Applications* 19(2): 595-618. doi:10.7155/jgaa.00370

On other backbone models:

Neal, Zachary P. 2022. "backbone: An R package to extract network backbones". *PLoS ONE* 17(5): e0269137. doi:10.1371/journal.pone.0269137

See Also

Other ties: `manip_ties_attr`, `manip_ties_num`, `modif_direction`, `modif_weight`

Other modifications: `modif_concepts`, `modif_direction`, `modif_from_parts`, `modif_from_ternary`, `modif_labels`, `modif_levels`, `modif_miss`, `modif_motifs`, `modif_paths`, `modif_permutation`, `modif_plexity`, `modif_project`, `modif_proximity`, `modif_scope`, `modif_split`, `modif_weight`

Examples

```
to_backbone(ison_networkers)
to_backbone(ison_networkers, filter = "disparity", threshold = 0.2)
```

`modif_concepts` *Modifying networks into concept lattices*

Description

`to_concepts()` projects a network into its concept lattice, also known as its Galois lattice. Each node of the lattice is a concept: a set of nodes together with the set of affiliations they all share, where neither set can grow without the other shrinking. Each tie points from a concept to a concept directly beneath it, one that holds fewer nodes but more shared affiliations.

Usage

```
to_concepts(.data)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods:

```
            data.frame default igraph matrix network stocnet tbl_graph
to_concepts          *       *      *      *       *       *         *
```

In a two-mode network, the first node set (e.g. the rows) supplies the concepts' members, their *extent*, and the second node set (e.g. the columns) their shared affiliations, their *intent*. The lattice is read from top to bottom. The concept at the top holds every node of the first set, and those affiliations, if any, that all of them share. The concept at the bottom holds every affiliation, and those nodes, if any, that have all of them. Moving down the lattice, each concept holds fewer nodes, who share more affiliations. Two nodes are in a concept together exactly where they share every one of its affiliations, and so the lattice records every membership of the original network.

A one-mode network is treated as the incidence of its nodes on their out-neighbours. Each concept is then a set of nodes together with every node that all of them send ties to, a maximal biclique of senders and receivers.

A valued or signed network is dichotomised first, so that every positive value counts as a tie, with a warning.

Value

An object of the same class as the function was given, modified as explained in the function description, details, or section. Functions that split a network return a list of such objects.

Labels

Each concept is named with a "reduced" label, listing only those nodes and affiliations that first appear at it. A node of the first set is named at the lowest concept whose extent holds it, and it is in the extent of every concept above that one too. An affiliation, written in braces, is named at the highest concept whose intent holds it, and it is in the intent of every concept below that one. A concept at which nothing first appears is named "C" and its position, counting from the top. Each label thus appears exactly once, and the full sets are kept in the `extent` and `intent` node attributes, with their sizes in `extent_size` and `intent_size`.

Projection without loss

Projecting a two-mode network with `to_mode1()` or `to_mode2()` records how much two nodes share, but not what they share or with whom else they share it. The concept lattice keeps both. Freeman and White (1993) propose it for this reason, and Freeman (2003) uses it to find the groups among the Southern Women. It grows quickly with the density of the network, however, and a lattice of more than a few dozen concepts is hard to read. Where a network has more than 1000 concepts, the function warns about this.

References

On Galois lattices:

Freeman, Linton C., and Douglas R. White. 1993. "Using Galois lattices to represent network data". *Sociological Methodology* 23: 127-145. doi:10.2307/271008

Freeman, Linton C. 2003. "Finding social groups: A meta-analysis of the southern women data". In *Dynamic Social Network Modeling and Analysis*, 39-97. Washington, DC: The National Academies Press.

On formal concept analysis:

Wille, Rudolf. 1982. "Restructuring lattice theory: An approach based on hierarchies of concepts". In *Ordered Sets*, 445-470. Dordrecht: Reidel. doi:10.1007/9789400977983_15

Ganter, Bernhard, and Rudolf Wille. 1999. *Formal Concept Analysis: Mathematical Foundations*. Berlin: Springer. doi:10.1007/9783642598302

See Also

Other modifications: `modif_backbone`, `modif_direction`, `modif_from_parts`, `modif_from_ternary`, `modif_labels`, `modif_levels`, `modif_miss`, `modif_motifs`, `modif_paths`, `modif_permutation`, `modif_plexity`, `modif_project`, `modif_proximity`, `modif_scope`, `modif_split`, `modif_weight`

Other projections: `modif_project`

Examples

```
to_concepts(ison_southern_women)
# autograph::graphr(to_concepts(ison_southern_women), "layered")
```

`modif_direction` *Modifying networks by formatting their directionality*

Description

These functions reformat manynet-consistent data.

- `to_directed()` reformats undirected network data to a directed network. In a one-mode network each tie is given a direction at random. In a two-mode network every tie runs from the first mode to the second, which `to_redirected()` reverses. A two-mode matrix cannot record direction, so it stays as it is.
- `to_undirected()` reformats directed network data to an undirected network, so that any pair of nodes with at least one directed edge will be connected by an undirected edge in the new network. By default this is equivalent to the "collapse" mode in `{igraph}`, but `rule` offers the other ways of reconciling a pair of ties running in opposite directions, which matters where the network is weighted.
- `to_redirected()` formats directed network data by flipping/transposing any existing direction such that senders become receivers and receivers become senders. This essentially has no effect on undirected networks or reciprocated ties.
- `to_reciprocated()` reformats directed network data such that every directed tie is reciprocated.
- `to_acyclic()` reformats network data to an acyclic graph.

If the format condition is not met, for example `to_undirected()` is used on a network that is already undirected, the network data is returned unaltered. No warning is given so that these functions can be used to ensure conformance.

Unlike the `as_*()` group of functions, these functions always return the same class as they are given, only transforming these objects' properties.

Usage

```
to_directed(.data)

to_undirected(
  .data,
  rule = c("collapse", "min", "max", "mean", "sum", "product")
)

to_redirected(.data)

to_reciprocated(.data)

to_acyclic(.data)
```

Arguments

`.data` An object of a `{manynet}`-consistent class:

- adjacency or incidence `matrix` from `{base}` R
- edgelist `data.frame` from `{base}` R or `tbl/tbl_df` from `{tibble}`
- stocnet `stocnet`, from the `{manynet}` package
- igraph `igraph`, from the `{igraph}` package
- network `network`, from the `{network}` package
- tidygraph `tbl_graph`, from the `{tidygraph}` package

`rule` How the values of a pair of ties running in opposite directions are reconciled into the single value of an undirected tie:

- "collapse" (the default) sums them, so that a tie exists wherever a tie existed in either direction. For an unweighted network this is igraph's "collapse" mode, since a tie in either direction gives 1 either way.
- "sum" is the same operation, named for the arithmetic rather than the intent.
- "min" takes the smaller of the two values, so that a tie is only as strong as the weaker direction. Use where a relationship needs to be confirmed from both sides, as in a mutual friendship nomination.
- "max" takes the larger, so that the stronger direction stands for the pair. Use where a single report is taken as sufficient evidence.
- "mean" averages them, treating the two directions as two readings of one underlying quantity.
- "product" multiplies them, so that a tie survives only where both directions are non-zero, and strong ties are rewarded disproportionately.

Values missing in one direction are not treated as agreement: they propagate, so that NA in either direction gives NA. Use impute_ties() first to state a different assumption.

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods:

	data.frame	default	igraph	matrix	network	stocnet	tbl_graph
to_acyclic		*	*			*	*
to_directed		*	*				
to_reciprocated		*	*	*			
to_redirected	*	*	*	*			
to_undirected	*	*	*	*	*	*	*

Value

An object of the same class as the function was given, modified as explained in the function description, details, or section. Functions that split a network return a list of such objects.

See Also

Other ties: manip_ties_attr, manip_ties_num, modif_backbone, modif_weight

Other modifications: modif_backbone, modif_concepts, modif_from_parts, modif_from_ternary, modif_labels, modif_levels, modif_miss, modif_motifs, modif_paths, modif_permutation, modif_plexity, modif_project, modif_proximity, modif_scope, modif_split, modif_weight

Examples

```
to_undirected(ison_networkers)
to_undirected(ison_networkers, rule = "min")
```

---

modif_from_parts *Joining parts of a network*

Description

These functions join a list of networks that are each a part of one network, holding some of its nodes and the ties among them, back into that one network. Each reverses one of the to_*s() functions that splits a network into parts.

- from_subgraphs() joins a list of subgraphs, as to_subgraphs() returns, into one network.
- from_egos() joins a list of ego networks, as to_egos() returns, into one whole network.

A node or a tie that more than one part holds is held once in the result.

Every function here returns the class of the networks in the list: a 'stocnet' from 'stocnet's, a 'tbl_graph' from 'tbl_graph's, an 'igraph' from 'igraph's, a 'network' from 'network's, and a single matrix from matrices, since the parts are all of one network.

To stack networks that each hold the same nodes, one for each moment, layer, or reporter, see `from_times()` and the other functions there.

Usage

```
from_subgraphs(netlist)

from_egos(netlist)
```

Arguments

`netlist`	A list of network, igraph, tidygraph, matrix, or edgelist objects.

Value

An object of the same class as the function was given, modified as explained in the function description, details, or section. Functions that split a network return a list of such objects.

See Also

Other modifications: `modif_backbone`, `modif_concepts`, `modif_direction`, `modif_from_ternary`, `modif_labels`, `modif_levels`, `modif_miss`, `modif_motifs`, `modif_paths`, `modif_permutation`, `modif_plexity`, `modif_project`, `modif_proximity`, `modif_scope`, `modif_split`, `modif_weight`

Examples

```
ison_adolescents |>
  mutate(unicorn = sample(c("yes", "no"), 8, replace = TRUE)) |>
  to_subgraphs(attribute = "unicorn") |>
  from_subgraphs()
ison_adolescents |>
  to_egos() |>
  from_egos()
```

`modif_from_ternary` *Joining networks along a third dimension*

Description

These functions join a list of networks that each hold the same nodes, one network for each moment, layer, or reporter, into one network. Each tie then records the network it came from in one of its columns: `time` for a moment, `layer` for a layer, and `by` for a reporter. Each reverses one of the `to_*s()` functions that splits a network along such a dimension.

- `from_times()` joins a list of the network at each moment, as `to_times()` returns, back into one network that records time. This is where new work on rejoining a network over time belongs; `from_waves()` and `from_slices()` are the older, form-specific spellings.
- `from_waves()` joins a list of network waves into a longitudinal network.
- `from_slices()` joins a list of time slices of a network into a dynamic network.
- `from_layers()` joins several networks over the same nodes into one multiplex network, keeping each as its own layer. `from_ties()` is an alias. Where `to_layers()` splits a multiplex network into its layers, `from_layers()` reassembles them. To combine the networks' tie values into a single value per dyad instead, use `to_aggregated()` on the result, or `join_ties()` for two networks.
- `from_reporters()` joins a list of the networks that each reporter reported, named after the reporters, into one cognitive social structure (see `is_cognitive()`).

Every function here returns the class of the networks in the list: a 'stocnet' from 'stocnet's, a 'tbl_graph' from 'tbl_graph's, an 'igraph' from 'igraph's, a 'network' from 'network's. A list of matrices gives a three-dimensional array, with one slice for each matrix, named after the list, since a matrix has no column in which to record the moment, layer, or reporter of a tie. `as_stocnet()` reads such an array back, once told what its slices hold.

A tie that a network records as missing stays missing from that network alone, so a node that did not report at one moment, or in one layer, is recorded as not reporting there and nowhere else.

To join networks that are each a part of one network, holding some of its nodes, see `from_subgraphs()` and `from_egos()`.

Usage

```
from_waves(netlist)

from_slices(netlist, remove.duplicates = FALSE)

from_times(netlist)

from_layers(..., layer_names)

from_ties(..., layer_names)

from_reporters(netlist)
```

Arguments

`netlist` A list of network, igraph, tidygraph, matrix, or edgelist objects, over the same nodes.

`remove.duplicates` Should duplicates be removed? By default FALSE. If TRUE, duplicated edges are removed.

`...` Two or more networks over the same nodes to be merged, or a single list of such networks, such as that returned by `to_layers()`. Nodes are matched by name where the networks are labelled, and by position where none of them is and they

are the same size. Unlabelled nodes are otherwise added as new nodes, with a warning, since which of the other nodes they are would be a guess.

layer_names A character vector of names for the different network objects, if not already named within the list.

Details

from_times() rejoins what to_times() returns, stamping each network's ties with the moment it names before binding them, and so inverts it for a network that stamps its moments. Where the networks carry the interval each tie lasts over, the moment is already in those ties, so they are bound and deduplicated rather than stamped again.

from_reporters() takes the reporter of each network from the names of the list, which may be the labels of the nodes or their positions. Where the list is not named, the networks are taken to be the reports of the first nodes, in order.

Value

An object of the same class as the function was given, modified as explained in the function description, details, or section. Functions that split a network return a list of such objects.

See Also

Other modifications: modif_backbone, modif_concepts, modif_direction, modif_from_parts, modif_labels, modif_levels, modif_miss, modif_motifs, modif_paths, modif_permutation, modif_plexity, modif_project, modif_proximity, modif_scope, modif_split, modif_weight

Examples

