

Package ‘Dynet’

October 7, 2026

Title Tidy Temporal Network Analysis

Version 0.5.1

Description Building, analysis and visualisation of temporal networks.

Supports several formats of relational data: interval data with explicit start and end times, contact data of instantaneous events, threaded interactions such as forum or chat data, and co-presence logs. Every analysis returns a tidy data frame carrying proper vertex labels. Metrics cover time-varying centrality, graph-level structure, edge formation and dissolution, burstiness, time-respecting paths, reachability and mixing. Offers a wide variety of network visualisations and animations, as well as plots of temporal network metrics across time.

License MIT + file LICENSE

Encoding UTF-8

Language en-GB

Depends R (>= 4.1)

Imports cograph, ggplot2, grDevices, graphics

Suggests av, gifski, knitr, network, networkDynamic, rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr

RoxygenNote 7.3.3

Config/testthat/edition 3

URL <https://pak.dynasite.org/Dynet/>, <https://github.com/mohsaqr/Dynet>

BugReports <https://github.com/mohsaqr/Dynet/issues>

LazyData true

Config/Needs/website av, gifski, igraph, network, networkDynamic, sna, tsna

NeedsCompilation no

Author Mohammed Saqr [aut, cre, cph]

Maintainer Mohammed Saqr <saqr@saqr.me>

Repository CRAN

Date/Publication 2026-10-07 09:10:10 UTC

Contents

add_arcs	4
add_nodes	5
add_ties	5
add_vertex_spells	6
animate	7
as.data.frame.dynet	11
as.data.frame.dynet_animation	13
as.data.frame.dynet_collapsed	14
as.data.frame.dynet_collapsed_list	15
as.data.frame.dynet_metric	16
as.data.frame.dynet_paths	17
as.data.frame.dynet_pathways	18
as.data.frame.dynet_path_network	19
as.data.frame.dynet_path_trajectories	20
as.data.frame.dynet_projection	21
as.data.frame.dynet_pshifts	22
as.data.frame.dynet_similarity	23
as.data.frame.dynet_snapshot	23
as_dynet	24
as_dynet.networkDynamic	25
burstiness	27
centrality_series	28
clear_observations	35
collapse_network	35
durations	37
dynet	39
events	43
forum_people	46
forum_posts	47
head.dynet_metric	48
induce_subgraph	48
metrics	49
mixing	53
mooc_people	56
mooc_posts	56
paths	57
pathways	61
path_centrality	62
path_network	64
path_trajectories	65
plot.dynet	67
plot.dynet_collapsed	72

plot.dynet_metric	72
plot.dynet_paths	74
plot.dynet_pathways	75
plot.dynet_path_network	76
plot.dynet_path_trajectories	76
plot.dynet_pshifts	77
plot.dynet_similarity	78
plot.dynet_snapshot	78
plot_path_trajectories	79
print.dynet	80
print.dynet_animation	81
print.dynet_collapsed	82
print.dynet_collapsed_list	82
print.dynet_metric	83
print.dynet_paths	84
print.dynet_pathways	84
print.dynet_path_trajectories	85
print.dynet_projection	86
print.dynet_pshifts	86
print.dynet_similarity	87
print.dynet_snapshot	88
projection	88
pshifts	90
reachability	92
remove_arcs	94
remove_nodes	95
remove_ties	96
remove_vertex_spells	97
rename_nodes	97
rename_sessions	98
school_contacts	99
seminar_attendance	100
set_observations	100
set_tie_sessions	101
set_vertex_spells	102
similarity	103
snapshots	104
summary.dynet	106
summary.dynet_animation	107
summary.dynet_collapsed_list	108
summary.dynet_metric	109
summary.dynet_paths	110
summary.dynet_pathways	110
summary.dynet_path_trajectories	111
summary.dynet_projection	112
summary.dynet_pshifts	112
summary.dynet_similarity	113
summary.dynet_snapshot	114

synthdata 114

tail.dynet_metric 115

thought_chains 116

update_nodes 117

update_ties 118

update_vertex_spells 119

Index 120

add_arcs	<i>Add directed temporal arcs</i>
----------	-----------------------------------

Description

The same operation as [add_ties\(\)](#), with one extra guarantee: the network must already be directed, so from and to keep the direction the caller means. Adding an arc to an undirected network raises a condition of class `dynet_needs_directed` rather than silently recording a symmetric tie.

Usage

```
add_arcs(dn, data, loops = FALSE)
```

Arguments

dn	A temporal network from dynet() .
data	A nonempty data frame with from, to, start, and end. Optional columns are weight, session, onset_censored, and terminus_censored. Endpoints must already exist in dn.
loops	Whether added self-loops are permitted. FALSE, the default, raises a condition of class <code>dynet_loop_not_allowed</code> .

Value

A new directed dynet object carrying the added spells, with the same structure as the input.

See Also

[add_ties\(\)](#), which does not require a directed network.

Examples

```
dn <- dynet(data.frame(from = "A", to = "B", start = 0, end = 1))
dn <- add_nodes(dn, "C")
add_arcs(dn, data.frame(from = "B", to = "C", start = 1, end = 2))
```

add_nodes	<i>Add nodes to a temporal network</i>
-----------	--

Description

Add nodes to a temporal network

Usage

```
add_nodes(dn, data)
```

Arguments

dn	A temporal network from <code>dynet()</code> .
data	A character vector of new node names or a data frame containing a name column and optional static attributes.

Details

Existing nodes and attributes are unchanged. Missing attribute values are filled with typed NA. If the source has a cograph grouping, each new node must supply its group through a groups column in data or through the source attribute from which that grouping was derived; otherwise a condition of class `dynet_missing_group` is raised.

Value

A new `dynet` object, of the same class and structure as the input, with the added nodes represented as implicit always-active isolates until ties or vertex activity are supplied. The input is unchanged.

Examples

```
dn <- dynet(data.frame(from = "A", to = "B", start = 0, end = 1))
add_nodes(dn, "C")
```

add_ties	<i>Add temporal ties</i>
----------	--------------------------

Description

Add temporal ties

Usage

```
add_ties(dn, data, loops = FALSE)
```

Arguments

dn	A temporal network from <code>dynet()</code> .
data	A nonempty data frame with from, to, start, and end. Optional columns are weight, session, onset_censored, and terminus_censored. Endpoints must already exist in dn.
loops	Whether added self-loops are permitted. FALSE, the default, raises a condition of class <code>dynet_loop_not_allowed</code> .

Details

Added times use the existing network clock. A sessioned network requires an existing session label on every row; mutation does not create a new session scheme. Implicit observation support expands to include the new raw ties, while explicit observation support remains fixed. Added raw rows are interval-format tie identities even when the source was originally a contact, threaded, or co-presence log.

Value

A new `dynet` object, of the same class and structure as the input. The input is unchanged; canonical temporal ties and every flattened cograph field are rebuilt together.

Examples

```
dn <- dynet(data.frame(
  from = "A", to = "B", start = 0, end = 1
))
dn <- add_nodes(dn, "C")
add_ties(dn, data.frame(from = "B", to = "C", start = 1, end = 2))
```

add_vertex_spells	<i>Add declared vertex-activity spells</i>
-------------------	--

Description

Add declared vertex-activity spells

Usage

```
add_vertex_spells(dn, data)
```

Arguments

dn	A temporal network.
data	A nonempty vertex-spell data frame with node, start, and end, plus optional session, onset_censored, and terminus_censored. Unlike <code>set_vertex_spells()</code> this argument is required; NULL is an error.

Value

A new dynet object, class `c("dynet", "netobject", "cograph_network")`, with the network's existing activity and data canonicalised together, so an added spell that overlaps or abuts an existing one for the same vertex is merged into it and canonical spell identifiers may change. Raises `dynet_unknown_node` for a vertex the network does not have, and `dynet_bad_input` when data is not a nonempty data frame, and `dynet_incompatible_vertex_spells` when session is supplied to a network with no session scheme – the same refusal `update_vertex_spells()` makes, rather than dropping the label.

Examples

```
dn <- dynet(school_contacts)
present <- data.frame(node = c("Ana", "Ben"), start = 0, end = 10)
enrolled <- set_vertex_spells(dn, present)
extended <- add_vertex_spells(enrolled,
                             data.frame(node = "Cara", start = 5, end = 20))
as.data.frame(extended, what = "vertex_spells")
```

animate

*Animate a temporal network over its measurement grid***Description**

Draws the network bin by bin over the measurement grid and writes the frames to an animated GIF or a video. The grid is the same four arguments every measuring verb takes, so an animation shows exactly what `snapshots()` tabulates and what `plot(dn, type = "snapshots")` draws as a filmstrip, with the bins joined by motion.

Usage

```
animate(
  dn,
  start = NULL,
  end = NULL,
  step = NULL,
  window = NULL,
  sessions = c("bounded", "collapse"),
  layout = "spring",
  measure = NULL,
  tween = 6L,
  fps = 12,
  file = tempfile(fileext = ".gif"),
  loop = TRUE,
  width = 800L,
  height = 800L,
  res = 120,
  palette = "okabe",
```

```

    tie_states = TRUE,
    timeline = TRUE,
    absent = c("fade", "away", "hide"),
    isolates = c("fade", "show", "hide"),
    ease = c("dwell", "continuous"),
    max_displacement = 0.08,
    anchor_strength = 1,
    layout_args = list(),
    seed = 42L,
    ...
)

```

Arguments

<code>dn</code>	A temporal network from <code>dynet()</code> .
<code>start, end, step, window</code>	The measurement grid, as in <code>snapshots()</code> . NULL, the default, takes each from the network's own observation window and bin width.
<code>sessions</code>	How to treat sessions, as in <code>centrality_series()</code> : "bounded" (the default) or "collapse". An animation draws calendar bins, so "separate" is not offered.
<code>layout</code>	"spring" (the default), "relaxed", "circle", "oval" or "groups", or a data frame of coordinates. See details.
<code>measure</code>	What node size follows. NULL, the default, keeps every vertex the same size. The name of a snapshot node measure from <code>centrality_series()</code> , such as "degree" or "betweenness", computes it on the animation's own grid, so a vertex grows and shrinks bin by bin. A node-level result of <code>centrality_series()</code> is matched by vertex and time; one computed with <code>window = "all"</code> holds a single value per vertex, so every vertex keeps one size for the whole film, for instance its degree over the whole period. The name of a numeric vertex attribute supplied through <code>dynet(nodes =)</code> does the same with the attribute's values, which is how a two-tier size (a circle of interest against everyone else) is drawn.
<code>tween</code>	Frames drawn per bin. One positive whole number, 6 by default.
<code>fps</code>	Frames per second. One positive number, 12 by default.
<code>file</code>	Path to write to, ending in .gif, .mp4 or .webm. Defaults to a GIF in the session's temporary directory; nothing is written to the working directory unless the path says so.
<code>loop</code>	For a GIF: TRUE, the default, repeats for ever; FALSE plays once; a positive whole number repeats that many times. Ignored for a video.
<code>width, height</code>	Frame size in pixels, 800 by default. A video needs both to be even.
<code>res</code>	Resolution passed to <code>grDevices::png()</code> , 120 by default.
<code>palette</code>	Palette specification, as in <code>plot.dynet()</code> . "okabe" by default.
<code>tie_states</code>	Whether to draw forming, persisting and dissolving ties differently. TRUE by default.
<code>timeline</code>	Whether to draw the timeline strip. TRUE by default.

absent	How a vertex is drawn in a bin where it is not present. "fade", the default, keeps it in place at a quarter of its opacity; "away" parks it, invisible, at the edge of the layout on its own side and glides it in over the transition in which it arrives and out over the one in which it leaves, opaque for most of the glide and, with <code>tie_states = TRUE</code> , wearing a thick ring in the forming colour on the way in and the dissolving colour on the way out; "hide" keeps it in place, invisible.
isolates	How a vertex that is present but has no tie in a bin is drawn. "fade", the default, at 35% of its opacity; "show" at full opacity; "hide" invisible.
ease	"dwell", the default, holds each bin still before it changes; "continuous" keeps everything moving, with positions on a spline through the bins and linear fades. See details.
max_displacement	How far a vertex may move between bins under <code>layout = "relaxed"</code> , in layout units. 0.08 by default; ignored otherwise.
anchor_strength	How strongly a vertex is pulled back towards its previous position under <code>layout = "relaxed"</code> . 1 by default; ignored otherwise.
layout_args	A named list of further arguments for <code>cograph::layout_spring()</code> , used by <code>layout = "spring"</code> and <code>"relaxed"</code> : <code>repulsion</code> , <code>attraction</code> , <code>area</code> , <code>gravity</code> , <code>iterations</code> or <code>cooling</code> . Empty by default. A larger repulsion opens a dense core.
seed	Seed for the spring layouts, so "spring" and "relaxed" are reproducible; 42 by default. Under a seed the caller's random state is restored on exit. NULL draws from the current random state instead and leaves it advanced, which is what makes successive unseeded calls differ.
...	Passed to <code>cograph::splot()</code> for every frame, so the whole drawing surface of <code>plot.dyndet()</code> 's network view is available. labels may also be the name of a vertex attribute supplied through <code>dyndet(nodes =)</code> , such as a short form of each name, which is then drawn in place of the vertex names; label itself is reserved by the node table, so give the attribute another name. Seven arguments are read as the animation's baselines rather than passed on: <code>edge_width_range</code> (the widths the weight scale maps onto, c(0.5, 3.5) by default), <code>edge_alpha</code> (0.6), <code>edge_color</code> (the colour of a tie when <code>tie_states = FALSE</code>), <code>node_size</code> (the size a vertex has without a measure) and <code>node_size_range</code> (the smallest and largest radius a measure maps onto; by default 0.55 and 1.8 times <code>node_size</code>), <code>node_alpha</code> (1) and <code>node_border_color</code> ("white"), the last two being what absent and isolates fade from.