```
ison_adolescents |>
  mutate_ties(wave = sample(1:4, 10, replace = TRUE)) |>
  to_waves(attribute = "wave") |>
  from_waves()
ison_adolescents |>
  mutate_ties(time = 1:10, increment = 1) |>
  add_ties(c(1,2), list(time = 3, increment = -1)) |>
  to_slices(slice = c(5,7)) |>
  from_slices()
from_times(to_times(ison_tailorshop))
marriage <- to_uniplex(ison_florentine, "marriage")
business <- to_uniplex(ison_florentine, "business")
from_layers(marriage = marriage, business = business)
reports <- list(A = create_ring(4), B = create_star(4))
from_reporters(lapply(reports, add_node_attribute, "name", LETTERS[1:4]))
```

`modif_labels` *Modifying node labels*

Description

These functions add some format to manynet-consistent data.

- `to_labelled()` reformats unlabelled network data to labelled network data from a vector of names or random baby names.
- `to_unlabelled()` reformats labelled network data to unlabelled network data.

If the format condition is not met, for example `to_undirected()` is used on a network that is already undirected, the network data is returned unaltered. No warning is given so that these functions can be used to ensure conformance.

Unlike the `as_*()` group of functions, these functions always return the same class as they are given, only transforming these objects' properties.

Usage

```
to_labelled(.data, names = NULL)

to_named(.data, names = NULL)

to_unlabelled(.data)

to_unnamed(.data)
```

Arguments

`.data` An object of a {manynet}-consistent class:
- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`names` Character vector of the node names. NULL by default.

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods:

	data.frame	default	igraph	matrix	network	tbl_graph
to_labelled	*	*	*	*	*	*
to_unlabelled	*	*	*	*	*	*

Value

An object of the same class as the function was given, modified as explained in the function description, details, or section. Functions that split a network return a list of such objects.

See Also

Other modifications: modif_backbone, modif_concepts, modif_direction, modif_from_parts, modif_from_ternary, modif_levels, modif_miss, modif_motifs, modif_paths, modif_permutation, modif_plexity, modif_project, modif_proximity, modif_scope, modif_split, modif_weight

---

modif_levels *Modifying network levels*

Description

These functions reformat the levels in manynet-consistent network data.

- to_onemode() reformats two-mode network data into one-mode network data by simply removing the nodeset 'type' information. Note that this is not the same as to_mode1() or to_mode2(); it does not project the two-mode network into one of its modes, but rather simply removes the distinction between the two modes.
- to_twomode() reformats one-mode network data into two-mode network data, using a mark to distinguish the two sets of nodes. It also reformats a hypergraph into a two-mode network, in which each hyperedge is a node of the second mode, and for this needs no mark.
- to_multilevel() reformats two-mode network data into multimodal network data, which allows for more levels and ties within modes.

If the format condition is not met, for example to_onemode() is used on a network that is already one-mode, the network data is returned unaltered. No warning is given so that these functions can be used to ensure conformance.

Note that a multilevel network is still two-mode, so as_matrix() returns its incidence matrix. For the square matrix of both modes, use to_onemode() or as_matrix(.data, twomode = FALSE).

Unlike the as_*() group of functions, these functions always return the same class as they are given, only transforming these objects' properties.

Usage

```
to_onemode(.data)

to_twomode(.data, mark)

to_multilevel(.data)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`mark` A logical vector marking two types or modes. By default "type".

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods:

```
               default igraph matrix network stocnet tbl_graph
to_multilevel        *      *      *               *         *
to_onemode           *      *      *                         *
to_twomode           *      *              *        *         *
```

Value

An object of the same class as the function was given, modified as explained in the function description, details, or section. Functions that split a network return a list of such objects.

See Also

Other modifications: `modif_backbone`, `modif_concepts`, `modif_direction`, `modif_from_parts`, `modif_from_ternary`, `modif_labels`, `modif_miss`, `modif_motifs`, `modif_paths`, `modif_permutation`, `modif_plexity`, `modif_project`, `modif_proximity`, `modif_scope`, `modif_split`, `modif_weight`

`modif_miss` *Imputing missing and incomplete network data*

Description

These functions impute what a network did not observe:

- `impute_ties()` imputes the ties a network records as missing, and the values of the ties it records as incomplete.
- `impute_nodes()` imputes the attributes of the nodes it records as incomplete.
- `to_imputed()` runs both in a single call.

A network is *missing* a tie where the tie itself was not observed, so that whether it exists is not known. A tie or a node is *incomplete* where it is there and observed, but an attribute of it is not known, such as the strength of a tie or the age of a node. Imputing the first is a question of existence, and imputing the second is a question of value, so each takes its own rules.

If there is nothing to impute, the network data is returned unaltered and no warning is given, so that these functions can be used to ensure conformance.

Usage

```
impute_ties(
  .data,
 rule = c("zero", "density", "reciprocity", "indegree", "mean", "median", "modal"),
  which = c("nonresponse", "unrecorded", "incomplete")
)

impute_nodes(
  .data,
  rule = c("modal", "mean", "median", "neighbourhood"),
  attribute = NULL
)

to_imputed(.data, ties = "zero", nodes = "modal")
```

Arguments

`.data`	An object of a {manynet}-consistent class: • adjacency or incidence `matrix` from {base} R • edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble} • stocnet `stocnet`, from the {manynet} package • igraph `igraph`, from the {igraph} package • network `network`, from the {network} package • tidygraph `tbl_graph`, from the {tidygraph} package
`rule`	How the imputed value is arrived at. See the Rules section for the options and what each does. By default "zero".
`which`	Which of the states a tie can be in to impute, one or more of "nonresponse", "unrecorded", and "incomplete". See the four states section. By default all three.
`attribute`	A character vector naming the node attributes to impute. By default NULL, which imputes every attribute that holds a missing value.
`ties`	The rule `to_imputed()` passes to `impute_ties()`, or NULL to leave the ties alone.
`nodes`	The rule `to_imputed()` passes to `impute_nodes()`, or NULL to leave the nodes alone.

Value

An object of the same class as the function was given, modified as explained in the function description, details, or section. Functions that split a network return a list of such objects.

What is recorded

Imputation manufactures data, so what was imputed, how much of it, and by which rule is recorded under the "imputation" name of the network's transformations, which `describe_transformations()` describes. Item 4.6 of the GRAND guidelines asks for the imputation method and the number of nodes or ties that were imputed, and places it among the transformations of raw data into analytic data, beside symmetrising, dichotomising, projecting, and aggregating, which is where the other `to_*()` functions record themselves.

One entry is added for the missing ties, one for the incomplete tie values, and one for each node attribute, so that a reader can tell which attributes hold manufactured values and which were observed throughout:

```
as_infolist(to_imputed(ison_classmates))$transformations$imputation
#> "73 missing ties (zero)"  "4 incomplete 'religion' values (modal)" ...
```

The element accumulates rather than replaces, so a network imputed in more than one step reports each of them in order. A matrix or an edgelist has nowhere to hold information about itself, so nothing is recorded for those two classes.

References

On missing data:

Krause, Robert, Mark Huisman, Christian Steglich, and Tom A.B. Snijders. 2020. "Missing data in cross-sectional networks: An extensive comparison of missing data treatment methods". *Social Networks*, 62: 99-112. doi:10.1016/j.socnet.2020.02.004

See Also

Other modifications: `modif_backbone`, `modif_concepts`, `modif_direction`, `modif_from_parts`, `modif_from_ternary`, `modif_labels`, `modif_levels`, `modif_motifs`, `modif_paths`, `modif_permutation`, `modif_plexity`, `modif_project`, `modif_proximity`, `modif_scope`, `modif_split`, `modif_weight`

Examples

```
missTest <- ison_adolescents |>
   add_tie_attribute("weight", c(1,NA,NA,1,1,1,NA,NA,1,1)) |>
   as_matrix()
missTest
impute_ties(missTest)
impute_ties(missTest, "mean")
impute_nodes(fict_lotr, "modal", "Race")
to_imputed(ison_classmates)
```

`modif_motifs` *Listing the motifs a network can contain*

Description

`to_motifs()` returns a named list of small networks representing the motifs (subgraphs, or isomorphism classes) that a network of a given type could contain. Unlike most other `to_*()` functions, it does not modify and return the network passed to it, but instead uses that network only to work out which motifs are relevant, returning the whole reference set of motifs as a list. It is meant to help interpret the results of motif counts, such as `netrics::net_x_triads()`.

Either `.data` or `n` (or both) can be given. When a network is passed to `.data`, the direction, signedness, and mode(s) of the motifs are inferred from it, as is the number of nodes `n` unless `n` is also given explicitly. Passing `n` alongside a network lets you choose, say, the dyadic, triadic, or tetradic motifs of that kind of network. When only `n` is given (e.g. `to_motifs(n = 3)`, or simply `to_motifs(3)`), the `directed`/`signed` arguments are respected instead, which is handy for teaching or illustration.

For one-mode networks, motifs are implemented undirected for `n = 2` to `n = 4`, directed for `n = 2` and `n = 3`, signed undirected for `n = 2` and `n = 3`, and signed directed for `n = 2`. Where more nodes are requested (or inferred from a larger network) than are implemented, the largest available motif set is returned.

For two-mode networks, the seven bipartite motifs up to four nodes are returned, labelled by their `bmotif` dictionary IDs (Simmons et al. 2019): motif 1 (a single tie), motifs 2-3 (three nodes), and motifs 4-7 (four nodes, namely the two stars, the 2x2 path, and the 2x2 four-cycle).

Usage

```
to_motifs(.data = NULL, n = NULL, directed = FALSE, signed = FALSE)

create_motifs(.data = NULL, n = NULL, directed = FALSE, signed = FALSE)
```

Arguments

`.data`	An optional `{manynet}`-consistent network (matrix, edgelist, igraph, tidygraph, network, or stocnet object), from which the direction, signedness, mode(s), and (unless `n` is given) number of nodes of the motifs are inferred. Either `.data` or `n` must be provided.
`n`	An optional number of nodes the motifs should contain, as a single integer (one-mode) or a length-two integer vector (two-mode). If omitted, it is inferred from `.data`; if given alongside `.data` it overrides the network's own size (e.g. to list the triadic motifs of a larger network). Either `.data` or `n` must be provided.
`directed`	Logical whether the motifs should be directed. By default FALSE. Ignored (and inferred instead) when `.data` is a network.
`signed`	Logical whether the motifs should be signed. By default FALSE. Ignored (and inferred instead) when `.data` is a network. Currently available for undirected `n = 2` and `n = 3`, and directed `n = 2` (see the "Signed directed dyads" section).

Value

A named list of {manynet}-compatible networks, one per motif.

Signed directed dyads

The six signed directed dyads (n = 2, directed, signed) are the Holland-Leinhardt dyad census (Null, Asymmetric, Mutual) refined by the sign of each arc. They correspond to the dyadic reciprocity motifs of Gallo et al. (2025) as follows:

to_motifs() label	Structure	Gallo et al. (2025)
Null	no ties	(empty dyad)
Asymmetric+	one positive arc	$L^{+}_{\rightarrow}$ (single positive)
Asymmetric-	one negative arc	$L^{-}_{\rightarrow}$ (single negative)
Mutual++	reciprocated, both positive	$L^{+}_{\leftrightarrow}$ (reciprocated positive)
Mutual-	reciprocated, both negative	$L^{-}_{\leftrightarrow}$ (reciprocated negative)
Mutual+-	reciprocated, discordant signs	$L^{\pm}_{\leftrightarrow}$ (reciprocated mixed)

References

Simmons, Benno I., Michelle J.M. Sweering, Maybritt Schillinger, Lynn V. Dicks, William J. Sutherland, and Riccardo Di Clemente. 2019. "bmotif: A package for motif analyses of bipartite networks." *Methods in Ecology and Evolution* 10(5): 695-701. doi:10.1111/2041210X.13149

Gallo, Anna, Fabio Saracco, Renaud Lambiotte, Diego Garlaschelli, and Tiziano Squartini. 2025. "Patterns of link reciprocity in directed, signed networks." *Physical Review E* 111(2): 024312. doi:10.1103/PhysRevE.111.024312

See Also

Other modifications: modif_backbone, modif_concepts, modif_direction, modif_from_parts, modif_from_ternary, modif_labels, modif_levels, modif_miss, modif_paths, modif_permutation, modif_plexity, modif_project, modif_proximity, modif_scope, modif_split, modif_weight

Examples

```
to_motifs(3)
to_motifs(n = 2, directed = TRUE)
to_motifs(create_ring(8), n = 3)
to_motifs(3, signed = TRUE)
to_motifs(2, directed = TRUE, signed = TRUE)
```

`modif_paths` *Modifying networks paths*

Description

These functions return tidygraphs containing only special sets of ties:

- `to_matching()` returns only the matching ties in some network data.
- `to_mentoring()` returns only ties to nodes' closest mentors.
- `to_tree()` returns the spanning tree in some network data or, if the data is unconnected, a forest of spanning trees.
- `to_dominating()` returns the dominating tree of the network.
- `to_geodesic()` returns a network of the shortest path lengths between nodes.