Details

Frames. Each bin is drawn tween times. Between one bin and the next the vertices glide to their new positions, a tie that is about to appear fades in and one that is about to vanish fades out, and a vertex whose measure changes grows or shrinks. Under `ease = "dwell"`, the default, the motion follows the smoothstep curve, so each bin holds still before it starts to change and the bins can be read one by one. Under `ease = "continuous"` nothing holds still: positions follow a Catmull-Rom spline through the bins, so a vertex moving across several bins traces one smooth path, and fades are linear. `tween = 1` gives one frame per bin with hard cuts. A film that feels episodic usually has

a grid whose bins do not overlap; a sliding window, step smaller than window, smooths the data itself, since a tie then persists across several bins.

Layouts. Under every layout but "relaxed" a vertex keeps one position for the whole animation, so the only thing that moves is the ties; those are the layouts to read structure from. "spring", the default, lays out the union of every frame once with `cograph::layout_spring()`, so pairs that met often sit close. "circle" and "oval" are rings, in vertex order. "groups" puts each partition on its own ring and needs a network built with `groups = .` "relaxed" lays each frame out again, seeded from the previous one and held near it by `max_displacement` and `anchor_strength`, then smooths every vertex's path with a centred triangular kernel over one bin each side; clusters can form and dissolve without vertices jumping, and no vertex moves further than `max_displacement` between consecutive bins. Each relaxed frame is framed by its active vertices: a vertex that is absent or has no tie in the bin is held at the border of that frame rather than drifting outward under repulsion, which would shrink the picture. `layout_args` tunes the spring layout for both. A data frame with columns `name` (or `node`), `x` and `y` fixes the positions yourself.

What is drawn. Tie width follows weight on one scale fixed across the whole animation, so a tie of the same weight has the same width in a quiet frame and a busy one. With `tie_states = TRUE` a tie forming during a transition is dotted and green, one persisting is solid and grey, and one dissolving is dashed and vermilion, so the distinction survives without colour. With `measure` given, node size follows that measure, again on one scale across every frame, with the area of the circle proportional to the measure's position in its range; see the argument for the three forms it takes.

Absence and idleness. A vertex that is not present in a bin, under declared vertex activity or observation bounds, is drawn as absent says: faded in place, parked out of sight at the edge of the layout and gliding in when it arrives and out when it leaves, or hidden in place. A network built without vertex activity has every vertex present in every bin; `set_vertex_spells(dn, "ties")` declares each vertex present from its first tie to its last. A vertex that is present but has no tie in a bin is drawn as isolates says. With `timeline = TRUE` a strip under the network shows the grid with a marker at the current time and the key to the drawing.

Files. The extension of file chooses the encoder: `.gif` is written by the `gifski` package, `.mp4` and `.webm` by the `av` package. A video needs even pixel dimensions. Writing the file is the point of the verb, but the tidy bin table is still what comes back, so the animation can be described without opening it.

Value

An object of class "dynet_animation": a tidy data frame with one row per bin and columns `bin`, `frame` (the first rendered frame of the bin), `time` (the bin's label on the network's time scale), `window_start` and `window_end` (its bounds), `nodes` (vertices present), `idle` (of those, vertices with no tie), `ties` (ties drawn), `forming` (ties not active in the previous bin, NA for the first), `dissolving` (ties not active in the next bin, NA for the last), and `file` (the same on every row). These are counts of what the picture shows, not the risk-set accounting of `events()`. The rendered-frame schedule is available through `as.data.frame(x, what = "frames")`. Returned invisibly, since writing the file is the verb's purpose.

Conditions

Raises `dynet_unknown_format` for a file extension other than `.gif`, `.mp4` or `.webm`; `dynet_needs_gifski` or `dynet_needs_av` when the encoder that extension needs is not installed; `dynet_needs_cograph` when `cograph` is not; `dynet_empty_result` when the grid holds no bin that can be drawn; `dynet_unknown_attribute`

for layout = "groups" on a network without a partition or a labels naming no vertex attribute; dynet_missing_column and dynet_unknown_node for a coordinate table that is incomplete; dynet_unknown_measure for a measure that is neither a measure `centrality_series()` offers nor a numeric vertex attribute, and whatever `centrality_series()` raises for one it refuses; dynet_bad_input for a `centrality_series()` result that is not node-level or lands on none of the bins, and a warning of class dynet_partial_measure when it lands on only some; and dynet_bad_input for a non-positive fps, tween, width, height or res, an odd video size, a loop that is neither logical nor a positive whole number, or a negative max_displacement or anchor_strength.

See Also

`snapshots()` for the same grid as a table, `plot.dynet()` with type = "snapshots" for it as a static filmstrip, and `centrality_series()` for the measures node size can follow.

Examples

```
if (requireNamespace("gifski", quietly = TRUE) &&
    requireNamespace("cograph", quietly = TRUE)) {
  dn <- dynet(school_contacts)
  frames <- animate(dn = dn, end = 8, step = 4, window = 4, tween = 2)
  frames
  summary(frames)
}

if (requireNamespace("av", quietly = TRUE) &&
    requireNamespace("cograph", quietly = TRUE)) {
  dn <- dynet(school_contacts)
  file <- tempfile(fileext = ".mp4")
  video <- animate(dn = dn, end = 8, step = 4, tween = 2, measure = "degree", file = file)
  summary(video)
}
```

as.data.frame.dynet *Tidy tables from a temporal network*

Description

Tidy tables from a temporal network

Usage

```
## S3 method for class 'dynet'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  what = c("edges", "nodes", "bins", "network", "observations", "observed_edges",
    "vertex_spells"),
```

```

measure = NULL,
sessions = c("bounded", "collapse", "separate"),
start = NULL,
end = NULL,
...
)

```

Arguments

x	A temporal network from <code>dynet()</code> .
row.names	Ignored; present for compatibility with the generic.
optional	Ignored; present for compatibility with the generic.
what	Which table to return: "edges", the default, for raw edge spells, "observed_edges" for derived observation fragments, "observations" for canonical observed support, "vertex_spells" for canonical declared vertex activity, "nodes" for the vertex table, "bins" for the measurement grid, or "network" for the aggregate edge list cograph renders.
measure	Optional centrality measures to annotate the vertex table with, valid only for what = "nodes". Each becomes one column holding the value over the whole observed period, so the vertex table can be filtered or ranked without a second call. Any measure <code>centrality_series()</code> accepts at snapshot scope is allowed, plus "indegree" and "outdegree"; anything else raises a <code>dynet_unknown_measure</code> error, and a measure that is not a character vector raises <code>dynet_bad_input</code> . Naming it for any other what raises a <code>dynet_bad_input</code> error too.
sessions	How sessions are treated while measure is computed: "bounded" (the default), "collapse" or "separate", as in <code>centrality_series()</code> . Ignored when measure is not given.
start, end	Measurement bounds passed to <code>centrality_series()</code> when measure is given, and ignored otherwise. Default to the observed range.
...	Ignored.

Value

A plain `data.frame`, one row per whatever what names.

"edges": one row per unchanged raw spell, with from, to, start, end, duration and weight. A session column is present when the network was built with sessions, `onset_censored` and `terminus_censored` when interval censoring was declared explicitly, and any column the construction carried through – thread for a threaded log, group for a co-presence log.

"observations": one row per canonical observation component, with observation, start, end, duration and instant.

"observed_edges": one row per derived observation fragment, with raw_spell, observation and fragment locating it, from, to, start, end (clipped to the observation), raw_start, raw_end (as supplied), weight, instant, the strict `left_observation_censored` and `right_observation_censored` flags, and duration. session and the explicit `onset_censored/terminus_censored` flags are copied unchanged from the raw spell when the network carries them.

"vertex_spells": one row per maximal declared activity component, with vertex_spell, node, start, end, duration, instant, session, onset_censored and terminus_censored. Half-open positive spells and exact points are both representable. Undeclared vertices are implicitly always active and receive no synthetic rows, so this table is empty for a network with no declared vertex activity.

"nodes": one row per vertex, with name, any static attributes supplied at construction, and one column per measure named in measure.

"bins": one row per measurement window, with bin, lo, hi, time (the bin's representative time) and closed (whether the upper bound is included). Bins are component-qualified under discontinuous observation.

"network": one row per aggregate vertex pair, with from, to and the summed weight cograph renders.

Examples

```
dn <- dynet(school_contacts)
spells <- as.data.frame(dn)
head(spells)
as.data.frame(dn, what = "nodes")
as.data.frame(dn, what = "vertex_spells")

# Annotate the vertex table so it can be filtered without a second call.
busy <- as.data.frame(dn, what = "nodes",
                      measure = c("degree", "indegree", "outdegree"))
subset(busy, degree > 15)
```

```
as.data.frame.dynet_animation
```

Coerce an animation to a data frame

Description

Coerce an animation to a data frame

Usage

```
## S3 method for class 'dynet_animation'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  what = c("bins", "frames"),
  ...
)
```

Arguments

`x` A `dynet_animation` from `animate()`.

`row.names, optional` As in `base::as.data.frame()`.

`what` "bins", the default, for one row per bin with the columns `animate()` documents; "frames" for one row per rendered frame, with frame, bin (the bin the frame belongs to), phase (how far along the transition to the next bin, in $[0, 1)$) and time (where the timeline marker stands).

`...` Ignored.

Value

A plain data.frame.

Examples

```
if (requireNamespace("gifski", quietly = TRUE) &&
    requireNamespace("cograph", quietly = TRUE)) {
  dn <- dynet(school_contacts)
  frames <- animate(dn, step = 6, window = 6, tween = 2)
  as.data.frame(frames)
  as.data.frame(frames, what = "frames")
}
```

as.data.frame.dynet_collapsed

Tidy tables from a collapsed temporal network

Description

Tidy tables from a collapsed temporal network

Usage

```
## S3 method for class 'dynet_collapsed'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  what = c("edges", "nodes"),
  ...
)
```

Arguments

`x` A network returned by `collapse_network()`.
`row.names, optional` Ignored; present for compatibility.
`what` "edges", the default, or "nodes".
`...` Ignored.

Value

A plain data.frame. For "edges", one row per collapsed vertex pair carrying every weighting side by side: from, to, binary, union_duration, total_duration, duration_fraction, spell_count, weight_sum, weighted_duration, latest_weight, first, last, and the activity.duration and activity.count aliases. For "nodes", one row per vertex with name, any static vertex attributes the network carries, activity_duration and its activity.duration alias. See `collapse_network()` for what each weighting means.

Examples

```
dn <- dynet(school_contacts)
collapsed <- collapse_network(dn)
as.data.frame(collapsed)
as.data.frame(collapsed, what = "nodes")
```

as.data.frame.dynet_collapsed_list

Tidy data frame of session-specific collapsed networks

Description

Stacks the per-session edge tables into one tidy frame with a session key, so a session-separated collapse reads like every other verb's result. Without this the base method flattened the list sideways into a single row of s1.from, s2.from, ... columns.

Usage

```
## S3 method for class 'dynet_collapsed_list'
as.data.frame(x, row.names = NULL, optional = FALSE, session = NULL, ...)
```

Arguments

`x` A dynet_collapsed_list from collapse_network(sessions = "separate").
`row.names` Ignored; present for compatibility with the generic.
`optional` Ignored; present for compatibility with the generic.
`session` Optional session name. Supply one to get that session's table alone, without the session key; the default, NULL, stacks them all. A name that is not one of the collapsed sessions raises a dynet_unknown_session error.
`...` Ignored.

Value

A plain data.frame, one row per collapsed pair per session, with session first and then the columns as.data.frame.dynet_collapsed() returns.