Usage

```
to_matching(.data, mark = "type", capacities = NULL)

to_mentoring(.data, elites = 0.1)

to_tree(.data)

to_dominating(.data, from, direction = c("out", "in"))

to_geodesic(.data)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`mark` A logical vector marking two types or modes. By default "type".

`capacities` An integer or vector of integers the same length as the nodes in the network that describes the maximum possible degree the node can have in the matched network.

`elites` The proportion of nodes to be selected as mentors. By default this is set at 0.1. This means that the top 10% of nodes in terms of degree, or those equal to the highest rank degree in the network, whichever is the higher, will be used to select the mentors.

Note that if nodes are equidistant from two mentors, they will choose one at random. If a node is without a path to a mentor, for example because they are an isolate, a tie to themselves (a loop) will be created instead. Note that this is a different default behaviour than that described in Valente and Davis (1999).

`from` The index or name of the node from which the path should be traced.

`direction` Character string, "out" bases the measure on outgoing ties, "in" on incoming ties, and "all" on either/the sum of the two. By default "all".

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods:

```
              data.frame default igraph matrix network stocnet tbl_graph
to_dominating                  *      *
to_geodesic                    *      *                     *         *
to_matching            *       *      *      *       *                *
to_mentoring                   *      *
to_tree                        *      *
```

Value

An object of the same class as the function was given, modified as explained in the function description, details, or section. Functions that split a network return a list of such objects.

Matching

This function attempts to solve the stable matching problem, also known as the stable marriage problem, upon a given two-mode network (or other network with a binary mark).

In the basic version, `to_matching()` uses `igraph::max_bipartite_match()` to return a network in which each node is only tied to one of its previous ties. The number of these ties left is its *cardinality*, and the algorithm seeks to maximise this such that, where possible, each node will be associated with just one node in the other mode or some other mark. The algorithm used is the push-relabel algorithm with greedy initialization and a global relabelling after every $\frac{n}{2}$ steps, where n is the number of nodes in the network.

In the more general version, each node may have a larger capacity, or even different capacities. Here an implementation of the Gale-Shapley algorithm is used, in which an iterative process of proposal and acceptance is repeated until all are matched or have exhausted their lists of preferences. This is, however, computationally slower.

Mentoring

This function returns a network in which each node is tied to its closest mentor. The mentors are selected as the top nodes in terms of degree, or those equal to the highest rank degree in the network, whichever is the higher. The mentees are then tied to the closest mentor in terms of geodesic distance. This can be useful for showing the hierarchical structure of a network, for example in an organisational context, or identifying the most influential nodes in a network.

Dominating

This function returns a network in which each node is tied to its closest dominating node. A dominating set D in a graph G is a subset of nodes such that every node not in D is adjacent to at least one node in D. The domination number, $\gamma(G)$, is the size of the smallest such set. This concept underpins many practical network problems: facility location, sensor placement, control theory applied to networks, and backbone selection in ad hoc wireless routing.

Geodesics

This function returns a weighted network of the same nodes, in which each tie is the length of the shortest path, or geodesic, from one node to another. A pair of nodes that no path joins has no tie, and so a matrix of the result holds 0 for it off the diagonal. Where the network is directed, a path follows the direction of its ties; use `to_undirected()` first for paths that do not.

Where the network is signed, only its positive ties are paths. Where it is weighted, a stronger tie is a shorter step: each tie costs the mean tie weight divided by its own weight (Opsahl, Agneessens, and Skvoretz 2010), so that a tie of average weight is a step of one. The paths of a cognitive social structure are those of its locally aggregated structure, in which a tie is present where both of its nodes report it. A two-mode network returns the paths between all of its nodes, of either mode.

In a hypergraph, a path runs from node to node through the hyperedges they share, and its length is the number of hyperedges it passes through (Berge 1973). The result is a network of the nodes alone, which is the same as the network of geodesics of the hypergraph's projection, `to_mode(.data, 1)`.

References

On matching:

Gale, David, and Lloyd Stowell Shapley. 1962. "College admissions and the stability of marriage". *The American Mathematical Monthly*, 69(1): 9–14. doi:10.2307/2312726

Goldberg, Andrew V., and Robert E. Tarjan. 1986. "A new approach to the maximum flow problem". *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*. 136-146. doi:10.1145/12130.12144

On mentoring:

Valente, Thomas, and Rebecca Davis. 1999. "Accelerating the Diffusion of Innovations Using Opinion Leaders", *Annals of the American Academy of Political and Social Science* 566: 56-67. doi:10.1177/000271629956600105

On minimum spanning trees:

Boruvka, Otakar. 1926. "O jistem problemu minimalnim". *Prace Mor. Prirodoved. Spol. V Brne III* 3: 37-58.

Kruskal, Joseph B. 1956. "On the shortest spanning subtree of a graph and the travelling salesman problem". *Proceedings of the American Mathematical Society* 7(1): 48-50. doi:10.1090/S0002-9939195600786867

Prim, R.C. 1957. "Shortest connection networks and some generalizations". *Bell System Technical Journal* 36(6):1389-1401. doi:10.1002/j.15387305.1957.tb01515.x

On dominating sets:

Bange, David W., Anthony E. Barkauskas, and Peter J. Slater. 1988. "Efficient dominating sets in graphs." In *Applications of Discrete Mathematics*, SIAM, pp. 189–99.

Haynes, Teresa W., Stephen T. Hedetniemi, and Peter J. Slater. 1998. *Fundamentals of Domination in Graphs*. CRC Press.

On geodesics:

Opsahl, Tore, Filip Agneessens, and John Skvoretz. 2010. "Node centrality in weighted networks: Generalizing degree and shortest paths". *Social Networks* 32(3): 245-251. doi:10.1016/j.socnet.2010.03.006

Berge, Claude. 1973. *Graphs and Hypergraphs*. North-Holland.

See Also

Other modifications: `modif_backbone`, `modif_concepts`, `modif_direction`, `modif_from_parts`, `modif_from_ternary`, `modif_labels`, `modif_levels`, `modif_miss`, `modif_motifs`, `modif_permutation`, `modif_plexity`, `modif_project`, `modif_proximity`, `modif_scope`, `modif_split`, `modif_weight`

Examples

```
to_matching(ison_southern_women)
to_geodesic(ison_adolescents)
```

`modif_permutation` *Network permutation*

Description

`to_permuted()` permutes the network using a Fisher-Yates shuffle on both the rows and columns (for a one-mode network) or on each of the rows and columns (for a two-mode network).

Usage

```
to_permuted(.data, with_attr = TRUE)
```

Arguments

`.data`	An object of a `{manynet}`-consistent class: • adjacency or incidence `matrix` from `{base}` R • edgelist `data.frame` from `{base}` R or `tbl/tbl_df` from `{tibble}` • stocnet `stocnet`, from the `{manynet}` package • igraph `igraph`, from the `{igraph}` package • network `network`, from the `{network}` package • tidygraph `tbl_graph`, from the `{tidygraph}` package
`with_attr`	Logical whether any attributes of the object should be retained. By default TRUE.

Value

An object of the same class as the function was given, modified as explained in the function description, details, or section. Functions that split a network return a list of such objects.

See Also

Other modifications: modif_backbone, modif_concepts, modif_direction, modif_from_parts, modif_from_ternary, modif_labels, modif_levels, modif_miss, modif_motifs, modif_paths, modif_plexity, modif_project, modif_proximity, modif_scope, modif_split, modif_weight

---

modif_plexity *Modifying network complexity*

Description

These functions reformat manynet-consistent data.

- to_anti() reformats network data into its complement, where only ties *not* present in the original network are included in the new network.
- to_simplex() reformats complex network data, containing loops, to simplex network data, without any loops. Parallel ties are kept, since a second tie between two nodes is not a loop; tie_is_parallel() marks them.
- to_uniplex() reformats multiplex network data to a single type of tie. to_layer() is an alias, using the layer-based vocabulary of layer_names(), net_layers(), and to_layers(). Use to_layers() to split a network into all of its layers at once.
- to_aggregated() combines the ties of a network that differ only in one tie column, dyad by dyad, according to a rule: its layers, the reports of its reporters, the gossip about each target, or its moments. It can also combine parallel ties. Where to_uniplex() selects one layer and discards the rest, to_aggregated() retains what every layer records. to_flat() is an alias that combines layers, and takes the rule as its second argument.
- to_disaggregated() turns tie weights back into that many parallel ties, reversing to_aggregated(over = NULL, rule = "sum").

If the format condition is not met, for example to_undirected() is used on a network that is already undirected, the network data is returned unaltered. No warning is given so that these functions can be used to ensure conformance.

Unlike the as_*() group of functions, these functions always return the same class as they are given, only transforming these objects' properties.

Usage

```
to_anti(.data)

to_simplex(.data)

to_uniplex(.data, layer, tie)

to_layer(.data, layer, tie)

to_aggregated(
```

```
  .data,
  over = "layer",
  rule = c("max", "min", "mean", "sum", "product"),
  reporters = c("all", "sender", "receiver", "both")
)

to_flat(
  .data,
  rule = c("max", "min", "mean", "sum", "product"),
  over = "layer"
)

to_disaggregated(.data)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`layer` Character string naming one of the layers, or tie types, in the network, i.e. one of those returned by `layer_names()`, to which the network should be reduced. Where a network holds no tie types, it is already uniplex and is returned unchanged.

`tie` Deprecated name for `layer`, retained for one version.

`over` The tie column over which ties are combined, one of the columns that `as_stocnet()` takes from the third dimension of an array:

- "layer" (the default) combines the layers of a multiplex network,
- "by" combines the reports of the reporters in a cognitive social structure,
- "about" combines the gossip about each target,
- "time" combines the moments of a longitudinal network.

`NULL` combines only parallel ties, the ties that `tie_is_parallel()` marks. `to_disaggregated()` reverses this where `rule = "sum"`.

Each tie counts by its weight, or by its sign where it has no weight, and otherwise as 1. Signed values can cancel one another out, and a combined value of zero is no tie.

`rule` How the networks' tie values are reconciled into a single value per dyad. A dyad tied in only one of the networks counts as untied in the others, which is what makes rules such as "min" and "product" meaningful.

- "max" (the default) takes the largest of the values, so that a tie in any of the networks is a tie in the result. This is the union of the networks.

- "min" takes the smallest, so that only ties present in all of them survive. This is their intersection.
- "mean" averages the values, treating the networks as several readings of one underlying relationship.
- "sum" adds them, so that ties reinforce one another.
- "product" multiplies them, so that a tie survives only where every network records one, and strong ties are rewarded disproportionately.

Missing values propagate rather than being ignored, so that a dyad unobserved in any of the networks is unobserved in the result. Use `impute_ties()` first to state a different assumption.

`reporters` Which reports are combined where `over = "by"`:

- "all" (the default) combines the reports of every reporter.
- "sender" takes, for the tie from i to j, the report of i.
- "receiver" takes, for the tie from i to j, the report of j.
- "both" combines the reports of i and j by the `rule`. With `rule = "min"` this is the intersection locally aggregated structure of Krackhardt (1987), and with `rule = "max"` it is the union one.

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods:

```
Warning in utils::.S3methods(f): generic function 'to_layer' dispatches methods
for generic 'to_uniplex'
Warning in utils::.S3methods(f): generic function 'to_layer' dispatches methods
for generic 'to_uniplex'
                array data.frame default igraph matrix stocnet tbl_graph
to_aggregated       *                 *                    *         *
to_anti                       *       *      *      *                *
to_disaggregated                      *             *       *
to_simplex                    *       *      *      *       *         *
to_uniplex                            2                                2
```

Value

An object of the same class as the function was given, modified as explained in the function description, details, or section. Functions that split a network return a list of such objects.

Aggregating reports

Over reporters, "max" gives a tie where any reporter reports one, "min" where all of them do, "mean" the share of the reporters that do, and "sum" how many of them do. A consensus structure is therefore `to_aggregated(.data, over = "by", rule = "mean")` followed by `to_unweighted(threshold = 0.5)`. A reporter that did not report is left out of the reports that are combined, rather than making every tie missing, and a tie is missing only where none of the reporters it is pooled over reported it. Egocentric data has no roster that all the egos report on, so its reports cannot be combined.

Disaggregating

`to_disaggregated()` turns each tie of weight w into w parallel ties, and so reverses `to_aggregated(over = NULL, rule = "sum")`. No other aggregation can be reversed, since the values it combined are not kept; `to_layers()`, `to_reporters()`, and `to_times()` split a network into its parts without losing them. The weights must be whole numbers. A negative weight becomes that many negative ties, held as weights of -1, which is how a signed network holds its signs. A matrix cannot hold parallel ties, so it is returned as it is.

References

On aggregating cognitive social structures:
Krackhardt, David. 1987. "Cognitive social structures". *Social Networks*, 9(2): 109-134. doi:10.1016/03788733(87)900098

See Also

Other modifications: `modif_backbone`, `modif_concepts`, `modif_direction`, `modif_from_parts`, `modif_from_ternary`, `modif_labels`, `modif_levels`, `modif_miss`, `modif_motifs`, `modif_paths`, `modif_permutation`, `modif_project`, `modif_proximity`, `modif_scope`, `modif_split`, `modif_weight`

Examples

```
to_anti(ison_southern_women)
as_tidygraph(create_filled(5)) |>
  mutate_ties(type = sample(c("friend", "enemy"), 10, replace = TRUE)) |>
  to_uniplex("friend")
to_aggregated(ison_florentine, rule = "sum")
to_disaggregated(to_aggregated(ison_koenigsberg, over = NULL, rule = "sum"))
```

`modif_project` *Modifying networks projection*

Description

These functions offer tools for projecting manynet-consistent data:

- `to_mode()` projects a two-mode network to a one-mode network of the node set given by the `mode` argument, which accepts either the index of the mode or its name.
- `to_mode1()` projects a two-mode network to a one-mode network of the first node set's (e.g. rows) joint affiliations to nodes in the second node set (columns).
- `to_mode2()` projects a two-mode network to a one-mode network of the second node set's (e.g. columns) joint affiliations to nodes in the first node set (rows).
- `to_linegraph()` projects a network to its line graph, where the ties become nodes and incident nodes become their ties.
- `to_hypergraph()` reformats one-mode or two-mode network data as hypergraph data, where ties can connect more than two nodes.
- `to_simplicial()` projects network data into a simplicial complex, such as the maximal cliques of a one-mode network.

Usage

```
to_mode1(
  .data,
 similarity = c("count", "jaccard", "rand", "pearson", "yule", "match", "overlap",
   "crossmin", "maxcrossmin", "sqdiff", "covariance", "bonacich", "ochiai", "ochiai2",
   "czekanowski", "sokalsneath", "hamann", "rogerstanimoto", "euclidean", "manhattan",
    "hamming", "cosine", "spearman", "kendall", "ruzicka")
)

to_mode2(
  .data,
 similarity = c("count", "jaccard", "rand", "pearson", "yule", "match", "overlap",
   "crossmin", "maxcrossmin", "sqdiff", "covariance", "bonacich", "ochiai", "ochiai2",
   "czekanowski", "sokalsneath", "hamann", "rogerstanimoto", "euclidean", "manhattan",
    "hamming", "cosine", "spearman", "kendall", "ruzicka")
)

to_mode(
  .data,
  mode = 1,
 similarity = c("count", "jaccard", "rand", "pearson", "yule", "match", "overlap",
   "crossmin", "maxcrossmin", "sqdiff", "covariance", "bonacich", "ochiai", "ochiai2",
   "czekanowski", "sokalsneath", "hamann", "rogerstanimoto", "euclidean", "manhattan",
    "hamming", "cosine", "spearman", "kendall", "ruzicka")
)

to_linegraph(.data)

to_hypergraph(.data)

to_simplicial(.data)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`similarity` Method for establishing ties in the projection, "count" by default.

The measures are grouped below by what they are sensitive to. Within a group they are monotone transformations of one another, and so rank dyads identically, differing only in their scale. Choosing between the groups therefore matters rather more than choosing within one.

Measures defined for valued as well as binary data:

- "count" counts the coinciding ties, the cross-product XX'. For valued data it sums the products of tie strengths. Interpret it as the degree of opportunity between two nodes. This is the default, and the quickest, since it can use `igraph::bipartite_projection()` directly.
- "match" counts the cells in which two nodes hold exactly the same value, joint absences included. For binary data this is "rand" multiplied by the number of nodes in the other mode, but for valued data it is the more general measure, since it registers agreement at any tie strength. Use it where the level of involvement is meaningful in itself.
- "crossmin" sums the smaller of each pair of tie strengths. Interpret it as the capacity two nodes could jointly bring to bear. For binary data it reduces to "count".
- "overlap" divides "crossmin" by the smaller of the two nodes' total tie strength, the Szymkiewicz-Simpson or overlap coefficient, and so lies in $[0, 1]$ for non-negative data. Use it where one node is much more active than the other, and the less active node's rarity should not depress the score. For binary data it is the count over the smaller degree. xUCINET keeps that count for valued data too, where it can exceed 1 and is no longer the coefficient.
- "ruzicka" divides "crossmin" by the two nodes' combined tie strength less than shared, Ruzicka's weighted Jaccard coefficient, and so also lies in $[0, 1]$. For binary data it is "jaccard", so use it where "jaccard" is wanted but the tie strengths should be kept.
- "maxcrossmin" takes the largest such minimum rather than their sum, so that a single strong shared affiliation stands for the pair. For binary data it collapses to an indicator of any shared affiliation, and so is of little use there.
- "sqdiff" inverts the sum of squared differences in tie strength, as $1/(1 + d)$, so that identical rows give 1 and larger values mean more alike, as for every other measure here. Recover the raw sum of squared differences as $1/x - 1$.
- "pearson" gives Pearson's product-moment correlation and "covariance" its unstandardised counterpart. Use "covariance" where the variance in involvement is itself of interest, and "pearson" where it is not.
- "spearman" and "kendall" are the rank counterparts of "pearson". Use them where tie strengths order the affiliations reliably but their spacing does not, as with ordinal ratings.
- "cosine" gives the cosine of the angle between two nodes' rows. It differs from "pearson" in not centring them first, so that it reads two nodes as alike where their involvements are proportional rather than where they depart from the average in the same direction. For binary data it agrees with "ochiai", except that a node with no ties at all is reported as no more similar to another than any other node, where "ochiai" would divide by zero.
- "euclidean" and "manhattan" invert the straight-line and the city-block distance between two rows, again as $1/(1 + d)$. "manhattan" sums the absolute differences in tie strength, so that a large discrepancy on one affiliation counts no more than the same total spread over several; "euclidean" penalises the concentrated discrepancy more heavily, as "sqdiff" does. Recover either raw distance as $1/x - 1$.

Measures defined for binary data only, where a valued network is dichotomised at zero with a warning. Writing a for the cells in which both nodes are present, b and c for those in which just one is, and d for those in which neither is:

- Sensitive to co-presence, ignoring joint absence: "jaccard" is $a/(a+b+c)$, opportunity weighted by participation; "czekanowski" (the Dice or Sorensen coefficient) is $2a/(2a + b + c)$, which double-weights co-presence; and "sokalsneath" is $a/(a+2(b+c))$, which instead double-weights mismatch. Use these where joint non-participation says nothing, as in a sparse affiliation network with many events. For valued data, "ruzicka" is the counterpart of "jaccard" that keeps the tie strengths.
- Sensitive to matching, counting joint absence as evidence: "rand", the Simple Matching Coefficient, is $(a + d)/(a + b + c + d)$; "hamann" is $((a + d) - (b + c))/(a + b + c + d)$, the same quantity rescaled onto $[-1, 1]$ so that its sign reports whether matches outnumber mismatches; and "rogerstanimoto" is $(a+d)/(a+2(b+c)+d)$, which double-weights mismatch. Interpret these as the degree of behavioural mirroring between two nodes, and use them where not attending is as informative as attending.
- Sensitive to association, through the odds ratio ad/bc: "yule" is Yule's Q, $(ad-bc)/(ad+bc)$, which has a straightforward reading as a rescaled odds ratio; and "bonacich" is $\sqrt{ad}/(\sqrt{ad}+\sqrt{bc})$, which is Yule's Y rescaled onto $[0, 1]$. Both saturate at 1 wherever b or c is zero.
- Geometric: "ochiai" is $a/\sqrt{(a + b)(a + c)}$, the cosine of the angle between two nodes' affiliation vectors, and "ochiai2" (also known as Sokal and Sneath's fifth measure) is $ad/\sqrt{(a + b)(a + c)(d + b)(d + c)}$, its counterpart including joint absence. Neither is monotone in any of the above, so both are worth trying alongside them.
- "hamming" inverts the Hamming distance, the number of cells in which the two nodes differ, as $1/(1 + d)$. It is a monotone transformation of "rand", and so ranks dyads identically, but states the disagreement as a count rather than as a proportion of agreement.

`mode` Which mode to project the network to, as an index, 1 or 2, as `FALSE` or `TRUE`, or as the name of the mode. Mode 1 is the first node set, e.g. the rows, and is the default. `FALSE` is the first mode and `TRUE` the second, as the modes of an unnamed two-mode network are marked.

A name is matched against the names of the modes, which `mode_names()` returns and `add_info()` sets. The match ignores case, plurals, and any other words in the name, so that a network whose modes are named "women" and "social events" can be projected with either "social events", "events", or "event". Where a name matches both modes, as "events" would where they are named "social events" and "work events", the function reports the ambiguity and asks for a name that tells them apart, or an index.

The network must have two modes. Projecting one mode of a network of three or more would need a second mode to project through, which this function does not yet accept, so it reports the modes it found instead.

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods:

	data.frame	default	igraph	matrix	network	stocnet	tbl_graph
to_hypergraph		*	*		*	*	*
to_linegraph		*	*				
to_mode1	*	*	*	*	*	*	*
to_mode2	*	*	*	*	*	*	*
to_simplicial		*	*		*	*	*

Value

An object of the same class as the function was given, modified as explained in the function description, details, or section. Functions that split a network return a list of such objects.

Comparison of two-mode projection methods

Category	Feature	manynet::to_mode1()/to_mode2()	igraph::bipartite_proje
Input	Dedicated function	yes	yes
	Accepted input classes	igraph, network, tidygraph, matrix, edgelist	igraph only
	Detects mode membership from	mode node attribute	type vertex attribute
Projection	Returns both projections at once	no — one per call	yes — list of two
	Projects mode 1 (actors)	to_mode1()	which = "true"
	Projects mode 2 (events)	to_mode2()	which = "false"
Weights	Raw co-membership counts	yes	yes (multiplicity = TRUE)
	Binary (unweighted) output	yes	yes (multiplicity = FALSE)
	Jaccard normalisation	yes	no
	Cosine normalisation	yes	no
	Other similarity measures	25 in all, see similarity	no
Attributes	Retains node attributes	yes	yes
	Retains edge attributes	weight only	weight only
	Removes self-loops automatically	yes	yes
Output	Output class matches input	yes	no
	Directed projection support	limited	no — undirected only
Usability	Lines of code (basic case)	1	1
	Lines of code (Jaccard weights)	1	~8 manual
	Pipe-friendly	yes	with wrappers

Hypergraphs

A hypergraph is a network in which a tie, a hyperedge, can connect any number of nodes, such as the people at a meeting or the authors of a paper. to_hypergraph() reformats one-mode or two-mode network data as hypergraph data and returns the class it was given. Nothing is lost either way.

For two-mode networks, each node of the second mode becomes a hyperedge that connects the nodes of the first mode tied to it. The attributes of the second-mode nodes become attributes of the hyperedges, and the attributes of the ties, such as their weights, become attributes of each node's membership. `to_twomode()` returns the two-mode network.

For one-mode networks, each tie becomes a hyperedge of its two nodes, and keeps its attributes. Where the ties are directed, the hyperedges are too: the sender is on the 'from' side and the receiver on the 'to' side. `to_mode()` returns the one-mode network.

A 'stocnet' object holds each hyperedge as one row of its ties, which lists the hyperedge's members in a `with` column (see the Hypergraphs section of `make_stocnet()`). Other classes have no hyperedges of their own, so they hold each hyperedge as a node of a second mode, and mark that they do so, so that `is_hypergraph()` recognises them and coercion back to a 'stocnet' object returns the hyperedges. A 'network' object is the exception, since `{network}` has hyperedges.

Simplicial complexes

A simplicial complex is a hypergraph in which every subset of a hyperedge is a hyperedge too, so that it is described in full by its largest hyperedges, or faces. `to_simplicial()` projects network data into the largest faces of a simplicial complex, and marks the result as one.

For one-mode networks, the faces are the maximal cliques of the network, since every subset of a clique is a clique. The direction, weights, and other attributes of the ties are not carried. `to_mode()` returns the one-mode network that the cliques cover.

For two-mode networks and hypergraphs, the faces are the hyperedges that are not a subset of another hyperedge.

References

On two-mode projection:

Borgatti, Stephen P., and Daniel S. Halgin. 2011. "Analyzing affiliation networks". In *The SAGE Handbook of Social Network Analysis*, 417-433. London: SAGE.

Bonacich, Phillip. 1972. "Technique for analyzing overlapping memberships". *Sociological Methodology* 4: 176-185. doi:10.2307/270732

See Also

`to_proximity()`, which applies these same measures to a one-mode network, comparing nodes on their ties to one another rather than on their affiliations to a second mode.

`to_cosine()`, which takes the cosine over the columns of a matrix without projecting it.

Other modifications: `modif_backbone`, `modif_concepts`, `modif_direction`, `modif_from_parts`, `modif_from_ternary`, `modif_labels`, `modif_levels`, `modif_miss`, `modif_motifs`, `modif_paths`, `modif_permutation`, `modif_plexity`, `modif_proximity`, `modif_scope`, `modif_split`, `modif_weight`

Other projections: `modif_concepts`

Examples

```
to_mode1(ison_southern_women)
to_mode2(ison_southern_women)
to_mode(ison_southern_women, 2)
```

```
to_mode(ison_southern_women, "events")
to_linegraph(ison_adolescents)
to_hypergraph(ison_southern_women)
to_simplicial(ison_adolescents)
```

`modif_proximity` *Node proximity*

Description

These functions construct a square, node-by-node matrix of how alike the nodes of a one-mode network are:

- `to_proximity()` compares each pair of nodes on their ties, using any of the measures that `to_mode1()` offers. Given a two-mode network, or a node-by-feature profile matrix such as a motif census, it compares the rows or the columns as they lie.
- `to_correlation()` performs a Pearson pairwise correlation, choosing how to treat the diagonal and the reciprocal ties from the network's format.
- `to_cosine()` takes the cosine of the angle between the columns of a matrix.

Where `to_mode1()` compares nodes on their affiliations to a second mode, these functions compare nodes on their ties to one another. The two share one measure vocabulary and differ only in the profile compared. For a two-mode network there is no difference, and `to_proximity()` returns what `to_mode1()` or `to_mode2()` would.

Usage

```
to_proximity(
  .data,
  similarity = .proj_measures,
  across = c("rows", "columns", "both"),
  dyad = NULL
)

to_correlation(.data, method = NULL)

to_cosine(.data)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package

- tidygraph `tbl_graph`, from the `{tidygraph}` package

`similarity` Method for establishing ties in the projection, "count" by default.