Examples

```
dn <- dynet(data.frame(
  from = c("A", "A"), to = c("B", "B"), start = c(0, 0), end = c(2, 3),
  session = c("s1", "s2")
), session = "session")
by_session <- collapse_network(dn, sessions = "separate")
as.data.frame(by_session)
as.data.frame(by_session, session = "s1")
```

```
as.data.frame.dynet_metric
```

Tidy data frame of a temporal measure

Description

Tidy data frame of a temporal measure

Usage

```
## S3 method for class 'dynet_metric'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  layout = c("long", "wide"),
  what = c("values", "diagnostics"),
  top = NULL,
  ...
)
```

Arguments

x	A dynet_metric produced by any measurement verb.
row.names	Ignored; present for compatibility with the generic.
optional	Ignored; present for compatibility with the generic.
layout	"long" gives one row per observation, which is the default and the shape every other verb expects. "wide" spreads time across columns, giving one row per vertex (or per measure for graph-level quantities), which is convenient for exporting a table. A measure with no time axis, such as reachability, is spread by measure instead: one row per vertex with one column per measure.

what	"values", the default, gives the measured values. "diagnostics" gives the record a prestige computation keeps when it cannot produce a value, which is what the accompanying warning refers to: one row per reporting block that was undefined, infeasible or nonconverged, with session, time, stage, status and reason, the solver's iterations and residual, the balance_* family for the row-column scaling step, and spectral_radius, eigenspace_dimension and eigen_residual for the eigen step. A result with nothing to report gives a zero-row frame of those same columns rather than NULL.
top	Keep only the top vertices with the largest mean value, and order the result from largest to smallest. A single positive number; NULL, the default, keeps every row in the measure's own order. It selects vertices, so it applies to what = "values" on a measure that has a node column; anything else raises a dynet_bad_input error.
...	Ignored.

Value

A plain data.frame. Long layout carries measure and value with one row per observation, alongside whichever columns say what was measured: session when the network has sessions, time for anything measured on a grid of bins, node for a vertex-level quantity, from and to for a pair-level one, raw_spell for per-spell edge durations and vertex_spell with implicit for per-spell vertex durations from `durations()`, and from_group and to_group for `mixing()`. A graph-level series carries time, measure and value alone.

Wide layout puts the identifying columns first and spreads what varies across the rest. A measure taken on a grid of bins spreads time: one column per bin, named t followed by the bin's time, leaving one row per vertex and measure. A measure with no time axis, such as reachability, spreads the measures instead: one column per measure, leaving one row per vertex. A measure with no time axis and only one measure is already wide and comes back unchanged.

Examples

```
dn <- dynet(school_contacts)
degree <- centrality_series(dn, measure = "degree")
as.data.frame(degree)
as.data.frame(degree, layout = "wide")
as.data.frame(degree, top = 5)
as.data.frame(degree, what = "diagnostics")
```

as.data.frame.dynet_paths

Tidy data frame of time-respecting paths

Description

Tidy data frame of time-respecting paths

Usage

```
## S3 method for class 'dynet_paths'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  what = c("paths", "steps"),
  ...
)
```

Arguments

x	A dynet_paths.
row.names	Ignored; present for compatibility with the generic.
optional	Ignored; present for compatibility with the generic.
what	"paths", the default, for the endpoint summary, or "steps" for the tidy reconstructed optimal routes. The latter includes endpoint-local path_id values for tied contact sequences.
...	Ignored.

Value

A plain data.frame. For "paths", one row per endpoint vertex, the source included, with the columns `paths()` documents: node, reachable, arrival_time, attained, latency, n_hops and n_paths, plus path_session and n_best_sessions under sessions = "bounded", and session and origin under sessions = "separate". For "steps", one row per step of every reconstructed optimal route, with endpoint (the vertex the route ends at), path_id (which of the tied optimal routes to that endpoint), path_session, step (position along the route, starting at the source), node (the vertex occupied at that step), time (when it was reached) and attained.

Examples

```
dn <- dynet(school_contacts)
reach <- paths(dn, from = "Ana")
as.data.frame(reach)
as.data.frame(reach, what = "steps")
```

```
as.data.frame.dynet_pathways
```

Tidy data frame of ranked pathways

Description

Tidy data frame of ranked pathways

Usage

```
## S3 method for class 'dynet_pathways'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  what = c("routes", "steps"),
  ...
)
```

Arguments

x	A dynet_pathways result.
row.names	Ignored; present for compatibility with the generic.
optional	Ignored; present for compatibility with the generic.
what	"routes", the default, gives one row per distinct route. "steps" gives one row per vertex visited – route, step, vertex and the time the route reaches it, preceded by from when pooled – which is the per-hop timing the plot draws and the shape to use for any waiting-time analysis of your own.
...	Ignored.

Value

A plain data.frame with the columns described in [pathways\(\)](#), most frequent first, or the per-step table when what = "steps".

Examples

```
dn <- dynet(school_contacts)
routes <- pathways(dn, from = "Ana")
as.data.frame(routes)
as.data.frame(routes, what = "steps")
```

as.data.frame.dynet_path_network

Tidy tables from a temporal path-union network

Description

Tidy tables from a temporal path-union network

Usage

```
## S3 method for class 'dynet_path_network'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  what = c("edges", "nodes"),
  ...
)
```

Arguments

x	A network returned by path_network() .
row.names, optional	Ignored.
what	"edges", the default, or "nodes".
...	Ignored.

Value

A plain data.frame. For "edges", one row per hop used by an optimal route, with from, to, weight, first_time, last_time and n_endpoints. For "nodes", one row per reached vertex, with name, arrival_time, latency, n_hops, n_paths and groups. See [path_network\(\)](#) for what each column means.

Examples

```
dn <- dynet(school_contacts)
routes <- paths(dn, from = "Ana")
union_network <- path_network(routes)
as.data.frame(union_network, what = "edges")
as.data.frame(union_network, what = "nodes")
```

```
as.data.frame.dynet_path_trajectories
```

Tidy table of a temporal trajectory tree

Description

Tidy table of a temporal trajectory tree

Usage

```
## S3 method for class 'dynet_path_trajectories'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

x	A result from <code>path_trajectories()</code> .
row.names, optional	Ignored.
...	Ignored.

Value

A plain data frame with one row per tree node, with the columns described in `path_trajectories()` and none of its attributes.

Examples

```
dn <- dynet(school_contacts)
routes <- paths(dn, from = "Ana")
trajectories <- path_trajectories(routes)
as.data.frame(trajectories)
```

as.data.frame.dynet_projection

Tidy tables from a time-projected network

Description

Tidy tables from a time-projected network

Usage

```
## S3 method for class 'dynet_projection'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  what = c("vertices", "edges"),
  ...
)
```

Arguments

x	A projection returned by <code>projection()</code> .
row.names	Ignored; present for compatibility with the generic.
optional	Ignored; present for compatibility with the generic.
what	"vertices", the default, returns vertex-time states, and "edges" returns directed within-slice and identity arcs.
...	Ignored.

Value

A plain data frame. Vertex rows contain state, optional session and observation, slice, time, start, end, closed, node, active, and copied node attributes. Edge rows contain from_state, to_state, from_node, to_node, optional session, from_slice, to_slice, from_time, to_time, edge_type, weight, n_spells, and lag.

Examples

```
dn <- dynet(school_contacts)
projected <- projection(dn, step = 4, window = 4)
as.data.frame(projected)
as.data.frame(projected, what = "edges")
```

```
as.data.frame.dynet_pshifts
```

Tidy data frame of participation shift counts

Description

Tidy data frame of participation shift counts

Usage

```
## S3 method for class 'dynet_pshifts'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

x	A dynet_pshifts result.
row.names	Ignored; present for compatibility with the generic.
optional	Ignored; present for compatibility with the generic.
...	Ignored.

Value

A plain data.frame carrying the same rows and columns as x. For a result built with output = "final" that is one row per shift type – shift, family and count – preceded by session when the result is session-local; the thirteen Gibson shift types are always present, including those with a count of zero. For output = "cumulative" it is thirteen rows per classified turn, adding sequence, event, time, speaker, target and group ahead of shift, family and count.

Examples

```
dn <- dynet(school_contacts)
shifts <- pshifts(dn)
as.data.frame(shifts)
```

```
as.data.frame.dynet_similarity
```

Tidy table of time-bin similarity

Description

Tidy table of time-bin similarity

Usage

```
## S3 method for class 'dynet_similarity'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

x	A result from similarity() .
row.names, optional	Ignored; present for compatibility with the generic.
...	Ignored.

Value

A plain data.frame, one row per ordered pair of time bins, with columns time, other, measure and value.

Examples

```
dn <- dynet(school_contacts)
resemblance <- similarity(dn, step = 4, window = 4)
as.data.frame(resemblance)
```

```
as.data.frame.dynet_snapshot
```

Tidy table of snapshot edges

Description

Tidy table of snapshot edges

Usage

```
## S3 method for class 'dynet_snapshot'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

x	A dynet_snapshot from <code>snapshots()</code> .
row.names	Ignored; present for compatibility with the generic.
optional	Ignored; present for compatibility with the generic.
...	Ignored.

Value

A plain data.frame with the same rows and columns.

Examples

```
dn <- dynet(school_contacts)
bins <- snapshots(dn)
bins_table <- as.data.frame(bins)
head(bins_table)
```

as_dynet

Convert an object to a Dynet temporal network

Description

Imports a temporal network held in another R representation, so that every Dynet verb applies to it. A method is supplied for networkDynamic objects; the method for dynet is the identity.

Usage

```
as_dynet(x, ...)

## S3 method for class 'dynet'
as_dynet(x, ...)
```

Arguments

x	An object representing a temporal network.
...	Passed to a class-specific method.

Value

A `dynet()` temporal network: an object of class `c("dynet", "netobject", "cograph_network")` carrying the tie ledger, the node table and the construction metadata. The dynet method is the identity, returning x unchanged, so `as_dynet()` is safe to call on an object that is already a temporal network.

See Also

`as_dynet.networkDynamic()`, which imports a networkDynamic object.

Examples

```

dn <- dynet(data.frame(from = "A", to = "B", start = 0, end = 1))
as_dynet(dn)
dn <- dynet(school_contacts)
same <- as_dynet(dn)
same

```

as_dynet.networkDynamic

Import a networkDynamic object

Description

Converts the edge-activity spell ledger, observation support, explicit vertex activity, weights, static vertex attributes, and static edge attributes of a networkDynamic object. Integer vertex identifiers are replaced by a complete unique vertex attribute. By default Dynet tries Name, Label, and then vertex.names, in that order.

Usage

```

## S3 method for class 'networkDynamic'
as_dynet(
  x,
  name_attribute = NULL,
  group_attribute = NULL,
  weight_attribute = "weight",
  session_attribute = NULL,
  interval = NULL,
  active_default = TRUE,
  import_edge_attributes = TRUE,
  ...
)

```

Arguments

x	A networkDynamic object.
name_attribute	Optional vertex attribute holding the public node names. NULL chooses the first complete unique attribute among Name, Label, and vertex.names.
group_attribute	Optional static vertex attribute used as the cograph grouping variable.
weight_attribute	Static edge attribute used as spell weight, "weight" by default. Supply NULL to use unit weights. A name that is not a static edge attribute of x also yields unit weights.

session_attribute	Optional static edge attribute used as the spell session label. NULL, the default, leaves the network unsessioned; a name that is not a static edge attribute raises a condition of class dynet_unknown_attribute.
interval	Positive measurement interval. NULL, the default, uses the legacy observation time increment when available, otherwise one.
active_default	Whether legacy edges with no explicit activity spell are active over the observation period, matching the same argument in networkDynamic. TRUE by default.
import_edge_attributes	Whether to retain compatible static legacy edge attributes on the raw Dynet spell ledger. TRUE by default; an attribute whose name would collide with a canonical spell column is prefixed with edge_.
...	Ignored.

Details

Dynamic edge attributes other than activity itself are not part of the networkDynamic spell-list interface and are therefore not imported. Ordinary per-edge attributes are repeated onto every imported spell of the corresponding aggregate edge, matched on the canonical endpoint order `dynet()` stores, so an undirected pair carries its own attributes.

Value

A `dynet()` temporal network: an object of class `c("dynet", "netobject", "cograph_network")` whose spell table has one row per imported edge-activity spell, with the legacy vertex and edge attributes, observation support, vertex activity and censor flags carried across. Its metadata additionally carries `legacy_source = "networkDynamic"`, the chosen `legacy_name_attribute`, the retained `legacy_edge_attributes` and any `legacy_edge_attribute_renames`.

See Also

`as_dynet()`, the generic.

Examples

```
if (requireNamespace("networkDynamic", quietly = TRUE) &&
    requireNamespace("network", quietly = TRUE)) {
  spells <- data.frame(onset = c(0, 1), terminus = c(2, 3),
                      tail = c(1, 2), head = c(2, 3))
  legacy <- networkDynamic::networkDynamic(edge.spells = spells)
  network::set.vertex.attribute(legacy, "Name", c("A", "B", "C"))
  dn <- as_dynet(legacy)
  as.data.frame(dn)
}
```

burstiness

*Burstiness and memory of each vertex's activity***Description**

Whether a vertex acts in bursts or at a steady pace. Burstiness compares the spread of the gaps between a vertex's events with their average: it approaches 1 for increasingly heterogeneous sequences, has theoretical reference value 0 for a Poisson process, and is -1 for a metronome. The memory coefficient asks a different question – whether a short gap tends to be followed by another short gap.