The measures are grouped below by what they are sensitive to. Within a group they are monotone transformations of one another, and so rank dyads identically, differing only in their scale. Choosing between the groups therefore matters rather more than choosing within one.

Measures defined for valued as well as binary data:

- "count" counts the coinciding ties, the cross-product XX'. For valued data it sums the products of tie strengths. Interpret it as the degree of opportunity between two nodes. This is the default, and the quickest, since it can use `igraph::bipartite_projection()` directly.
- "match" counts the cells in which two nodes hold exactly the same value, joint absences included. For binary data this is "rand" multiplied by the number of nodes in the other mode, but for valued data it is the more general measure, since it registers agreement at any tie strength. Use it where the level of involvement is meaningful in itself.
- "crossmin" sums the smaller of each pair of tie strengths. Interpret it as the capacity two nodes could jointly bring to bear. For binary data it reduces to "count".
- "overlap" divides "crossmin" by the smaller of the two nodes' total tie strength, the Szymkiewicz-Simpson or overlap coefficient, and so lies in $[0, 1]$ for non-negative data. Use it where one node is much more active than the other, and the less active node's rarity should not depress the score. For binary data it is the count over the smaller degree. xUCINET keeps that count for valued data too, where it can exceed 1 and is no longer the coefficient.
- "ruzicka" divides "crossmin" by the two nodes' combined tie strength less than shared, Ruzicka's weighted Jaccard coefficient, and so also lies in $[0, 1]$. For binary data it is "jaccard", so use it where "jaccard" is wanted but the tie strengths should be kept.
- "maxcrossmin" takes the largest such minimum rather than their sum, so that a single strong shared affiliation stands for the pair. For binary data it collapses to an indicator of any shared affiliation, and so is of little use there.
- "sqdiff" inverts the sum of squared differences in tie strength, as $1/(1 + d)$, so that identical rows give 1 and larger values mean more alike, as for every other measure here. Recover the raw sum of squared differences as $1/x - 1$.
- "pearson" gives Pearson's product-moment correlation and "covariance" its unstandardised counterpart. Use "covariance" where the variance in involvement is itself of interest, and "pearson" where it is not.
- "spearman" and "kendall" are the rank counterparts of "pearson". Use them where tie strengths order the affiliations reliably but their spacing does not, as with ordinal ratings.
- "cosine" gives the cosine of the angle between two nodes' rows. It differs from "pearson" in not centring them first, so that it reads two nodes as alike where their involvements are proportional rather than where they depart from the average in the same direction. For binary data it agrees with

"ochiai", except that a node with no ties at all is reported as no more similar to another than any other node, where "ochiai" would divide by zero.

- "euclidean" and "manhattan" invert the straight-line and the city-block distance between two rows, again as $1/(1+d)$. "manhattan" sums the absolute differences in tie strength, so that a large discrepancy on one affiliation counts no more than the same total spread over several; "euclidean" penalises the concentrated discrepancy more heavily, as "sqdiff" does. Recover either raw distance as $1/x - 1$.

Measures defined for binary data only, where a valued network is dichotomised at zero with a warning. Writing a for the cells in which both nodes are present, b and c for those in which just one is, and d for those in which neither is:

- Sensitive to co-presence, ignoring joint absence: "jaccard" is $a/(a+b+c)$, opportunity weighted by participation; "czekanowski" (the Dice or Sorensen coefficient) is $2a/(2a+b+c)$, which double-weights co-presence; and "sokalsneath" is $a/(a+2(b+c))$, which instead double-weights mismatch. Use these where joint non-participation says nothing, as in a sparse affiliation network with many events. For valued data, "ruzicka" is the counterpart of "jaccard" that keeps the tie strengths.
- Sensitive to matching, counting joint absence as evidence: "rand", the Simple Matching Coefficient, is $(a+d)/(a+b+c+d)$; "hamann" is $((a+d)-(b+c))/(a+b+c+d)$, the same quantity rescaled onto $[-1,1]$ so that its sign reports whether matches outnumber mismatches; and "rogerstanimoto" is $(a+d)/(a+2(b+c)+d)$, which double-weights mismatch. Interpret these as the degree of behavioural mirroring between two nodes, and use them where not attending is as informative as attending.
- Sensitive to association, through the odds ratio ad/bc: "yule" is Yule's Q, $(ad-bc)/(ad+bc)$, which has a straightforward reading as a rescaled odds ratio; and "bonacich" is $\sqrt{ad}/(\sqrt{ad}+\sqrt{bc})$, which is Yule's Y rescaled onto $[0,1]$. Both saturate at 1 wherever b or c is zero.
- Geometric: "ochiai" is $a/\sqrt{(a+b)(a+c)}$, the cosine of the angle between two nodes' affiliation vectors, and "ochiai2" (also known as Sokal and Sneath's fifth measure) is $ad/\sqrt{(a+b)(a+c)(d+b)(d+c)}$, its counterpart including joint absence. Neither is monotone in any of the above, so both are worth trying alongside them.
- "hamming" inverts the Hamming distance, the number of cells in which the two nodes differ, as $1/(1+d)$. It is a monotone transformation of "rand", and so ranks dyads identically, but states the disagreement as a count rather than as a proportion of agreement.

`across` Which margin of the network the operation is taken over. "rows" is the sending margin, so that each node is treated by the ties it sends; "columns" is the receiving margin; and "both" combines them. The two margins coincide for an undirected network, where all three options therefore agree. A two-mode network raises no such question, so "rows" and "columns" there are just the two nodesets.

Note that both the default and what "both" combines differ by function. `to_normalised()` defaults to "both", where it divides by the square root of the two denominators multiplied together, since that is the only rescaling that preserves a network's

symmetry. `to_proximity()` defaults to "rows", the conventional profile for structural equivalence, and there "both" compares each node's sent and received ties together.

`dyad` How the cells within the compared pair's own dyad are treated.

Four of the cells in any comparison are not like the others: i's tie to j, j's tie to i, and each node's self-tie. These cannot be compared where they lie the way the rest can. In column j, node i's entry is a tie to j, but node j's entry is its own self-tie: two different things in the same position. Two managers who happen to be tied to one another are not less structurally equivalent for it, but comparing those cells where they lie makes them look that way.

- "exclude" drops all four cells, so that nodes are compared only on their ties to third parties. This is the strictest reading of structural equivalence, and the default for an undirected network.
- "reciprocal" drops the self-ties, but compares i's tie to j against j's tie to i, so that a reciprocated pair counts as alike there. This is the default for a directed network.
- "complex" does the same, and additionally compares the two nodes' self-ties against each other. This is the default where the network has self-ties, since there they carry information.
- "include" compares every cell where it lies, with no special treatment. Use it only where the diagonal genuinely holds comparable values. It is also the quickest, since it needs no pairwise walk.

By default the appropriate treatment is chosen from the network's format, as described above.

A two-mode network has no dyad of this kind, so only `NULL` or "include" is accepted for one. Pass "include" for a profile matrix whose rows are nodes and whose columns are something else, such as a motif census. A square census with no dimnames otherwise reads as a one-mode network, and would have its cells dropped as if they were ties.

`method` How the diagonal and the reciprocal ties are treated. "all" includes all information, "diag" excludes the diagonal (self-ties), "recip" excludes the diagonal but compares pairs' reciprocal ties, and "complex" compares pairs' reciprocal ties and their self ties. By default the appropriate method is chosen based on the network format.

These correspond to `to_proximity()`'s `dyad` argument, except that "recip" and "complex" append each node's tie to the other to profiles otherwise taken over the columns. `to_proximity()` instead keeps the profile and the appended cells on the same margin. The two therefore agree for an undirected network but not for a directed one.

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods:

```
                data.frame default igraph matrix network stocnet tbl_graph
to_correlation                   *            *
```

```
to_cosine                    *           *
to_proximity          *      *     *     *      *      *        *
```

Value

An object of the same class as the function was given, modified as explained in the function description, details, or section. Functions that split a network return a list of such objects.

Node correlation

`to_correlation()` performs a Pearson pairwise correlation. It includes a switch: whereas for a two-mode network it performs a regular correlation, including all rows, for an undirected network it performs a correlation on a matrix with the diagonals removed, for a reciprocated network it includes the difference between reciprocated ties, and for complex networks it includes also the difference between the self ties in each pairwise calculation. This function runs in $O(mn^2)$ complexity.

Like the other functions here, it returns the class of object it was given, so that the correlations between the nodes of a network are returned as a weighted network of the same class. The correlations are calculated on a matrix, however, so a network is coerced to one and the result coerced back. Passing a matrix, e.g. `to_correlation(as_matrix(.data))`, spares both, and returns the matrix of correlations that further calculation most often wants.

Node cosine

`to_cosine()` takes the cosine of the angle between each pair of the matrix's columns, without projecting it. It is `to_proximity()`'s "cosine" measure over the columns rather than the rows, and so accepts a two-mode network, where it describes the second mode.

References

On structural equivalence:

Lorrain, Francois, and Harrison C. White. 1971. "Structural equivalence of individuals in social networks". *The Journal of Mathematical Sociology* 1(1): 49-80. doi:10.1080/0022250X.1971.9989788

See Also

`to_mode1()`, which applies the same measures to a two-mode network, comparing nodes on their affiliations rather than on their ties.

Other modifications: `modif_backbone`, `modif_concepts`, `modif_direction`, `modif_from_parts`, `modif_from_ternary`, `modif_labels`, `modif_levels`, `modif_miss`, `modif_motifs`, `modif_paths`, `modif_permutation`, `modif_plexity`, `modif_project`, `modif_scope`, `modif_split`, `modif_weight`

Examples

```
to_proximity(ison_algebra, "pearson")
to_proximity(ison_adolescents, "jaccard")
to_proximity(ison_southern_women, "ruzicka", across = "columns")
```

modif_scope *Modifying networks scope*

Description

These functions offer tools for transforming manynet-consistent objects (matrices, igraph, tidygraph, or network objects). Transforming means that the returned object may have different dimensions than the original object.

- `to_ego()` scopes a network into the local neighbourhood of a given node.
- `to_component()` scopes a network to a single one of its components, either the `component`th largest or the one containing a named node. It is the singular counterpart of `to_components()`, which returns a list of all of them, largest first.
- `to_giant()` scopes a network into one including only the main component and no smaller components or isolates. It is a wrapper, such that `to_giant(.data)` is `to_component(.data, 1)`.
- `to_time()` scopes a network to the network as it stood at a given moment, in whichever of the ways set out in the Time section it records time. `to_wave()` is an alias, using the wave-based vocabulary of `net_waves()` and `to_waves()`. For one network per moment, see `to_times()`.
- `to_reporter()` scopes a cognitive social structure to the report of one of its reporters. For one network per reporter, see `to_reporters()`.
- `to_subgraph()` scopes a network into a subgraph by filtering on some node-related logical statement.
- `to_blockmodel()` reduces a network to the ties between the blocks of a given partition membership vector.