Two vertices can post the same number of times and differ entirely on both.

Usage

```
burstiness(
  dn,
  measure = c("burstiness", "memory", "events"),
  sessions = c("bounded", "collapse", "separate"),
  plot = FALSE
)
```

Arguments

dn	A temporal network from <code>dynet()</code> .
measure	One or more of "burstiness", "memory", "events" and "mean_gap". Defaults to the first three. Anything else raises a <code>dynet_unknown_measure</code> error.
sessions	How to treat sessions: "bounded" (the default), "collapse" or "separate", as in <code>centrality_series()</code> .
plot	Whether to draw the result as well as return it. Drawing is a side effect in the manner of <code>graphics::hist()</code> : the verb still returns its tidy table, invisibly when it has drawn, so <code>plot = TRUE</code> saves the wrapping <code>plot()</code> call without changing what comes back. Use <code>plot()</code> on the result when the figure needs arguments of its own.

Details

One raw spell row contributes its start time once to each distinct incident vertex. A self-loop is one event, equal-time rows remain distinct events, direction does not alter incidence, and interval ends and weights are ignored. Sorted equal times therefore create legitimate zero gaps. Explicitly onset-censored limits are not observed onset events and are excluded; terminus censoring does not affect this onset sequence.

If the usable interevent gaps are $\tau_1, \dots, \tau_k$, burstiness is

$$B = (\sigma - \mu) / (\sigma + \mu),$$

where μ is their mean and $\sigma = \sqrt{k^{-1} \sum_i (\tau_i - \mu)^2}$ is the population standard deviation of the equal-mass empirical gap distribution. `mean_gap` needs at least one gap. Burstiness needs at least two and is NA if every usable gap is zero. Its finite-sample range is $[-1, 1)$.

Memory is the ordinary Pearson correlation between consecutive gaps. It needs at least two adjacent-gap pairs and nonzero variation on both sides; otherwise it is NA. In `sessions = "bounded"`, primitive gaps and adjacent pairs are formed within each session and then pooled, so no cross-session gap is introduced. Collapse includes calendar gaps after erasing labels; separate returns session-local blocks over the fixed vertex universe.

Value

A `dynet_metric` at node level with no time column: one row per vertex and measure. Attributes record the event identity, dispersion, memory, loop, weight, and session-gap conventions as `event_identity = "incident_spell_start"`, `dispersion = "population"`, `memory = "lag1_pearson"`, `loop_contribution = "one_event"`, `weights = "ignored"`, and mode-specific `session_gaps`.

References

Goh, K.-I., & Barabasi, A.-L. (2008). Burstiness and memory in complex systems. *Europhysics Letters*, 81(4), 48002, equations 1 and 4. doi:[10.1209/02955075/81/48002](https://doi.org/10.1209/02955075/81/48002)

Examples

```
dn <- dynet(school_contacts)
burstiness(dn)
```

centrality_series	<i>Time-varying vertex centrality</i>
-------------------	---------------------------------------

Description

Centrality for every vertex at every time point. Ask for several measures in one call and they arrive stacked in a single tidy frame, one row per vertex, time point and measure.

Each value measures the network as it stands in one time bin, so the result is a trajectory of ordinary centrality. Order within a bin is not used: every tie active in the bin counts as present. Centrality computed from time-respecting paths across the whole period is [path_centrality\(\)](#).

Usage

```
centrality_series(
  dn,
  measure = "degree",
  sessions = c("bounded", "collapse", "separate"),
  sample = NULL,
  damping = 0.85,
  mode = c("all", "out", "in"),
```

```

start = NULL,
end = NULL,
step = NULL,
window = NULL,
exponent = 1,
prestige = "indegree",
rescale = FALSE,
lambda = 1,
plot = FALSE
)

```

Arguments

dn	A temporal network from <code>dynet()</code> .
measure	One or more of "degree", "strength", "prestige", "closeness", "betweenness", "eigenvector", "pagerank", "hub", "authority", "coreness", "constraint", "power", "harary", "information", "load", "flow_betweenness", or "diffusion". The deprecated names "indegree" and "outdegree", which warn with class <code>dynet_deprecated</code> and are replaced by <code>measure = "degree"</code> with <code>mode = "in"</code> or <code>mode = "out"</code> . Defaults to "degree". Any other name raises an error of class <code>dynet_unknown_measure</code> ; the directed-only measures "prestige", "hub", "authority" and the two deprecated names "indegree" and "outdegree" raise <code>dynet_needs_directed</code> on an undirected network.
sessions	How to treat sessions: "bounded" (the default) keeps paths inside a session, "collapse" ignores sessions, "separate" reports each session on its own rows. "separate" on a network built without a session column raises an error of class <code>dynet_no_sessions</code> .
sample	Deprecated. "instant" is equivalent to <code>window = 0</code> ; "window" uses the current positive/default window.
damping	Damping factor for PageRank; a single number strictly between zero and one, 0.85 by default.
mode	Which edges count on a directed network: "all" both directions, "out" outgoing only, "in" incoming only. Defaults to "all"; any other string raises an error of class <code>dynet_bad_input</code> . Name several at once – <code>mode = c("all", "in", "out")</code> – to get degree, in-degree and out-degree from a single call; the extra directions are then labelled <code>degree_in</code> and <code>degree_out</code> in the measure column, while a call naming one direction keeps the plain measure name. Applies to "degree", "strength", "closeness", "coreness", "harary", "eigenvector" and "diffusion"; the remaining measures have a single directional definition and ignore it. Ignored entirely on an undirected network. In-degree is therefore <code>mode = "in"</code> . The old "indegree" and "outdegree" measure names remain as deprecated aliases.
start, end	First and last time at which to measure. Default to the observed range. A network built from dates may be addressed with dates.
step	How often to measure. Defaults to the interval the network was built with.
window	How much time each measurement covers. Defaults to <code>step</code> , which tiles the period into disjoint bins. A larger value slides an overlapping window; 0 samples

	the network at each point in time. "all" measures the whole observed period as one window, closed on the right so an event at the final instant is inside it; it cannot be combined with step, and under sessions = "separate" or discontinuous observation it gives one window per session or observed component.
exponent	Attenuation factor for Bonacich "power", a single finite number, 1 by default. Positive rewards being connected to well-connected others; negative rewards the opposite, which is the bargaining reading.
prestige	Prestige definition, "indegree" by default. "indegree" counts distinct active incoming dyads. "indegree.rownorm" first gives every active sender one unit split equally across its distinct outgoing dyads, then sums the received mass. "indegree.rowcolnorm" balances a total-support binary adjacency to doubly stochastic form; feasible scores are necessarily uniform. "domain" counts the distinct other vertices with a directed path into each vertex in the active snapshot. "domain.proximity" discounts that incoming domain fraction by its mean directed hop distance. "eigenvector" uses the unique nonnegative Perron ray of the transposed binary adjacency. "eigenvector.rownorm" first divides every nonzero binary sender row by its outgoing-dyad count and then solves the same certified incoming Perron equation. "eigenvector.colnorm" instead divides every nonzero binary receiver column by its incoming-dyad count before solving. "eigenvector.rowcolnorm" first certifies total support and balances binary adjacency to doubly stochastic form. Prestige is directed and snapshot-only.
rescale	Whether to divide prestige by its total independently inside every reported time/session block; FALSE by default. Zero-total count/proximity definitions return NaN; structurally undefined spectral definitions return NA. This argument requires measure = "prestige", and raises dynet_bad_input otherwise.
lambda	Nonnegative multiplier for "diffusion", 1 by default. Diffusion degree is the sum of the selected degree of a vertex and all of its one-step neighbours, multiplied by lambda.
plot	Whether to draw the result as well as return it. Drawing is a side effect in the manner of <code>graphics::hist()</code> : the verb still returns its tidy table, invisibly when it has drawn, so <code>plot = TRUE</code> saves the wrapping <code>plot()</code> call without changing what comes back. Use <code>plot()</code> on the result when the figure needs arguments of its own.

Details

step and window are separate on purpose. step is how often you look; window is how much of the timeline each look takes in. Setting them equal partitions the period; setting window larger than step is a rolling window, which keeps the resolution of the smaller step while smoothing over the noise of a sparse bin. The arguments match `tsna::tSnaStats()`, where they are called `time.interval` and `aggregate.dur`.

Snapshot "degree" counts distinct active binary dyads, so duplicate, split and overlapping spells do not multiply it; mode = "all" on a directed snapshot is in-degree plus out-degree. "strength" is the same margin taken over spell weights rather than over binary dyads. In a positive window each spell contributes its weight in proportion to the share of its duration that falls inside the window, $\text{weight} * \text{overlap} / \text{duration}$, so a spell straddling two tiled windows splits its weight between

them and the pieces add back to the whole. A point contact has no duration to split and contributes its full weight to the window holding it. With `window = 0` every active spell contributes its full weight at that instant. The share uses the spell's recorded duration, so the part of a spell outside the observation period is not reassigned to observed windows. `snapshots()` and the network plots keep full weights per bin, so their weight column is not the input to this strength.

Snapshot "closeness" is **not** Freeman's $1/\sum_z d_{sz}$, which is undefined once a snapshot is disconnected – and a time bin almost always is. It is the reciprocal of the mean geodesic distance to the vertices a vertex can actually reach: with R_s the reachable nonself set,

$$C(s) = |R_s| / \sum_{z \in R_s} d_{sz},$$

which is zero for an isolate and equals Freeman's normalised closeness $(n - 1)/\sum_z d_{sz}$ on a connected snapshot. "harary" is the reciprocal of eccentricity, zero for a vertex that cannot reach everything.

"eigenvector", "hub" and "authority" are certified the way eigenvector prestige is: a snapshot whose spectral radius is zero (no cycle) or whose Perron root is repeated (components of equal weight) has no single answer, and every vertex of that block is NA under a warning of class `dynet_eigen_undefined`. "eigenvector" is uniquely determined when the Perron eigenvalue has a one-dimensional eigenspace; strong connectivity is a sufficient condition. Disconnected snapshots with equally dominant components can have more than one correct eigenvector, so read the result as a within-snapshot ranking rather than an automatically comparable number across the whole series.

Indegree prestige is the column sum of the directed binary active-dyad adjacency matrix. It is exactly snapshot degree with `mode = "in"`: duplicate, split, and overlapping spells and edge weights do not multiply the result, while an explicitly retained directed loop contributes once. With `rescale = TRUE`, the column sums are divided by their block total. A zero total is mathematically undefined and is returned as literal NaN.

Row-normalised indegree prestige first converts every nonzero binary adjacency row to sum one; zero rows remain all zero. Its column sums are the received sender-nomination mass, so their total is the number of active senders. `rescale = TRUE` divides again by that block total. This closed-form transform is the `sna::prestige(cmode = "indegree.rownorm")` definition on binary matrices. Dynet deliberately ignores edge weights, whereas `sna` uses their magnitudes on valued matrices.

Row-column-normalised prestige uses deterministic Sinkhorn–Knopp scaling only when the full binary vertex matrix has total support: every active dyad must belong to a perfect matching. It preserves all binary dyads and does not remove isolates or unsupported edges. Infeasible blocks return NA for every vertex with a classed warning. A feasible transform has every incoming column sum equal to one, so raw prestige is uniformly one and rescaled prestige uniformly $1/n$; this definition is a transform diagnostic, not a vertex ranking. Dynet uses fixed-order sweeps, maximum absolute row/column residual $1e-12$, and at most 10,000 sweeps. It never returns a partial iterate. This deliberately differs from the randomised loose-tolerance annealer in `sna` 2.8.

Domain prestige is incoming indegree in the directed reachability graph after excluding its reflexive diagonal. If `H[i, j]` records whether $i = j$ or a directed path from i to j exists, then `p[j] = sum(H[, j]) - 1`. Every distinct reaching vertex counts once, regardless of path length or multiplicity. Loops cannot add self credit, isolates score zero, and a zero-total rescaling returns literal NaN. Closure is computed on the binary active snapshot for each reporting block, not on chronologically ordered temporal journeys through the raw spells.

Domain-proximity prestige additionally uses the shortest incoming hop distances. For the nonself domain `D[j]`, let `r[j]` be its size and `s[j]` the sum of its finite distances into j . The score is zero

when $r[j] = 0$ and otherwise $r[j]^2 / ((n - 1) * s[j])$: the incoming domain fraction divided by mean hop distance. Unreachable vertices are omitted before the distance sum. This deliberately fixes an arithmetic artefact in `sna 2.8`, whose `FALSE * Inf` operation incorrectly zeros partial nonempty domains.

Eigenvector prestige solves $t(B) \% \% p = \rho * p$ for the nonnegative Perron ray of binary adjacency B . It requires positive spectral radius and a one-dimensional Perron eigenspace. Raw scores have Euclidean norm one; `rescale = TRUE` makes their sum one. Zero-radius or nonunique blocks return all NA with a classed warning and diagnostics. Periodic cycles remain valid even when negative or complex roots share the spectral radius. Dynet uses direct eigenvalues plus an SVD nullity/residual check at tolerance $1e-10$, orients the ray as nonnegative, and never applies elementwise absolute value.

Row-normalised eigenvector prestige first forms binary adjacency B and divides each nonzero sender row by its number of distinct outgoing dyads; zero rows remain exactly zero. It then solves the certified incoming Perron equation for the transpose of that row-stochastic matrix. Thus each active sender distributes one unit of recursive nomination mass, with no teleportation or dangling-row imputation. Binary session union and retained loop policy occur before row normalisation. The positive-radius, geometric- uniqueness, nonnegative-sign, L2/sum-scale, warning, and diagnostic rules are otherwise exactly those of ordinary eigenvector prestige.

Column-normalised eigenvector prestige divides each nonzero binary receiver column by its number of distinct incoming dyads; zero columns remain zero. It solves the incoming Perron equation only after that transform. If every vertex has positive indegree, the transformed transpose is row-stochastic and every certified score is necessarily uniform. Nonuniform defined scores therefore require a zero-indegree vertex. Binary union and retained-loop policy precede normalisation; certification and scaling remain those above.

Row-column-normalised eigenvector prestige composes the total-support and deterministic Sinkhorn–Knopp contract with the certified Perron contract. Infeasible support and nonconvergent balancing terminate before the spectral solve. A completed doubly stochastic transform always has the all-ones Perron ray, but reducible transforms have several such rays and remain undefined. Every fully certified score is therefore exactly uniform: $1 / \sqrt{t(n)}$ raw or $1 / n$ rescaled. This selector diagnoses support, balance, and irreducibility; it is not a vertex ranking.

Declared vertex activity induces the eligible vertex population before any kernel is evaluated. Positive windows independently use any-time vertex and edge unions before induction, while `window = 0` evaluates the exact state. Results remain rectangular over the fixed vertex universe: inactive vertices receive typed NA, while eligible isolates keep the centrality kernel’s ordinary static result.

Value

A `dynet_metric`: a tidy data frame with one row per vertex, time point and measure. Columns are session (only under `sessions = "separate"`, which is the only mode that keeps session labels apart), time, node, measure and value. Print it, `summary()` it, `plot()` it, or take the plain frame with `as.data.frame()`. Prestige stores its mathematical choices as direct attributes for a prestige-only result and as named records under `measure_metadata` otherwise. When a prestige variant is structurally undefined or fails to converge, the affected values are NA, a warning says how many reporting blocks were affected, and a record naming the stage and reason for each comes out through `as.data.frame(x, what = "diagnostics")`.

Conditions

Errors: `dynet_unknown_measure` (a measure not listed above), `dynet_needs_directed` ("prestige", "hub", "authority", "indegree" or "outdegree" on an undirected network), `dynet_no_sessions` (sessions = "separate" without a session column), `dynet_outside_observation` (the requested range misses observed support; it also carries `dynet_bad_input`), and `dynet_bad_input` for every other broken contract – `dn` not a `dynet`, an unknown mode, an out-of-range damping, exponent, lambda, prestige, rescale, start, end, step or window, and `rescale = TRUE` without measure = "prestige".

Warnings: `dynet_deprecated` (measure = "indegree"/"outdegree", or the retired sample argument), `dynet_eigen_undefined` and `dynet_kernel_singular` (both also carrying `dynet_measure_undefined`) when a snapshot's eigenvector, hub, authority, Bonacich power or information kernel has no unique answer, and `dynet_prestige_infeasible`, `dynet_prestige_nonconvergence` and `dynet_prestige_eigen_undefined` when a prestige variant is structurally undefined or fails to converge.

References

- Holme, P., & Saramaki, J. (2012). Temporal networks. *Physics Reports*, 519(3), 97-125.
- Wasserman, S., & Faust, K. (1994). *Social Network Analysis: Methods and Applications*. Cambridge University Press, Chapter 5.
- Butts, C. T. (2024). *sna: Tools for Social Network Analysis*, version 2.8. doi:10.32614/CRAN.package.sna.
- Lin, N. (1976). *Foundations of Social Research*. McGraw-Hill.
- Freeman, L. C. (1979). Centrality in social networks: conceptual clarification. *Social Networks*, 1(3), 215-239. doi:10.1016/03788733(78)900217
- Brandes, U. (2001). A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2), 163-177.
- Bonacich, P. (1987). Power and centrality: a family of measures. *American Journal of Sociology*, 92(5), 1170-1182.
- Hage, P., & Harary, F. (1995). Eccentricity and centrality in networks. *Social Networks*, 17(1), 57-63.
- Stephenson, K., & Zelen, M. (1989). Rethinking centrality. *Social Networks*, 11(1), 1-37.
- Goh, K.-I., Kahng, B., & Kim, D. (2001). Universal behavior of load distribution in scale-free networks. *Physical Review Letters*, 87(27), 278701.
- Freeman, L. C., Borgatti, S. P., & White, D. R. (1991). Centrality in valued graphs. *Social Networks*, 13(2), 141-154.
- Page, L., Brin, S., Motwani, R., & Winograd, T. (1999). *The PageRank citation ranking: bringing order to the web*. Technical Report 1999-66, Stanford InfoLab.
- Kleinberg, J. M. (1999). Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5), 604-632. doi:10.1145/324133.324140.
- Seidman, S. B. (1983). Network structure and minimum degree. *Social Networks*, 5(3), 269-287. doi:10.1016/0378-8733(83)90028-X.
- Burt, R. S. (1992). *Structural Holes: The Social Structure of Competition*. Harvard University Press.

Kundu, S., Murthy, C. A., & Pal, S. K. (2011). A new centrality measure for influence maximization in social networks. In *Pattern Recognition and Machine Intelligence*, Lecture Notes in Computer Science 6744 (pp. 242-247). Springer. doi:10.1007/978-3-642-21786-9_40.

Bonacich, P. (1972). Factoring and weighting approaches to status scores and clique identification. *Journal of Mathematical Sociology*, 2, 113-120. doi:10.1080/0022250X.1972.9989806.

Berman, A., & Plemmons, R. J. (1994). *Nonnegative Matrices in the Mathematical Sciences*. SIAM. doi:10.1137/1.9781611971262.

Sinkhorn, R. (1964). A relationship between arbitrary positive matrices and doubly stochastic matrices. *Annals of Mathematical Statistics*, 35, 876-879. doi:10.1214/aoms/1177703591.

Sinkhorn, R., & Knopp, P. (1967). Concerning nonnegative matrices and doubly stochastic matrices. *Pacific Journal of Mathematics*, 21, 343-348. doi:10.2140/pjm.1967.21.343.

Knight, P. A. (2008). The Sinkhorn-Knopp algorithm: convergence and applications. *SIAM Journal on Matrix Analysis and Applications*, 30, 261-275. doi:10.1137/060659624.

See Also

`path_centrality()` for closeness and betweenness on time-respecting paths; `reachability()` for temporal reach.

Examples

```
dn <- dynet(school_contacts)

centrality_series(dn, measure = "degree")
centrality_series(dn, measure = c("degree", "betweenness"))
centrality_series(dn, measure = "prestige", rescale = TRUE)
centrality_series(dn, measure = "prestige",
  prestige = "indegree.rownorm")
centrality_series(dn, measure = "prestige", prestige = "domain")
centrality_series(dn, measure = "prestige",
  prestige = "domain.proximity")
centrality_series(dn, measure = "prestige", prestige = "eigenvector")
centrality_series(dn, measure = "prestige",
  prestige = "eigenvector.rownorm")
centrality_series(dn, measure = "prestige",
  prestige = "eigenvector.colnorm")
centrality_series(dn, measure = "prestige",
  prestige = "eigenvector.rowcolnorm")

# A seven-day window, stepped one day at a time.
centrality_series(dn, measure = "degree", step = 1, window = 7)

degree <- centrality_series(dn, measure = "degree")
summary(degree)
```

clear_observations	<i>Restore implicit observation support</i>
--------------------	---

Description

Restore implicit observation support

Usage

```
clear_observations(dn)
```

Arguments

dn A temporal network.

Value

A new dynet object, class `c("dynet", "netobject", "cograph_network")`, observed continuously from its earliest raw start through its latest raw end. Every explicit observation field is dropped from the metadata and the bin count is recomputed over the raw range; spells and attributes are untouched. Safe on a network that never had explicit observations, which is returned with only its recorded call changed.

Examples

```
dn <- dynet(school_contacts)
first_week <- set_observations(dn, start = 0, end = 7)
restored <- clear_observations(first_week)
restored
```

collapse_network	<i>Collapse temporal activity to a static weighted network</i>
------------------	--

Description

Creates a static cograph network from the exact observed, endpoint-valid activity in a requested time range. Every collapsed edge retains all common duration summaries, so choosing one weighting does not discard the others.

Usage

```
collapse_network(
  dn,
  start = NULL,
  end = NULL,
  weight = c("binary", "union_duration", "total_duration", "duration_fraction",
    "spell_count", "weight_sum", "weighted_duration", "latest_weight"),
  sessions = c("bounded", "collapse", "separate"),
  censored = c("include", "exclude")
)
```

Arguments

<code>dn</code>	A temporal network from <code>dynet()</code> or <code>as_dynet()</code> .
<code>start, end</code>	Collapse bounds. Default to the observed range. Positive intervals are clipped to <code>[start, end]</code> ; genuine points at either bound are retained. An end before start raises a <code>dynet_bad_input</code> error.
<code>weight</code>	Edge field used as the cograph weight: "binary" (the default), "union_duration", "total_duration", "duration_fraction", "spell_count", "weight_sum", "weighted_duration", or "latest_weight". Every field is present in the edge table whichever one is chosen; this names only the one cograph draws with.
<code>sessions</code>	Session handling. "bounded", the default, respects session-specific endpoint activity before pooling, "collapse" erases session labels, and "separate" returns one collapsed cograph network per session.
<code>censored</code>	Whether raw edge and vertex identities carrying an explicit censor flag are "include"d, the default, or "exclude"d. Exclusion drops the whole raw identity, never one observed fragment alone.

Value

A `dynet_collapsed` cograph netobject, whose two tidy tables are reached with `as.data.frame(x, what = "edges")` and `as.data.frame(x, what = "nodes")`. With `sessions = "separate"`, a named `dynet_collapsed_list` of such objects, one per session.

The edge table carries one row per collapsed pair and every weighting at once, so choosing one does not discard the others: `from`, `to`, `binary` (1 for a pair that was ever active), `union_duration` (time the pair was active, overlaps counted once), `total_duration` (summed spell lengths, overlaps counted twice), `duration_fraction` (`union_duration` over the pair's joint activity opportunity, NA when that opportunity is zero), `spell_count`, `weight_sum`, `weighted_duration` (weight times duration, summed), `latest_weight` (the weight of the last spell to end), `first` and `last` (the pair's earliest onset and latest terminus), and the `activity.duration` and `activity.count` aliases for compatibility with `networkDynamic::network.collapse()`.

The node table carries one row per vertex, with `name`, any static vertex attributes the network was built with, `activity_duration` (time the vertex was active, overlaps counted once) and its `activity.duration` alias.

Examples

```
dn <- dynet(data.frame(
  from = c("A", "A"), to = c("B", "B"),
  start = c(0, 1), end = c(2, 3)
))
flat <- collapse_network(dn, weight = "union_duration")
as.data.frame(flat)
```

durations

How long each relationship lasted

Description

Pair unit returns one row per vertex pair and measure, summarising every retained raw spell they shared. Spell unit returns each retained raw edge identity. Vertex-activity unit returns fixed-universe vertex summaries; vertex-spell unit returns canonical vertex-activity components. Duration is what separates an interval network from a contact network: a pair that met fifty times briefly and a pair that met once at length have the same edge weight in a static network and nothing else in common.