Usage

```
to_ego(.data, node, max_dist = 1, min_dist = 0, direction = c("out", "in"))

to_time(.data, time = NULL)

to_wave(.data, time = NULL)

to_reporter(.data, reporter)

to_component(.data, component = 1, connectivity = c("weak", "strong"))

to_giant(.data, connectivity = c("weak", "strong"))

to_subgraph(.data, ...)

to_blockmodel(.data, membership, FUN = mean)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`node` Name or index of node.

`max_dist` The maximum breadth of the neighbourhood. By default 1.

`min_dist` The minimum breadth of the neighbourhood. By default 0. Increasing this to 1 excludes the ego, and 2 excludes ego's direct alters.

`direction` Character string, "out" bases the measure on outgoing ties, "in" on incoming ties, and "all" on either/the sum of the two. By default "all".

`time` A moment at which to present the network. A moment beyond the last one the network records reverts to the last, except in an interval network, which is defined between the moments it records as well as at them.

`reporter` Label or index of the node whose report on the network is to be retained, in a cognitive social structure.

`component` Which component to retain. By default 1, i.e. the largest (giant) component, with 2 the second largest, and so on. Alternatively, the name of a node, in which case the component containing that node is retained.

`connectivity` Character string, "weak" treats a directed network's components as if the network were undirected, and "strong" requires ties in both directions between members. This is ignored for undirected networks, where the two notions coincide. Note that the default differs by function: marks that assert connectedness default to "strong", while functions that scope or split a network into components default to "weak".

`...` Arguments passed on to dplyr::filter

`membership` A vector of partition memberships. For two-mode networks this is a single vector covering the nodes in both modes, and not one vector per mode; the blocks of each mode are established separately from it, so that the result has one row per block of the first mode and one column per block of the second.

`FUN` A function for summarising block content. By default `mean`. Other recommended options include `median`, `sum`, `min` or `max`.

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods:

```
              array data.frame default igraph matrix network stocnet tbl_graph
to_blockmodel                 *       *      *      *       *                *
to_component                  *       *      *      *       *       *        *
```

```
to_ego                                  *      *                      *        *
to_egos                        *        *      *      *      *       *        *
to_reporter       *                     *                             *
to_subgraph                    *        *      *      *      *       *        *
to_subgraphs                            *      *             *       *        *
```

Value

An object of the same class as the function was given, modified as explained in the function description, details, or section. Functions that split a network return a list of such objects.

Time

Two things about how a network records time are independent of each other, and the network is scoped to a moment by both of them.

How a moment is *represented* can be read from the ties. A network either stamps each tie with the point it was recorded at, in a 'time' column, or states the interval each tie lasts over, in 'begin' and 'end' columns.

How a moment *relates to the one before it* cannot be read from the ties, and the network declares it in `info$update`. Where this is "replace", which it is by default, each moment re-states the ties, so the network at a moment is the ties stamped with it. Where it is "increment", each row is a change to a tie's value, so the network at a moment is every row up to and including it, accumulated, and a tie that has accumulated to zero is no longer a tie. An interval tie carries its own lifespan, so nothing is declared about it: it is active at a moment where it began at or before that moment and has not yet ended (`begin <= time < end`, the half-open convention shared with `network::networkDynamic`, so a tie that ends and one that begins at the same instant do not overlap). A tie with no 'end' is right-censored, and active from its beginning onwards.

`info$observation` records a third thing, how densely the network is observed: a "panel" of a few complete re-observations, or a stream of many "event" records. This describes a network rather than scoping it. The two go together without entailing each other: a panel re-observes the whole network at each wave and so is usually "replace", and an event stream is often "increment", but an event that states a value afresh is "replace" too. `is_longitudinal()` marks the first, `is_dynamic()` the second, and no network is both.

In every case the nodal changes recorded up to the moment are applied, and a layer that states something holding throughout is carried into whichever moment is asked for, since such a layer is a constant covariate rather than an observation of that moment. A layer holds throughout where the network declares it "cross-sectional", or where it records that layer at a single moment while another layer spans several.

`to_blockmodel()`

Reduced graphs provide summary representations of network structures by collapsing groups of connected nodes into single nodes while preserving the topology of the original structures. Like the other scoping functions, the reduced graph is returned in the same class as the input: a blockmodel matrix of summarised block content for matrix input, and the corresponding weighted network otherwise. Memberships are usually obtained from one of the `node_in_*()` functions in `{netrics}`.

See Also

to_times() for one network per moment, and net_times() for how many moments there are to ask for.

Other modifications: modif_backbone, modif_concepts, modif_direction, modif_from_parts, modif_from_ternary, modif_labels, modif_levels, modif_miss, modif_motifs, modif_paths, modif_permutation, modif_plexity, modif_project, modif_proximity, modif_split, modif_weight

Examples

```
  # The ties a panel observed in a given wave:
  to_time(ison_monks, 2)
  # The ties an interval network held in a given year:
  to_time(irps_wwi, 1901)
  # The state an event network had accumulated to by a given day:
  to_time(irps_nuclear, as.Date("2011-04-01"))
css <- as_stocnet(array(rbinom(27, 1, 0.5), c(3, 3, 3)), attribute = "by")
to_reporter(css, 2)
  to_component(fict_greys, 2)
  to_component(fict_greys, "Miranda Bailey")
  to_giant(fict_greys)
```

---

modif_split *Splitting networks into lists*

Description

These functions offer tools for splitting manynet-consistent objects (matrices, igraph, tidygraph, or network objects) into lists of networks.

- to_egos() splits a network into ego (or focal) networks.
- to_subgraphs() splits a network into subgraphs on some given node attribute.
- to_layers() splits a multiplex network into its layers, i.e. a list of uniplex networks, one per tie type. Use to_uniplex(), or its alias to_layer(), to retain just one of them.
- to_reporters() splits a cognitive social structure into the report of each of its reporters, one network per reporter. Use to_reporter() to retain just one of them, and from_reporters() to join them again.
- to_components() splits a network into its components, ordered from the largest to the smallest. Use to_component() to retain just one of them.
- to_times() splits a network into the network as it stood at each moment it records, however it records time. This is where new work on splitting a network by time belongs; to_waves() and to_slices() are the older, form-specific spellings.
- to_waves() splits a panel network into a list of its waves.
- to_slices() splits a network that increments its ties into the state it had accumulated to at each of the given time slice(s).

Usage

```
to_egos(.data, max_dist = 1, min_dist = 0, direction = c("out", "in"))

to_subgraphs(.data, attribute)

to_layers(.data)

to_reporters(.data)

to_components(.data, connectivity = c("weak", "strong"))

to_waves(.data, attribute = "wave", panels = NULL, cumulative = FALSE)

to_times(.data, times = NULL)

to_slices(.data, attribute = "time", slice = NULL)
```

Arguments

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

`max_dist` The maximum breadth of the neighbourhood. By default 1.

`min_dist` The minimum breadth of the neighbourhood. By default 0. Increasing this to 1 excludes the ego, and 2 excludes ego's direct alters.

`direction` Character string, "out" bases the measure on outgoing ties, "in" on incoming ties, and "all" on either/the sum of the two. By default "all".

`attribute` One or two attributes used to slice data.

`connectivity` Character string, "weak" treats a directed network's components as if the network were undirected, and "strong" requires ties in both directions between members. This is ignored for undirected networks, where the two notions coincide. Note that the default differs by function: marks that assert connectedness default to "strong", while functions that scope or split a network into components default to "weak".

`panels` Would you like to select certain waves? NULL by default. That is, a list of networks for every available wave is returned. Users can also list specific waves they want to select.

`cumulative` Whether to make wave ties cumulative. FALSE by default. That is, each wave is treated isolated.

`times` The moments to return the network at. By default `NULL`, in which case every moment the network records is returned, as `net_times()` counts them.

`slice` Character string or character list indicating the date(s) or integer(s) range used to slice data (e.g slice = c(1:2, 3:4)).

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods:

```
               array data.frame default diff_model igraph matrix network stocnet
to_components                *       *                *      *       *       *
to_egos                      *       *                *      *       *       *
to_layers                    *       *                *              *
to_reporters      *                  *                                       *
to_slices                            *                *
to_subgraphs                         *                *              *       *
to_times                             *                *                      *
to_waves                     *       *          *     *
               tbl_graph
to_components          *
to_egos                *
to_layers              *
to_reporters
to_slices              *
to_subgraphs           *
to_times               *
to_waves               *
```

`to_times()` returns the network as it stood at each moment it records, whichever way it records time, by calling `to_time()` on each in turn. `to_waves()` and `to_slices()` are the older, form-specific spellings: `to_waves()` splits a panel by its waves, and `to_slices()` accumulates an event network up to each of its moments. Unlike them, `to_times()` always returns a list, named by the moments and ordered by them, even where the network records only one, so that `net_times(.data)` and `length(to_times(.data))` always agree.

Value

An object of the same class as the function was given, modified as explained in the function description, details, or section. Functions that split a network return a list of such objects.

`to_layers()`

The layers of a multiplex network are held in a tie attribute, `type` in tidygraph/igraph objects and `layer` in 'stocnet' objects. Each layer is extracted by `to_uniplex()`, so that the layers returned here are the same networks as retrieving them one at a time, and the returned list is named by the tie types found in the network. Where a network holds no tie types it is already uniplex, and a list of length one is returned.

`to_reporters()`

Every node of a cognitive social structure is a reporter, so the list holds one network for each node, named by its label, including the reports of nodes that reported no ties. A reporter that did not report at all gives a network whose ties are all missing. Ties that no node reported, such as a layer taken from records, belong to no report and are left out. A network that names no reporter is returned as a list of length one, as `to_layers()` returns a network that holds no layers. Egocentric data cannot be split this way, since each ego reports on its own alters; see `to_egos()`.

See Also

Other modifications: `modif_backbone`, `modif_concepts`, `modif_direction`, `modif_from_parts`, `modif_from_ternary`, `modif_labels`, `modif_levels`, `modif_miss`, `modif_motifs`, `modif_paths`, `modif_permutation`, `modif_plexity`, `modif_project`, `modif_proximity`, `modif_scope`, `modif_weight`

Examples

```
  to_egos(ison_adolescents)
  # graphs(to_egos(ison_adolescents,2))
ison_adolescents |>
  mutate(unicorn = sample(c("yes", "no"), 8,
                          replace = TRUE)) |>
  to_subgraphs(attribute = "unicorn")
as_tidygraph(create_filled(5)) |>
  mutate_ties(type = sample(c("friend", "enemy"), 10, replace = TRUE)) |>
  to_layers()
to_reporters(as_stocnet(array(rbinom(27, 1, 0.5), c(3, 3, 3)),
                        attribute = "by"))
  to_components(to_uniplex(fict_marvel, "relationship"))
  # Strong decomposition of a directed network returns many small components,
  # ordered here from largest to smallest, so just the largest is shown:
  to_components(fict_starwars, connectivity = "strong")[[1]]
ison_adolescents |>
  mutate_ties(wave = sample(1995:1998, 10, replace = TRUE)) |>
  to_waves(attribute = "wave")
  length(to_times(irps_wwi))
  to_times(ison_tailorshop)
ison_adolescents |>
  mutate_ties(time = 1:10, increment = 1) |>
  add_ties(c(1,2), list(time = 3, increment = -1)) |>
  to_slices(slice = 7)
```

`modif_weight` *Modifying tie weight formats*

Description

These functions reformat tie attributes like their weight or sign:

- `to_unweighted()` reformats weighted network data to unweighted network data, with all tie weights removed.
- `to_unsigned()` reformats signed network data to unsigned network data, keeping just the "positive" or the "negative" ties, or "both", which keeps every tie but replaces its sign with its magnitude.
- `to_positive()` keeps just the positive ties of a signed network, and returns an unsigned network unaltered. It is a guard for the functions that cannot read a negative tie, such as the path-based measures, for which a negative tie is hostility rather than a channel.
- `to_normalised()` rescales tie weights relative to the other ties of the same node, so that a value reads as a share rather than a count.

If the format condition is not met, for example `to_undirected()` is used on a network that is already undirected, the network data is returned unaltered. No warning is given so that these functions can be used to ensure conformance.

Unlike the `as_*()` group of functions, these functions always return the same class as they are given, only transforming these objects' properties.

Usage