Usage

```
durations(
  dn,
  measure = c("events", "total", "mean"),
  sessions = c("bounded", "collapse", "separate"),
  censored = c("include", "exclude"),
  unit = c("pair", "spell", "vertex_activity", "vertex_spell", "node_ties"),
  mode = c("out", "in", "all"),
  plot = FALSE
)
```

Arguments

dn	A temporal network from <code>dynet()</code> .
measure	For pair unit, one or more of "events" (number of spells), "total" (summed duration), "union" (binary pair occupancy), "mean", "median", "first", and "last"; its default is <code>c("events", "total", "mean")</code> . For spell unit, one or more of "duration", "first", and "last"; its default is "duration". Vertex-activity unit allows the pair-like measures and defaults to "events", "total", and "union"; vertex-spell unit allows the same measures as edge spell and defaults to "duration". Node-ties unit allows "events" (incident raw-spell endpoint stubs), "total" (their summed endpoint-valid duration), and "union" (binary incident calendar exposure), defaulting to events and total. A measure the chosen unit does not offer raises a <code>dynet_unknown_measure</code> error.
sessions	How to treat sessions: "bounded" (the default), "collapse" or "separate", as in <code>centrality_series()</code> .

censored	Whether to "include" known follow-up, the default, or "exclude" an entire edge raw spell or canonical vertex component with either explicit outer censor flag. Administrative observation cuts never cause exclusion.
unit	"pair", the default, retains the existing pair summary and adds union duration; "spell" returns one row per retained raw edge-spell identity; "vertex_activity" returns fixed-node aggregates; "vertex_spell" returns retained canonical vertex-activity identities; "node_ties" returns fixed-node incident-tie quantities.
mode	For unit = "node_ties", "out" (the default), "in", or "all" endpoint incidence. Undirected networks normalise every request to "all". Supplying mode explicitly for any other duration unit raises a <code>dynet_incompatible_duration_mode</code> error; leaving it at its default is what makes the other units legal.
plot	Whether to draw the result as well as return it. Drawing is a side effect in the manner of <code>graphics::hist()</code> : the verb still returns its tidy table, invisibly when it has drawn, so <code>plot = TRUE</code> saves the wrapping <code>plot()</code> call without changing what comes back. Use <code>plot()</code> on the result when the figure needs arguments of its own.

Details

Positive spell duration is the observed time during which both endpoints are eligible. Genuine eligible point contacts are retained with duration zero. Pair total sums these raw-spell durations, so overlapping identities intentionally multiply time; pair union counts binary calendar occupancy once. Consequently $\text{union} \leq \text{total}$, and union cannot exceed the pair's eligible opportunity time. Pair events counts retained raw identities. Formally, if retained raw spell i has endpoint-valid fragments $F[i]$, then $\text{duration}[i] = \sum((b - a) \text{ for } [a, b) \text{ in } F[i])$. For pair p , $\text{total}[p] = \sum(\text{duration}[i])$, while $\text{union}[p]$ is the Lebesgue measure of the calendar union of every positive fragment belonging to p . Point contacts therefore count as events and spells but contribute zero duration. These conventions follow the spell and dyad distinction in `tsna::edgeDuration()` (Butts, 2024, doi:10.32614/CRAN.package.tsna), with Dynet additionally applying its observation and endpoint-eligibility contract.

Collapse erases edge and vertex session labels before gating. Bounded gates within each session and then pools spell identities while unioning overlapping pair occupancy once on the shared calendar. Separate returns local blocks. Weights and vertex censor flags do not affect durations. Excluding raw edge censoring removes the whole identity, never only an observed fragment.

For a retained canonical vertex component k , let $S[k]$ be its observed support, $d[k]$ its total positive width, and $f[k]$ and $l[k]$ its extrema. Vertex total is $\sum(d[k])$, while union is the measure of the calendar union of every positive $S[k]$; therefore $0 \leq \text{union} \leq \text{total}$. Points count as identities with zero duration. A declared vertex with no retained support has zero events/total/union and missing mean/median/first/last. A wholly undeclared vertex has one measurement-only implicit always-active identity over observed support. This is stream-graph node presence duration as in Latapy, Viard, and Magnien (2018), doi:10.1007/s13278-018-0537-7, and agrees with `tsna::vertexDuration()` only under matching continuous-observation, non-session conventions.

Node-ties uses the same endpoint-valid raw spell supports as pair/spell duration. Directed out credits the tail, in credits the head, and all adds both endpoint stubs. A retained loop therefore contributes once to out, once to in, and twice to additive all-mode events/total; undirected results use the same two-stub rule. In contrast, node-tie union Boolean-unions all positive incident fragments, so loops, reciprocal overlap, duplicate rows, and simultaneous neighbours occupy calen-

dar time only once. Consequently $\text{union} \leq \text{total}$, and directed all equals out plus in only for events and total. Formally, for endpoint-stub multiplicity $c[v, i, m]$, retained raw identity duration $d[i]$, and positive support $F[i]$, node-tie events are $\text{sum}(c[v, i, m])$, total is $\text{sum}(c[v, i, m] * d[i])$, and union is the measure of the calendar union of all $F[i]$ having positive multiplicity. These union values cannot exceed the corresponding eligible vertex-activity union. Isolates, inactive vertices, and loopless singletons receive exact zeros for every node-tie measure. The additive quantities match `tsna::tiedDuration()` only for continuous observation, static eligible endpoints, uncensored matched spells, and no sessions; `tsna` is not an oracle for union, gaps/points, endpoint schedules, source censor filtering, or session policies (Bender-deMoll and Morris, 2025, doi:10.32614/CRAN.package.tsna).

Value

A `dynet_metric`. Pair and edge-spell units are edge-level: pair has columns `from`, `to`, `measure`, and `value`, while spell additionally has `raw_spell`. Administrative observation and endpoint-activity fragments are recombined by raw spell before the censored policy is applied. Vertex-activity unit has `node`, `measure`, and `value`; vertex-spell additionally has `vertex_spell` and `implicit`; both vertex units are node-level metrics. Node-ties is also node-level and has the fixed schema `node`, `measure`, and `value` (plus `session` only for separate mode).

Examples

```
dn <- dynet(school_contacts)
durations(dn)
durations(dn, measure = "union")
durations(dn, unit = "spell", measure = "duration")
durations(dn, unit = "vertex_activity")
durations(dn, unit = "vertex_spell")
durations(dn, unit = "node_ties", mode = "all")
tie_durations <- durations(dn)
summary(tie_durations, by = "measure")
```

dynet

Build a temporal network

Description

Builds a temporal network from a relational log. One constructor covers the four shapes relational data actually arrives in, and the shape is inferred from the arguments you name:

interval Each row is an edge active over $[start, end)$. Used when the data carry an end time or a duration.

contact Each row is an instantaneous event with a time and no duration – a message, a click, a citation.

threaded Forum, chat or email data. An edge is treated as active from its own post until the last post in the same thread, following Saqr and Nouri (2020). Name the thread argument to select this.

copresence Two-mode attendance data. Actors sharing a group become connected for the span of that group. Name actor and group.

Every other column of data is kept as a tie attribute: it appears in `as.data.frame()` and can be selected on with `induce_subgraph(ties = )`. The exceptions are the canonical spell fields themselves – `duration`, `weight`, `session`, `thread`, `onset_censored` and `terminus_censored` – which are dropped even when they were never named as arguments, because the spell table owns those names. A non-atomic column raises `dynet_bad_tie_attribute`, and a factor is carried as character. Co-presence logs keep none, because their rows are memberships rather than ties.

Column names are resolved case-insensitively from a table of aliases, so `Sender/Receiver`, `source/target` and `onset/terminus` are all understood without being spelled out. Times may be numeric, `Date`, `POSIXct` or character date-time strings; character and date-time input is converted to elapsed time since the first event in a readable unit.

Vertices are addressed by name everywhere in this package. Integer vertex indices are used internally for speed but are never part of any result.

Explicit observation bounds are administrative measurement limits, not a destructive filter. A positive spell $[s, e)$ contributes the half-open intersection $[\max(s, L), \min(e, U))$ when it has positive duration, while a genuine instantaneous event is retained at either L or U . Original endpoints remain the only formation and dissolution events, censoring is never inferred from equality with a limit, and temporal paths must both start and finish inside the declared interval.

Vertex activity is a separate declaration. A vertex with at least one row in `vertex_spells` is eligible only on the union of those half-open positive spells and exact points; a vertex with no row remains eligible at all times. Snapshot measurements independently union eligible vertices and active edges over each positive window and then induce on eligible endpoints; point snapshots evaluate both at the exact time. Explicit vertex censor flags describe raw outer-boundary state and are never inferred from observation limits or used to alter eligibility.

Usage

```
dynet(
  data,
  from = NULL,
  to = NULL,
  start = NULL,
  end = NULL,
  duration = NULL,
  time = NULL,
  thread = NULL,
  actor = NULL,
  group = NULL,
  session = NULL,
  weight = NULL,
  nodes = NULL,
  groups = NULL,
  format = c("auto", "interval", "contact", "threaded", "copresence"),
  thread_clock = c("absolute", "relative"),
  directed = TRUE,
  interval = 1,
```

```

    time_unit = "auto",
    observation_start = NULL,
    observation_end = NULL,
    observation_spells = NULL,
    loops = FALSE,
    min_thread_posts = 1L,
    onset_censored = NULL,
    terminus_censored = NULL,
    vertex_spells = NULL
)

```

Arguments

<code>data</code>	Data frame holding one relational event per row.
<code>from, to</code>	Column names for the source and target vertex. Auto-detected from <code>from/to</code> , <code>source/target</code> , <code>sender/receiver</code> , <code>tail/head</code> , <code>ego/alter</code> .
<code>start, end</code>	Column names for the start and end of an edge spell. Auto-detected from <code>start/end</code> , <code>onset/terminus</code> , <code>begin/finish</code> .
<code>duration</code>	Column name for a spell duration, used in place of <code>end</code> .
<code>time</code>	Column name for an event time, used in place of <code>start</code> . Auto-detected from <code>time</code> , <code>timestamp</code> , <code>date</code> , <code>datetime</code> .
<code>thread</code>	Column name identifying a conversation thread. Naming it selects the threaded format.
<code>actor, group</code>	Column names for the actor and the shared group. Naming both selects the co-presence format.
<code>session</code>	Column name for a session or period grouping. Sessions act as walls that time-respecting paths do not cross.
<code>weight</code>	Column name for event multiplicity. <code>NULL</code> auto-detects a column named <code>weight</code> , <code>weights</code> or <code>strength</code> (and says so); with <code>none</code> , every row counts once.
<code>nodes</code>	Optional data frame of vertex attributes. The vertex key is auto-detected (<code>node</code> , <code>vertex.id</code> , <code>id</code> , <code>name</code> , ...), or given as the first column. When the key is not <code>name</code> and the table also has a <code>name</code> column, the vertices are named by <code>name</code> : edge endpoints and vertex spells given by key are translated, and the key stays on the node table as an attribute. A key with no row in <code>nodes</code> keeps the key as its name, with a <code>dynet_unnamed_nodes</code> warning.
<code>groups</code>	Name of a column in <code>nodes</code> to use as the vertex partition. Written into the places <code>cograph</code> looks for it, so <code>cograph::splot()</code> colours and groups by it without further argument. A name that is not a column of <code>nodes</code> raises a condition of class <code>dynet_unknown_attribute</code> .
<code>format</code>	One of <code>"auto"</code> (the default), <code>"interval"</code> , <code>"contact"</code> , <code>"threaded"</code> , <code>"copresence"</code> . <code>"auto"</code> infers the format from the arguments you name and the columns present.
<code>thread_clock</code>	For a threaded log, <code>"absolute"</code> (the default) keeps every post on the calendar; <code>"relative"</code> puts each thread on its own clock, measured from the thread's first post, so a tie opens at the time since its thread began and closes when the thread ends. Threads are then comparable by how they unfold rather than by when they happened, the convention of the <i>Trees of Thought</i> study. Requires <code>thread</code> .

<code>directed</code>	Whether edges are directed, TRUE by default. Co-presence networks are always undirected.
<code>interval</code>	Width of one time bin, in the network's time unit. Defaults to 1.
<code>time_unit</code>	Unit for converting Date/POSIXct/character times: "auto" (the default), "seconds", "minutes", "hours", "days" or "weeks". Numeric times are left alone and reported as "step".
<code>observation_start, observation_end</code>	Optional bounds of the continuous observation interval. Supply numeric values in the network's internal time scale, or Date/POSIXct values for a calendar network. Either bound may be omitted, in which case the corresponding raw event limit is used. Positive spells are measured on their half-open intersection with this interval; instantaneous events are retained at either boundary. Raw spell endpoints returned by <code>as.data.frame()</code> are never changed.
<code>observation_spells</code>	Optional data frame with exactly two columns, <code>start</code> and <code>end</code> , defining discontinuous observed support. Overlapping and adjacent positive intervals are merged; isolated points are retained. This is mutually exclusive with <code>observation_start</code> and <code>observation_end</code> ; supplying both raises a condition of class <code>dynet_conflicting_observation</code> .
<code>loops</code>	Whether to keep self-loops. FALSE, the default, drops them with a message, which is almost always what relational logs need; TRUE keeps them and reports how many. A kept loop is counted by degree, and contributes two to it, since both of its endpoint stubs are incident to the same vertex. In a threaded log a dropped self-reply is dropped before the thread's lifetime is computed, so it neither opens a tie nor keeps its thread alive.
<code>min_thread_posts</code>	For a threaded log, the smallest number of posts a thread must hold, after self-loops have been dropped, for its posts to enter the network. The default 1 keeps every thread; 2 drops threads that never became an exchange, the rule of Saqr (2024). Dropped threads are reported with a message. Requires <code>thread</code> .
<code>onset_censored, terminus_censored</code>	Optional logical column names for explicit raw interval-boundary censor state. These selectors are available only for interval input, are never auto-detected, and may not flag a zero-duration point.
<code>vertex_spells</code>	Optional tidy vertex-activity table: one row per period in which a vertex is present, with a <code>node</code> column (<code>node</code> , <code>vertex.id</code> , <code>name</code> , ...) and a <code>start</code> and <code>end</code> column (<code>start/end</code> , <code>onset/terminus</code> , ...), resolved through the same alias table as <code>data</code> , plus optional exact columns <code>session</code> , <code>onset_censored</code> , and <code>terminus_censored</code> . Any other column is ignored, so a node table that carries entry and exit times can be passed as it is. Positive spells use <code>[start, end)</code> and points are exact. Overlapping and adjacent positive spells are unioned independently by node and session. A vertex absent from this table remains active at all times.

Value

An object of class `c("dynet", "netobject", "cograph_network")`. It is a cograph network, so `cograph::splot()` draws it directly and every cograph rendering argument applies. Use `as.data.frame()`

for the tidy spell table, `as.data.frame(x, what = "nodes")` for the vertex table, `as.data.frame(x, what = "network")` for the aggregate edge list, `summary()` for the description and `plot()` for a picture. Nothing in this package requires you to reach into the object.

References

- Saqr, M., & Nouri, J. (2020). High resolution temporal network analysis to understand and improve collaborative learning. *Proceedings of the Tenth International Conference on Learning Analytics & Knowledge*, 314-319.
- Holme, P., & Saramaki, J. (2012). Temporal networks. *Physics Reports*, 519(3), 97-125.
- Butts, C. T. (2008). network: a package for managing relational data in R. *Journal of Statistical Software*, 24(2), 1-36.

Examples

```
# An interval log: each row carries its own start and end
dynet(school_contacts)

# A threaded log: edge stays active until its thread falls silent
dynet(forum_posts, thread = "thread")

# A co-presence log: actors sharing a group become connected
dynet(seminar_attendance, actor = "student", group = "seminar")

# Declare observation time without rewriting the source spell.
bounded <- dynet(data.frame(
  from = "A", to = "B", start = -2, end = 8
), observation_start = 0, observation_end = 5)
as.data.frame(bounded)

# Declare changing vertex eligibility without altering edge spells.
scheduled <- dynet(data.frame(
  from = "A", to = "B", start = 0, end = 10
), vertex_spells = data.frame(
  node = c("A", "A"), start = c(0, 7), end = c(4, 10)
))
as.data.frame(scheduled, what = "vertex_spells")
```

events

Edge formation and dissolution over time

Description

When relationships are born and when they die. In a static network every edge is present at once; here the turnover itself is the finding. A course typically shows formation front-loaded and dissolution piling up at the end, and a group that never dissolves an edge is behaving differently from one that constantly re-forms them.

Usage

```
events(
  dn,
  measure = c("formation", "dissolution"),
  sessions = c("bounded", "collapse", "separate"),
  start = NULL,
  end = NULL,
  step = NULL,
  window = NULL,
  plot = FALSE
)
```

Arguments

dn	A temporal network from <code>dynet()</code> .
measure	One or more of "formation" (spells beginning in the bin), "dissolution" (spells ending in the bin), "active" (spells alive during the bin), "new_pairs" (vertex pairs meeting for the first time), "formation_fraction" (confirmed binary pair formations divided by their exact two-sided inactive risk set), "dissolution_fraction" (confirmed binary pair dissolutions divided by their exact two-sided active risk set), "formation_rate" (confirmed formations divided by exact integrated inactive eligible pair-time), and "dissolution_rate" (confirmed dissolutions divided by exact integrated active eligible pair-time). Defaults to <code>c("formation", "dissolution")</code> . Anything else raises a <code>dynet_unknown_measure</code> error. The two fractions need <code>window = 0</code> (<code>dynet_transition_requires_instant</code> otherwise) and the two rates need a positive window (<code>dynet_rate_requires_positive_window</code>), so asking for a fraction and a rate in one call raises <code>dynet_incompatible_transition_windows</code> .
sessions	How to treat sessions: "bounded" (the default), "collapse" or "separate", as in <code>centrality_series()</code> .
start, end	First and last time at which to measure. Default to the observed range. A network built from dates may be addressed with dates.
step	How often to measure. Defaults to the interval the network was built with.
window	How much time each measurement covers. Defaults to <code>step</code> , which tiles the period into disjoint bins. A larger value slides an overlapping window; <code>0</code> samples the network at each point in time. "all" measures the whole observed period as one window, closed on the right so an event at the final instant is inside it; naming <code>step</code> as well raises a <code>dynet_bad_input</code> error, and under <code>sessions = "separate"</code> or discontinuous observation it gives one window per session or observed component.
plot	Whether to draw the result as well as return it. Drawing is a side effect in the manner of <code>graphics::hist()</code> : the verb still returns its tidy table, invisibly when it has drawn, so <code>plot = TRUE</code> saves the wrapping <code>plot()</code> call without changing what comes back. Use <code>plot()</code> on the result when the figure needs arguments of its own.

Details

Formation and dissolution are counted inside each window, so overlapping windows ($\text{window} > \text{step}$) count the same event more than once by design – that is what a rolling total is. Setting window equal to step, the default, gives disjoint counts that sum to the total turnover. Explicitly onset-censored raw limits are not formations, and explicitly terminus-censored limits are not dissolutions. A left-censored observed tie is prior evidence for new_pairs; raw censor state never changes activity.

Formation fraction is defined only with $\text{window} = 0$. For a positive half-open interval $[s, e)$, its pre-batch state at t is $s < t \leq e$ and its post-batch state is $s \leq t < e$. These predicates are binary-unioned per nonloop ordered pair or undirected dyad after the entire timestamp batch. Points are absent on both sides. A pair enters risk only when observation and both endpoints are eligible immediately before and after t and the pair is inactive before. A formation is confirmed when it is active after and at least one contributing positive raw spell has a known onset at t . The ratio is in $[0, 1]$; zero risk returns NA.

Duplicate, overlapping, or adjacent raw spells cannot multiply pair-state transitions. Observation and vertex boundaries are excluded by two-sided eligibility. Onset censoring suppresses confirmation but not state; terminus censoring, weights, loops, and point contacts do not contribute. Collapse erases labels, bounded authorises within sessions before unioning each calendar pair, and separate returns session-local fractions.

Dissolution fraction is the dual exact-time quantity. For each nonloop pair, let E^- and E^+ be binary-union state on the symbolic one-sided limits and let L mean at least one positive raw spell ends exactly at the timestamp with a known terminus. The numerator is $Z * E^- * (1 - E^+) * L$, where Z requires two-sided observation and endpoint eligibility; the denominator is $\text{sum}(Z * E^-)$, including pairs that remain active. Zero risk returns NA_real_, while positive risk with no confirmed dissolution returns zero. Censor flags do not change state: one known terminus confirms a disappearance but an all-censored disappearance is unconfirmed. Duplicate, overlapping, adjacent, and tied rows are unioned; points, loops, weights, onset censoring, and administrative observation/activity boundaries do not create transitions. Collapse erases labels, bounded unions authorised session-local states, and separate reports local rows. Positive windows are rejected because "dissolution_rate" owns dissolution rates.

Dissolution rate is the active-risk dual over a positive window. Its numerator sums confirmed binary pair dissolutions at included timestamp batches; its denominator integrates exact eligible active nonloop pair-time over observation, vertex, edge, and window change cells. Right-censored termini retain state and exposure but do not confirm an event, while one known duplicate suffices. Zero active exposure returns NA_real_; positive exposure without a confirmed dissolution is zero. The unit is inverse network time. It is not raw terminus intensity, spell-duration sum, or an average of instantaneous fractions; positive windows are required, and this is the rate "dissolution_rate" reports.

Formation rate is the positive-window counterpart. Its numerator sums the confirmed binary pair formations at each included timestamp, while its denominator integrates exact inactive eligible nonloop pair-time over change-point cells cut by the window, observation components, vertex activity, and edge state. It is not an average of instantaneous fractions, a raw-onset intensity, or an ever-observed-pair quantity. Zero exposure returns NA_real_; positive exposure with no confirmed formation returns zero. The unit is inverse network time and scales inversely with positive time scaling. Points have zero exposure, onset censoring suppresses only confirmation, and gap/boundary, duplicate, overlap, adjacency, loop, weight, and session rules follow the same ledger

as "formation_fraction". window = 0 is rejected because that measure owns the instantaneous fractions.

Value

A dynet_metric at graph level, one row per time point and measure.

References

Andersen, P. K., & Gill, R. D. (1982). Cox’s regression model for counting processes: a large sample study. *Annals of Statistics*, 10, 1100-1120. doi:10.1214/aos/1176345976

Butts, C. T., Leslie-Cook, A., Krivitsky, P. N., & Bender-deMoll, S. (2024). *networkDynamic: Dynamic Extensions for Network Objects*, version 0.11.5. doi:10.32614/CRAN.package.networkDynamic

Examples

```
dn <- dynet(school_contacts)
events(dn)
turnover <- events(dn, measure = c("formation", "dissolution"))
plot(turnover)
events(dn, measure = "formation_fraction", start = 1, end = 1,
       window = 0)
events(dn, measure = "dissolution_fraction", start = 1, end = 1,
       window = 0)
events(dn, measure = "formation_rate", start = 1, end = 2,
       window = 1)
events(dn, measure = "dissolution_rate", start = 1, end = 2,
       window = 1)
```

forum_people	<i>People in the discussion forum</i>
--------------	---------------------------------------

Description

Vertex attributes for forum_posts. Passed to dynet() through its nodes argument, these become available to mixing().

Usage

forum_people

Format

- A data.frame with 20 rows and 3 columns:
- name** Character. Matches the sender and receiver names in forum_posts.
 - role** Character. "Student" (16), "Teacher" (3) or "Facilitator" (1).
 - achievement** Character. "High", "Middle" or "Low" for students; NA for the four staff.

Source

Simulated, not observed. Generated deterministically under a fixed seed by `data-raw/make-data.R`.

Examples

```
dn <- dynet(forum_posts, thread = "thread", nodes = forum_people)
mixing(dn, attribute = "role")
```

forum_posts	<i>Discussion forum posts</i>
-------------	-------------------------------

Description

Posts in a course discussion forum over roughly eight weeks. Each row is one post directed at an earlier poster in the same thread. Because a post has a timestamp but no end, the duration of the tie has to be derived: `dynet()` treats a post as active until the last post in its thread, so a message that provoked a long argument stays live longer than one that fell flat.

Usage

```
forum_posts
```

Format

A data frame with 241 rows and 4 columns:

sender Character. Who wrote the post.

receiver Character. Who the post replied to.

timestamp POSIXct (UTC). When the post was made. Posts run from 2024-09-02 to 2024-10-27, just under eight weeks.

thread Character. The discussion thread it belongs to; 62 threads.

Details

Pairs with `forum_people`, which carries the roles used for mixing analysis.

Source

Simulated, not observed. Generated deterministically under a fixed seed by `data-raw/make-data.R`.

Examples

```
dynet(forum_posts, thread = "thread", nodes = forum_people)
```

head.dynet_metric	<i>First rows of a temporal measure</i>
-------------------	---

Description

Truncates the rows without rewriting what the measure is. The printed header still describes the series the rows came from, and a `first n of N rows` line records the truncation.

Usage

```
## S3 method for class 'dynet_metric'
head(x, n = 6L, ...)
```

Arguments

<code>x</code>	A <code>dynet_metric</code> .
<code>n</code>	Number of rows to keep. Defaults to six.
<code>...</code>	Passed to the default method.

Value

A `dynet_metric` with at most `n` rows, carrying the source counts so its header stays true to the series.

Examples

```
dn <- dynet(school_contacts)
degree <- centrality_series(dn, step = 4, window = 4)
head(degree)
```

induce_subgraph	<i>Extract an induced temporal subgraph</i>
-----------------	---

Description

Extract an induced temporal subgraph

Usage

```
induce_subgraph(dn, nodes = NULL, ties = NULL, keep_isolates = FALSE)
```

Arguments

<code>dn</code>	A temporal network.
<code>nodes</code>	Which vertices to keep. Either a condition on the vertex table, evaluated the way <code>subset()</code> evaluates one – <code>degree > 20, room == "A" & betweenness > 0</code> – with any centrality it names computed over the whole observed period; or a character vector of names, a factor, a logical mask, or any data frame carrying a name or node column. Only ties whose two endpoints are in this set are eligible. Default <code>NULL</code> , meaning every vertex.
<code>ties</code>	Which ties to keep. Either a condition on the spell table, evaluated the way <code>subset()</code> evaluates one – <code>course == "g1", duration > 2 & weight >= 1</code> – over the columns as <code>.data.frame(dn)</code> returns, tie attributes included; or integer row positions or a logical mask over that same table, in that order, a mask having exactly as many elements as there are spells. Default <code>NULL</code> , meaning every tie. At least one of <code>nodes</code> and <code>ties</code> must be supplied.
<code>keep_isolates</code>	Whether named nodes without a selected tie remain. Default <code>FALSE</code> ; it has an effect only when <code>nodes</code> is supplied.