```
to_unsigned(.data, keep = c("positive", "negative", "both"))

to_positive(.data)

to_unweighted(.data, threshold = 1)

to_signed(.data, mark = NULL)

to_weighted(.data, measure = NULL)

to_normalised(
  .data,
  rule = c("max", "mean", "sum"),
  across = c("both", "rows", "columns")
)

to_normalized(
  .data,
  rule = c("max", "mean", "sum"),
  across = c("both", "rows", "columns")
)
```

Arguments

`.data` An object of a {`manynet`}-consistent class:

- adjacency or incidence `matrix` from {`base`} R
- edgelist `data.frame` from {`base`} R or `tbl/tbl_df` from {`tibble`}
- stocnet `stocnet`, from the {`manynet`} package

- igraph `igraph`, from the `{igraph}` package
- network `network`, from the `{network}` package
- tidygraph `tbl_graph`, from the `{tidygraph}` package

`keep` In the case of a signed network, whether to retain the "positive" or the "negative" ties, or "both", which retains every tie but replaces its sign with its magnitude.

`threshold` For a matrix, the threshold to binarise/dichotomise at.

`mark` A mark (logical vector) the length of the ties in the network.

`measure` A numeric vector (measure) that will be added as the tie weights to the network. If this is NULL, then the tie weights will be drawn from a Poisson distribution with $\lambda = 4$.

`rule` How each tie value is rescaled, relative to the other values recorded for the same node.

- "max" (the default) divides by the largest of them, so that a node's strongest tie is 1 and its others are read against that.
- "mean" divides by the average of them, counting every dyad and not just those tied, so that 1 marks a tie of typical strength.
- "sum" divides by the total of them, so that they add to 1 and each reads as the share of the node's ties that goes to that partner.

Missing values propagate rather than being ignored, so that a node with an unobserved tie has all its values unobserved. Use `impute_ties()` first to state a different assumption.

`across` Which margin of the network the operation is taken over. "rows" is the sending margin, so that each node is treated by the ties it sends; "columns" is the receiving margin; and "both" combines them. The two margins coincide for an undirected network, where all three options therefore agree. A two-mode network raises no such question, so "rows" and "columns" there are just the two nodesets.

Note that both the default and what "both" combines differ by function. `to_normalised()` defaults to "both", where it divides by the square root of the two denominators multiplied together, since that is the only rescaling that preserves a network's symmetry. `to_proximity()` defaults to "rows", the conventional profile for structural equivalence, and there "both" compares each node's sent and received ties together.

Details

Not all functions have methods available for all object classes. Below are the currently implemented S3 methods:

	data.frame	default	igraph	matrix	network	stocnet	tbl_graph
to_normalised	*	*	*	*	*	*	*
to_signed	*	*	*	*	*	*	*
to_unsigned	*	*	*	*	*	*	*
to_unweighted	*	*	*	*	*	*	*
to_weighted		*	*		*		*

`to_normalised()` divides by the sending node's denominator for "rows", so that a value says what share of i's ties goes to j, and by the receiving node's denominator for "columns".

Rescaling a one-mode network across its rows or columns makes it asymmetric, since what i sends j is generally not what j sends i. Where such a network is undirected, each tie is therefore split into two, and the network is returned directed.

Value

An object of the same class as the function was given, modified as explained in the function description, details, or section. Functions that split a network return a list of such objects.

See Also

Other ties: `manip_ties_attr`, `manip_ties_num`, `modif_backbone`, `modif_direction`

Other modifications: `modif_backbone`, `modif_concepts`, `modif_direction`, `modif_from_parts`, `modif_from_ternary`, `modif_labels`, `modif_levels`, `modif_miss`, `modif_motifs`, `modif_paths`, `modif_permutation`, `modif_plexity`, `modif_project`, `modif_proximity`, `modif_scope`, `modif_split`

Examples

```
marvel <- to_uniplex(fict_marvel, "relationship")
to_unsigned(marvel, "positive")
to_unsigned(marvel, "both")
to_positive(marvel)
to_normalised(ison_networkers, rule = "sum", across = "rows")
```

`progress` *Sequence and progress functions*

Description

These functions wrap `{cli}` functions and elements to build an attractive command line interface (CLI).

- `snet_progress_step()` for progress steps.
- `snet_progress_along()` for progress along a vector.
- `snet_progress_seq()` for progress along a sequence.
- `snet_progress_nodes()` for progress along the nodes of a network.

A progress bar reassures a user that a long task is running, so it shows from `options(snet_verbosity = 'verbose')` upwards. See the verbosity section of interface.

Usage

```
snet_progress_step(..., .envir = parent.frame())

snet_progress_along(..., .envir = parent.frame())

snet_progress_seq(..., .envir = parent.frame())

snet_progress_nodes(..., .envir = parent.frame())

seq_nodes(.data)

seq_ties(.data)
```

Arguments

`...` One or more character strings. For most of these functions, if multiple strings are passed these will be pasted together.

`.envir` This argument is just to inherit the parent frame in the (likely) event that the function is used within another function.

`.data` An object of a {manynet}-consistent class:

- adjacency or incidence `matrix` from {base} R
- edgelist `data.frame` from {base} R or `tbl/tbl_df` from {tibble}
- stocnet `stocnet`, from the {manynet} package
- igraph `igraph`, from the {igraph} package
- network `network`, from the {network} package
- tidygraph `tbl_graph`, from the {tidygraph} package

Index

∗ **Galois lattice**
modif_concepts, 147
∗ **attributes**
member_names, 143
∗ **changes**
manip_changes, 111
mark_format_change, 128
∗ **coercions**
coerce_graph, 5
coerce_list, 9
coerce_matrix, 12
∗ **concept lattice**
modif_concepts, 147
∗ **datasets**
fict_actually, 14
fict_friends, 16
fict_greys, 17
fict_lotr, 18
fict_marvel, 19
fict_potter, 21
fict_starwars, 22
fict_thrones, 24
irps_911, 27
irps_blogs, 28
irps_books, 30
irps_corruption, 31
irps_nuclear, 32
irps_revere, 34
irps_supremecourt, 35
irps_tribes, 36
irps_usgeo, 38
irps_wwi, 39
ison_adolescents, 40
ison_algebra, 41
ison_bankwiring, 42
ison_brandes, 44
ison_classmates, 45
ison_dolphins, 47
ison_emotions, 48
ison_florentine, 49
ison_fraternity, 51
ison_hightech, 52
ison_judo_moves, 54
ison_karateka, 55
ison_koenigsberg, 57
ison_laterals, 58
ison_lawfirm, 60
ison_monks, 62
ison_networkers, 64
ison_physicians, 65
ison_southern_women, 68
ison_tailorshop, 70
∗ **diffusion**
make_play, 91
∗ **formal concept analysis**
modif_concepts, 147
∗ **global**
manip_globals, 113
∗ **info**
manip_info, 115
∗ **makes**
make_collect, 71
make_create, 73
make_ego, 77
make_emails, 78
make_explicit, 80
make_groups, 80
make_growth, 85
make_learning, 87
make_play, 91
make_random, 94
make_read, 98
make_write, 108
∗ **manipulations**
manip_changes, 111
manip_globals, 113
manip_info, 115
manip_nodes_attr, 117

manip_nodes_num, 119
manip_ties_attr, 121
manip_ties_num, 123
∗ **marking**
mark_features, 125
mark_is, 135
∗ **marks**
mark_format_change, 128
mark_format_cognitive, 129
mark_format_node, 131
mark_format_tie, 133
∗ **measures**
measure_attributes_nodes, 136
measure_attributes_ties, 137
measure_dims, 140
∗ **missingness**
measure_missingness, 142
∗ **models**
make_learning, 87
make_play, 91
∗ **modifications**
modif_backbone, 144
modif_concepts, 147
modif_direction, 149
modif_from_parts, 151
modif_from_ternary, 152
modif_labels, 155
modif_levels, 156
modif_miss, 157
modif_motifs, 160
modif_paths, 162
modif_permutation, 165
modif_plexity, 166
modif_project, 169
modif_proximity, 175
modif_scope, 180
modif_split, 183
modif_weight, 186
∗ **nodes**
manip_nodes_attr, 117
manip_nodes_num, 119
∗ **projections**
modif_concepts, 147
modif_project, 169
∗ **ties**
manip_ties_attr, 121
manip_ties_num, 123
modif_backbone, 144
modif_direction, 149
modif_weight, 186

add_info *(manip_info)*, 115
add_node_attribute *(manip_nodes_attr)*, 117
add_nodes *(manip_nodes_num)*, 119
add_tie_attribute *(manip_ties_attr)*, 121
add_ties *(manip_ties_num)*, 123
arrange_changes *(manip_changes)*, 111
arrange_globals *(manip_globals)*, 113
arrange_nodes *(manip_nodes_num)*, 119
arrange_ties *(manip_ties_attr)*, 121
as, *76*, *101*, *110*
as_changelist *(coerce_list)*, 9
as_changelist(), *105*
as_diffnet *(coerce_graph)*, 5
as_diffusion *(coerce_graph)*, 5
as_edgelist *(coerce_list)*, 9
as_globallist *(coerce_list)*, 9
as_graphAM *(coerce_graph)*, 5
as_hypergraph *(coerce_graph)*, 5
as_igraph *(coerce_graph)*, 5
as_infolist *(coerce_list)*, 9
as_matrix *(coerce_matrix)*, 12
as_missinglist *(coerce_list)*, 9
as_missinglist(), *104*, *107*, *142*
as_network *(coerce_graph)*, 5
as_nodelist *(coerce_list)*, 9
as_siena *(coerce_graph)*, 5
as_stocnet *(coerce_graph)*, 5
as_tidygraph *(coerce_graph)*, 5

bind_changes *(manip_changes)*, 111
bind_globals *(manip_globals)*, 113
bind_node_attributes *(manip_nodes_attr)*, 117
bind_nodes *(manip_nodes_num)*, 119
bind_ties *(manip_ties_num)*, 123

class_describe, 4
clear_glossary *(glossary)*, 25
coerce_graph, 5, *11*, *13*
coerce_list, *8*, 9, *13*
coerce_matrix, *8*, *11*, 12
collect_cran *(make_collect)*, 71
collect_ego *(make_ego)*, 77
collect_ego(), *79*
collect_emails *(make_emails)*, 78

collect_pkg (make_collect), 71
create_components (make_create), 73
create_core (make_create), 73
create_cycle (make_create), 73
create_degree (make_create), 73
create_empty (make_create), 73
create_explicit (make_explicit), 80
create_filled (make_create), 73
create_lattice (make_create), 73
create_motifs (modif_motifs), 160
create_ring (make_create), 73
create_star (make_create), 73
create_tree (make_create), 73
create_wheel (make_create), 73
create_windmill (make_create), 73

data_overview, 14
delete_changes (manip_changes), 111
delete_globals (manip_globals), 113
delete_incomplete (manip_nodes_num), 119
delete_isolates (manip_nodes_num), 119
delete_isolates(), *72, 73*
delete_node_attribute (manip_nodes_attr), 117
delete_nodes (manip_nodes_num), 119
delete_tie_attribute (manip_ties_attr), 121
delete_ties (manip_ties_num), 123
describe_changes (class_describe), 4
describe_network (class_describe), 4
describe_nodes (class_describe), 4
describe_ties (class_describe), 4
describe_transformations (class_describe), 4

fict_actually, 14
fict_friends, 16
fict_greys, 17
fict_lotr, 18
fict_marvel, 19
fict_potter, 21
fict_starwars, 22
fict_thrones, 24
filter_changes (manip_changes), 111
filter_globals (manip_globals), 113
filter_nodes (manip_nodes_num), 119
filter_ties (manip_ties_num), 123
from_egos (modif_from_parts), 151
from_egos(), *153*
from_layers (modif_from_ternary), 152
from_reporters (modif_from_ternary), 152
from_slices (modif_from_ternary), 152
from_subgraphs (modif_from_parts), 151
from_subgraphs(), *153*
from_ties (modif_from_ternary), 152
from_times (modif_from_ternary), 152
from_times(), *152*
from_waves (modif_from_ternary), 152

generate_citations (make_growth), 85
generate_communities (make_groups), 80
generate_configuration (make_random), 94
generate_core (make_groups), 80
generate_degree (make_random), 94
generate_fire (make_growth), 85
generate_islands (make_groups), 80
generate_man (make_random), 94
generate_random (make_random), 94
generate_scalefree (make_growth), 85
generate_smallworld (make_groups), 80
generate_utilities (make_random), 94
gloss (glossary), 25
glossary, 25

impute_nodes (modif_miss), 157
impute_nodes(), *142*
impute_ties (modif_miss), 157
impute_ties(), *142*
interface, 26, *189*
irps_911, 27
irps_blogs, 28
irps_books, 30
irps_corruption, 31
irps_nuclear, 32
irps_revere, 34
irps_supremecourt, 35
irps_tribes, 36
irps_usgeo, 38
irps_wwi, 39
is_acyclic (mark_features), 125
is_aperiodic (mark_features), 125
is_attributed (mark_format_node), 131
is_changing (mark_format_change), 128
is_cognitive (mark_format_cognitive), 129
is_cognitive(), *100*, *153*
is_complex (mark_format_tie), 133
is_connected (mark_features), 125

is_directed (mark_format_tie), 133
is_disaggregated (mark_format_tie), 133
is_dynamic (mark_format_change), 128
is_dynamic(), *100*
is_edgelist (mark_is), 135
is_egocentric (mark_format_cognitive), 129
is_egocentric(), *79*, *99*, *100*
is_egolist (mark_format_node), 131
is_eulerian (mark_features), 125
is_gossip (mark_format_cognitive), 129
is_graph (mark_is), 135
is_hypergraph (mark_format_node), 131
is_hypergraph(), *8*, *174*
is_labelled (mark_format_node), 131
is_list (mark_is), 135
is_longitudinal (mark_format_change), 128
is_longitudinal(), *140*
is_manynet (mark_is), 135
is_multilevel (mark_format_node), 131
is_multiplex (mark_format_tie), 133
is_perfect_matching (mark_features), 125
is_signed (mark_format_tie), 133
is_spatial (mark_format_node), 131
is_spatial(), *100*, *110*
is_twomode (mark_format_node), 131
is_uniplex (mark_format_tie), 133
is_weighted (mark_format_tie), 133
ison_adolescents, 40
ison_algebra, 41
ison_bankwiring, 42
ison_brandes, 44
ison_classmates, 45
ison_dolphins, 47
ison_emotions, 48
ison_florentine, 49
ison_fraternity, 51
ison_hightech, 52
ison_judo_moves, 54
ison_karateka, 55
ison_koenigsberg, 57
ison_laterals, 58
ison_lawfirm, 60
ison_monks, 62
ison_networkers, 64
ison_physicians, 65
ison_southern_women, 68
ison_tailorshop, 70

join_nodes (manip_nodes_attr), 117
join_ties (manip_ties_attr), 121

layer_is_directed (member_names), 143
layer_names (member_names), 143
layer_ties (measure_dims), 140

make_collect, 71, *76*, *77*, *79*, *80*, *84*, *87*, *89*, *93*, *97*, *101*, *110*
make_create, *73*, 73, *77*, *79*, *80*, *84*, *87*, *89*, *93*, *97*, *101*, *110*
make_ego, *73*, *76*, 77, *79*, *80*, *84*, *87*, *89*, *93*, *97*, *101*, *110*
make_emails, *73*, *76*, *77*, 78, *80*, *84*, *87*, *89*, *93*, *97*, *101*, *110*
make_explicit, *73*, *76*, *77*, *79*, 80, *84*, *87*, *89*, *93*, *97*, *101*, *110*
make_groups, *73*, *76*, *77*, *79*, *80*, 80, *87*, *89*, *93*, *97*, *101*, *110*
make_growth, *73*, *76*, *77*, *79*, *80*, *84*, 85, *89*, *93*, *97*, *101*, *110*
make_learning, *73*, *76*, *77*, *79*, *80*, *84*, *87*, 87, *93*, *97*, *101*, *110*
make_mnet, 89
make_play, *73*, *76*, *77*, *79*, *80*, *84*, *87*, *89*, 91, *97*, *101*, *110*
make_random, *73*, *76*, *77*, *79*, *80*, *84*, *87*, *89*, *93*, 94, *101*, *110*
make_read, *73*, *76*, *77*, *79*, *80*, *84*, *87*, *89*, *93*, *97*, 98, *110*
make_stocnet, 101
make_stocnet(), *8*, *100*, *101*, *174*
make_write, *73*, *76*, *77*, *79*, *80*, *84*, *87*, *89*, *93*, *97*, *101*, 108
manip_changes, 111, *114*, *117*, *119*, *121*, *123*, *125*, *129*
manip_globals, *112*, 113, *117*, *119*, *121*, *123*, *125*
manip_info, *112*, *114*, 115, *119*, *121*, *123*, *125*
manip_nodes_attr, *112*, *114*, *117*, 117, *121*, *123*, *125*
manip_nodes_num, *112*, *114*, *117*, *119*, 119, *123*, *125*
manip_ties_attr, *112*, *114*, *117*, *119*, *121*, 121, *125*, *147*, *151*, *189*
manip_ties_num, *112*, *114*, *117*, *119*, *121*, *123*, 123, *147*, *151*, *189*

mark_features, 125, *136*
mark_format_change, *112*, 128, *130*, *132*, *134*
mark_format_cognitive, *129*, 129, *132*, *134*
mark_format_node, *129, 130*, 131, *134*
mark_format_tie, *129, 130*, *132*, 133
mark_is, *127*, 135
measure_attributes_nodes, 136, *140, 141*
measure_attributes_ties, *137*, 137, *141*
measure_dims, *137*, *140*, 140
measure_missingness, 142
member_names, 143
mode_names *(*member_names*)*, 143
mode_nodes *(*measure_dims*)*, 140
modif_backbone, *123*, *125*, 144, *149*, *151, 152*, *154*, *156, 157*, *159*, *161*, *165, 166*, *169*, *174*, *179*, *183*, *186*, *189*
modif_concepts, *147*, 147, *151, 152*, *154*, *156, 157*, *159*, *161*, *165, 166*, *169*, *174*, *179*, *183*, *186*, *189*
modif_direction, *123*, *125*, *147*, *149*, 149, *152*, *154*, *156, 157*, *159*, *161*, *165, 166*, *169*, *174*, *179*, *183*, *186*, *189*
modif_from_parts, *147*, *149*, *151*, 151, *154*, *156, 157*, *159*, *161*, *165, 166*, *169*, *174*, *179*, *183*, *186*, *189*
modif_from_ternary, *147*, *149*, *151, 152*, 152, *156, 157*, *159*, *161*, *165, 166*, *169*, *174*, *179*, *183*, *186*, *189*
modif_labels, *147*, *149*, *151, 152*, *154*, 155, *157*, *159*, *161*, *165, 166*, *169*, *174*, *179*, *183*, *186*, *189*
modif_levels, *147*, *149*, *151, 152*, *154*, *156*, 156, *159*, *161*, *165, 166*, *169*, *174*, *179*, *183*, *186*, *189*
modif_miss, *147*, *149*, *151, 152*, *154*, *156, 157*, 157, *161*, *165, 166*, *169*, *174*, *179*, *183*, *186*, *189*
modif_motifs, *147*, *149*, *151, 152*, *154*, *156, 157*, *159*, 160, *165, 166*, *169*, *174*, *179*, *183*, *186*, *189*
modif_paths, *147*, *149*, *151, 152*, *154*, *156, 157*, *159*, *161*, 162, *166*, *169*, *174*, *179*, *183*, *186*, *189*
modif_permutation, *147*, *149*, *151, 152*, *154*, *156, 157*, *159*, *161*, *165*, 165, *169*, *174*, *179*, *183*, *186*, *189*
modif_plexity, *147*, *149*, *151, 152*, *154*, *156, 157*, *159*, *161*, *165, 166*, 166, *174*, *179*, *183*, *186*, *189*
modif_project, *147*, *149*, *151, 152*, *154*, *156, 157*, *159*, *161*, *165, 166*, *169*, 169, *179*, *183*, *186*, *189*
modif_proximity, *147*, *149*, *151, 152*, *154*, *156, 157*, *159*, *161*, *165, 166*, *169*, *174*, 175, *183*, *186*, *189*
modif_scope, *147*, *149*, *151, 152*, *154*, *156, 157*, *159*, *161*, *165, 166*, *169*, *174*, *179*, 180, *186*, *189*
modif_split, *147*, *149*, *151, 152*, *154*, *156, 157*, *159*, *161*, *165, 166*, *169*, *174*, *179*, *183*, 183, *189*
modif_weight, *123*, *125*, *147*, *149*, *151, 152*, *154*, *156, 157*, *159*, *161*, *165, 166*, *169*, *174*, *179*, *183*, *186*, 186
mutate *(*manip_nodes_attr*)*, 117
mutate_changes *(*manip_changes*)*, 111
mutate_globals *(*manip_globals*)*, 113
mutate_info *(*manip_info*)*, 115
mutate_nodes *(*manip_nodes_attr*)*, 117
mutate_ties *(*manip_ties_attr*)*, 121

net_attributes *(*manip_info*)*, 115
net_dims *(*measure_dims*)*, 140
net_layers *(*measure_dims*)*, 140
net_modes *(*measure_dims*)*, 140
net_name *(*member_names*)*, 143
net_node_attributes *(*member_names*)*, 143
net_node_incomplete *(*measure_missingness*)*, 142
net_node_missing *(*measure_missingness*)*, 142
net_nodes *(*measure_dims*)*, 140
net_tie_attributes *(*member_names*)*, 143
net_tie_incomplete *(*measure_missingness*)*, 142
net_tie_missing *(*measure_missingness*)*, 142
net_ties *(*measure_dims*)*, 140
net_times *(*measure_dims*)*, 140
net_times(), *183*
net_waves *(*measure_dims*)*, 140
node_attribute *(*measure_attributes_nodes*)*, 136
node_is_mode *(*measure_attributes_nodes*)*, 136

node_labels (measure_attributes_nodes), 136
node_names (measure_attributes_nodes), 136

play_diffusion (make_play), 91
play_learning (make_learning), 87
play_segregation (make_learning), 87
print.mnet (make_mnet), 89
print.stocnet (make_stocnet), 101
print_all (make_mnet), 89
print_glossary (glossary), 25
progress, 189

read_dynetml (make_read), 98
read_edgelist (make_read), 98
read_gdf (make_read), 98
read_geojson (make_read), 98
read_gexf (make_read), 98
read_gml (make_read), 98
read_graphml (make_read), 98
read_hif (make_read), 98
read_hif(), *106*
read_matrix (make_read), 98
read_nodelist (make_read), 98
read_pajek (make_read), 98
read_ucinet (make_read), 98
rename_changes (manip_changes), 111
rename_globals (manip_globals), 113
rename_nodes (manip_nodes_attr), 117
rename_ties (manip_ties_attr), 121

select_changes (manip_changes), 111
select_globals (manip_globals), 113
select_nodes (manip_nodes_attr), 117
select_ties (manip_ties_attr), 121
seq_nodes (progress), 189
seq_ties (progress), 189
snet_abort (interface), 26
snet_info (interface), 26
snet_minor_info (interface), 26
snet_progress_along (progress), 189
snet_progress_nodes (progress), 189
snet_progress_seq (progress), 189
snet_progress_step (progress), 189
snet_prompt (interface), 26
snet_success (interface), 26
snet_unavailable (interface), 26
snet_warn (interface), 26
summarise_ties (manip_ties_attr), 121

table_data (data_overview), 14
tie_attribute (measure_attributes_ties), 137
tie_is_backbone (measure_attributes_ties), 137
tie_is_parallel (measure_attributes_ties), 137
tie_is_twomode (measure_attributes_ties), 137
tie_signs (measure_attributes_ties), 137
tie_weights (measure_attributes_ties), 137
to_acyclic (modif_direction), 149
to_aggregated (modif_plexity), 166
to_aggregated(), *78, 79*
to_anti (modif_plexity), 166
to_backbone (modif_backbone), 144
to_backbone(), *140*
to_blockmodel (modif_scope), 180
to_blockmodel(), *72*
to_component (modif_scope), 180
to_components (modif_split), 183
to_concepts (modif_concepts), 147
to_correlation (modif_proximity), 175
to_cosine (modif_proximity), 175
to_cosine(), *174*
to_directed (modif_direction), 149
to_disaggregated (modif_plexity), 166
to_dominating (modif_paths), 162
to_ego (modif_scope), 180
to_ego(), *72, 73*
to_egos (modif_split), 183
to_flat (modif_plexity), 166
to_geodesic (modif_paths), 162
to_giant (modif_scope), 180
to_giant(), *72*
to_hypergraph (modif_project), 169
to_hypergraph(), *8*, *106*, *110*
to_imputed (modif_miss), 157
to_labelled (modif_labels), 155
to_layer (modif_plexity), 166
to_layers (modif_split), 183
to_linegraph (modif_project), 169
to_matching (modif_paths), 162
to_mentoring (modif_paths), 162
to_mode (modif_project), 169
to_mode(), *106*

to_mode1 (modif_project), 169
to_mode1(), *179*
to_mode2 (modif_project), 169
to_motifs (modif_motifs), 160
to_multilevel (modif_levels), 156
to_named (modif_labels), 155
to_normalised (modif_weight), 186
to_normalized (modif_weight), 186
to_onemode (modif_levels), 156
to_permuted (modif_permutation), 165
to_positive (modif_weight), 186
to_proximity (modif_proximity), 175
to_proximity(), *174*
to_reciprocated (modif_direction), 149
to_redirected (modif_direction), 149
to_reporter (modif_scope), 180
to_reporters (modif_split), 183
to_reporters(), *180*
to_signed (modif_weight), 186
to_simplex (modif_plexity), 166
to_simplicial (modif_project), 169
to_simplicial(), *106*
to_slices (modif_split), 183
to_subgraph (modif_scope), 180
to_subgraph(), *72*
to_subgraphs (modif_split), 183
to_time (modif_scope), 180
to_time(), *78, 79*, *104*, *106*, *112*, *114*, *128*, *140*, *185*
to_times (modif_split), 183
to_times(), *154*, *180*, *183*
to_tree (modif_paths), 162
to_twomode (modif_levels), 156
to_twomode(), *106*
to_undirected (modif_direction), 149
to_uniplex (modif_plexity), 166
to_uniplex(), *72, 73*
to_unlabelled (modif_labels), 155
to_unlabelled(), *79*
to_unnamed (modif_labels), 155
to_unsigned (modif_weight), 186
to_unweighted (modif_weight), 186
to_wave (modif_scope), 180
to_waves (modif_split), 183
to_weighted (modif_weight), 186

validate_stocnet (make_stocnet), 101

write_dynetml (make_write), 108
write_edgelist (make_write), 108
write_gdf (make_write), 108
write_geojson (make_write), 108
write_gexf (make_write), 108
write_gml (make_write), 108
write_graphml (make_write), 108
write_hif (make_write), 108
write_matrix (make_write), 108
write_nodelist (make_write), 108
write_pajek (make_write), 108
write_ucinet (make_write), 108