Value

A new `dynet` object, class `c("dynet", "netobject", "cograph_network")`, carrying only the selected spells, the vertices they touch (plus any isolate named in `nodes` when `keep_isolates = TRUE`), those vertices' activity spells, and all static node and tie attributes. Metadata is rebuilt, so the observed range and the canonical spell identifiers describe the subgraph, not the parent. Raises `dynet_unknown_node` for a name that is not a vertex, `dynet_empty_network` when the selection leaves no vertex or no tie, and `dynet_bad_input` when neither `nodes` nor `ties` is supplied or a selection is malformed.

Examples

```
dn <- dynet(school_contacts)

# A condition on the vertex table, in one call.
induce_subgraph(dn, degree > 16)

# Names still work, as does anything carrying them.
induce_subgraph(dn, nodes = c("Ana", "Ben", "Cara"))
```

Description

Graph properties measured on each time bin, returned as a time series. This is where a temporal network earns its keep: a single density for the whole course tells you nothing about a group that was dense in week two and silent in week five.

Usage

```
metrics(
  dn,
  measure = "density",
  sessions = c("bounded", "collapse", "separate"),
  sample = NULL,
  start = NULL,
  end = NULL,
  step = NULL,
  window = NULL,
  plot = FALSE
)
```

Arguments

<code>dn</code>	A temporal network from <code>dynet()</code> .
<code>measure</code>	One or more measure names, "density" by default: "density", "edges", "active_nodes", "isolates", "transitivity", "reciprocity", "components", "components_strong", "largest_component", "mean_distance", "diameter", "mutual", "asymmetric", "null", "assortativity", "centralization_degree", "centralization_betweenness", "centralization_closeness", "triads", "connectedness", "efficiency", "hierarchy", "lubness". "triads" expands to the sixteen triad classes; those four are Krackhardt's indices of how far a directed network departs from a pure out-tree, only one of which is hierarchy itself. Lightweight structural summaries are "degree_mean", "degree_variance", "degree_min", "degree_max", "mean_degree", "indegree_1_5", "outdegree_1_5", "triangles", "concurrent_nodes", "concurrent_share", "in_2stars", "out_2stars", and "two_paths". Exact window-integrated quantities are "temporal_density", "observed_pair_density", "onset_intensity", and "observed_pair_onset_intensity". Any other name raises an error of class <code>dynet_unknown_measure</code> . Eight of these read direction and need a directed network, raising <code>dynet_needs_directed</code> on an undirected one: "reciprocity", "mutual", "asymmetric", "null", "in_2stars", "out_2stars", "indegree_1_5" and "outdegree_1_5".
<code>sessions</code>	How to treat sessions, as in <code>centrality_series()</code> : "bounded" (the default) keeps each session apart while pooling the reported rows, "collapse" ignores session labels, and "separate" reports each session on its own rows and needs a network built with a session column, raising <code>dynet_no_sessions</code> otherwise.
<code>sample</code>	Deprecated. "instant" is equivalent to <code>window = 0</code> ; "window" uses the current positive/default window.
<code>start, end</code>	First and last time at which to measure. Default to the observed range. A network built from dates may be addressed with dates.
<code>step</code>	How often to measure. Defaults to the interval the network was built with.
<code>window</code>	How much time each measurement covers. Defaults to <code>step</code> , which tiles the period into disjoint bins. A larger value slides an overlapping window; 0 samples the network at each point in time. "all" measures the whole observed period as one window, closed on the right so an event at the final instant is inside it; it

	cannot be combined with <code>step</code> , and under <code>sessions = "separate"</code> or discontinuous observation it gives one window per session or observed component.
<code>plot</code>	Whether to draw the result as well as return it. Drawing is a side effect in the manner of <code>graphics::hist()</code> : the verb still returns its tidy table, invisibly when it has drawn, so <code>plot = TRUE</code> saves the wrapping <code>plot()</code> call without changing what comes back. Use <code>plot()</code> on the result when the figure needs arguments of its own.

Details

"density" counts the any-time union of realised edges against eligible possible edges in each bin. The four temporal selectors instead integrate exact state over positive observed time inside every reporting window. "temporal_density" is binary occupied pair-time divided by all eligible nonloop ordered-pair time (directed) or dyad time (undirected). "observed_pair_density" uses the same numerator but restricts opportunity to pairs having endpoint-valid evidence anywhere in the complete stored history. That cohort is not reset by reporting windows or observation gaps. `summary()` reports the first quantity over the pooled full history.

If $Y[r](t)$ is exact simultaneous endpoint eligibility, $E[r](t)$ is binary edge presence, and H is the ever-observed pair set, the exposure ledgers are $R = \sum(r) \text{ integral}(Y[r](t) dt)$, $O = \sum(r) \text{ integral}(Y[r](t) E[r](t) dt)$, and $R_H = \sum(r \text{ in } H) \text{ integral}(Y[r](t) dt)$. The two occupancies are O/R and O/R_H and lie in $[0, 1]$. Loops, weights, duplicates, and censor flags cannot multiply occupancy. Integration stops at the observation period, which defaults to the span of the data, so a last window reaching past the final spell is measured over its observed part only, and tiled windows pool to the whole-period value. `tsna::tEdgeDensity()` uses the same observation-period rule.

"onset_intensity" and "observed_pair_onset_intensity" divide the number of known raw spell starts by R and R_H . Each nonloop raw row, including a point contact, contributes once when its start is observed, not onset-censored, and has exactly eligible endpoints. Termini are not events; overlaps and duplicates remain distinct onsets. Intensities are nonnegative, unbounded, and measured in inverse network time. A zero denominator gives NA for every temporal selector, even when a point event exists. A positive denominator with a zero numerator gives zero. Therefore `window = 0` makes all four temporal selectors undefined while snapshot measures retain their exact point-state meanings.

`step` and `window` are separate on purpose: `step` is how often you look, `window` is how much of the timeline each look takes in. A seven-day window stepped one day at a time smooths a noisy series without giving up daily resolution. They match `time.interval` and `aggregate.dur` in `tsna::tSnaStats()`.

When vertex activity was declared in `dynet()`, every measure is computed on the endpoint-induced eligible vertex set for the window. Positive windows independently use the any-time vertex and edge unions before induction; `window = 0` evaluates the exact state. Density and census opportunities, components, isolate counts, largest-component shares, and Freeman denominators therefore use eligible rather than fixed order.

"assortativity" is Newman's degree assortativity computed with the **total** degree (in plus out) at both ends of every arc, on directed and undirected snapshots alike. It is not the directed out-degree-to-in-degree variant `igraph::assortativity_degree(directed = TRUE)` reports, and the two disagree on directed data.

"centralization_closeness" is a Freeman centralisation of this package's own closeness (reachable vertices divided by reachable distance, so it is defined on a disconnected snapshot), with the theoretical maximum of that score: $n - 1$ for a directed snapshot (a single arc from the centre) and $n - 2$ for an undirected one (one isolated dyad). It matches `sna::centralization(closeness)` on connected snapshots and not on disconnected ones, where `sna`'s closeness is zero everywhere.

Krackhardt's four indices – "connectedness", "efficiency", "hierarchy" and "lubness" – describe how far a directed network departs from a pure out-tree. "hierarchy" and "lubness" are undefined on some graphs (no connected pair, no component of three) and report NaN rather than a number that would mislead.

Triad census cost grows with the cube of the vertex count. On a network of a few hundred vertices it is the slowest measure here by a wide margin.

The lightweight structural selectors use the binary, loop-free induced snapshot. Directed total degree is in-degree plus out-degree; "degree_variance" is the sample variance across eligible vertices. "mean_degree" is the mean out-degree for a directed graph (identically, the mean in-degree) and the ordinary mean degree for an undirected graph; this matches ERGM's `meandeg` statistic. "indegree_1_5" and "outdegree_1_5" sum the corresponding vertex degrees raised to 1.5. Directed "triangles" is the sum of cyclic and transitive triples, while an undirected triangle is counted once. A concurrent vertex has relations to at least two distinct neighbours active at the same instant, so a reciprocal dyad still supplies only one neighbour. Unlike the other structural selectors, concurrency is not read from the window's union snapshot: with a positive window, two ties that fall in the same window without overlapping in time do not make their shared vertex concurrent. A vertex counts in a window when it is concurrent at any instant inside it; "concurrent_nodes" is the number of such vertices and "concurrent_share" divides it by the window's eligible vertices. With `window = 0` the snapshot is itself an instant, so both readings coincide. Half-open spells that only meet at a boundary do not overlap, and a point contact is concurrent with every relation active at its timestamp. `mixing()` answers a different question – which groups are connected somewhere in the window – and requires no simultaneity. "in_2stars" and "out_2stars" sum `choose(degree, 2)` over directed in- and out-degrees. Directed "two_paths" counts ordered $i \rightarrow j \rightarrow k$ paths with $i \neq k$; undirected two-paths count each unordered wedge once. Empty eligible snapshots return zero for all selectors.

Value

A `dynet_metric` at graph level: one row per time point and measure, with columns `session` (only under `sessions = "separate"`, the one mode that keeps session labels apart), `time`, `measure` and `value`. "triads" contributes sixteen rows per time point, whose measure entries are `triad_003`, `triad_012`, ..., `triad_300`. Print it, `summary()` it, `plot()` it, or take the plain frame with `as.data.frame()`.

Conditions

Errors: `dynet_unknown_measure` (a name outside the forty above), `dynet_needs_directed` (one of the eight direction-reading selectors on an undirected network), `dynet_no_sessions` (`sessions = "separate"` without a session column), `dynet_outside_observation` (the requested range misses observed support; it also carries `dynet_bad_input`), and `dynet_bad_input` for every other broken contract – `dn` not a `dynet`, a non-character measure, an empty measure, an out-of-range start, end, step or window, end before start, and step combined with `window = "all"`.

Warning: `dynet_deprecated` for the retired `sample` argument.

References

- Freeman, L. C. (1979). Centrality in social networks: conceptual clarification. *Social Networks*, 1, 215-239. doi:[10.1016/03788733\(78\)900217](https://doi.org/10.1016/03788733(78)900217)
- Butts, C. T., Leslie-Cook, A., Krivitsky, P. N., & Bender-deMoll, S. (2024). *networkDynamic: Dynamic Extensions for Network Objects*, version 0.11.5. doi:[10.32614/CRAN.package.networkDynamic](https://doi.org/10.32614/CRAN.package.networkDynamic)
- Holme, P., & Saramaki, J. (2012). Temporal networks. *Physics Reports*, 519(3), 97-125. doi:[10.1016/j.physrep.2012.03.001](https://doi.org/10.1016/j.physrep.2012.03.001)
- Latapy, M., Viard, T., & Magnien, C. (2018). Stream graphs and link streams for the modeling of interactions over time. *Social Network Analysis and Mining*, 8, 61. doi:[10.1007/s1327801805377](https://doi.org/10.1007/s1327801805377)
- Andersen, P. K., & Gill, R. D. (1982). Cox's regression model for counting processes: a large sample study. *Annals of Statistics*, 10, 1100-1120. doi:[10.1214/aos/1176345976](https://doi.org/10.1214/aos/1176345976)
- Krackhardt, D. (1994). Graph theoretical dimensions of informal organizations. In *Computational Organization Theory* (pp. 89-111). Lawrence Erlbaum.
- Newman, M. E. J. (2002). Assortative mixing in networks. *Physical Review Letters*, 89, 208701. doi:[10.1103/PhysRevLett.89.208701](https://doi.org/10.1103/PhysRevLett.89.208701)
- Morris, M., & Kretzschmar, M. (1997). Concurrent partnerships and the spread of HIV. *AIDS*, 11(5), 641-648. doi:[10.1097/0000203019970500000012](https://doi.org/10.1097/0000203019970500000012)
- Holland, P. W., & Leinhardt, S. (1976). Local structure in social networks. *Sociological Methodology*, 7, 1-45. doi:[10.2307/270703](https://doi.org/10.2307/270703)
- Wasserman, S., & Faust, K. (1994). *Social Network Analysis: Methods and Applications*. Cambridge University Press.

Examples

```
dn <- dynet(school_contacts)
metrics(dn, measure = "density")
metrics(dn, measure = c("density", "reciprocity", "transitivity"))
metrics(dn, measure = "density", step = 1, window = 3)
dyads <- metrics(dn, measure = c("mutual", "asymmetric"))
plot(dyads)
```

mixing

Mixing between vertex groups over time

Description

How much each kind of vertex interacted with each other kind, in every time bin. This is the question a temporal network answers that a static one cannot: not whether high and low achievers mixed, but *when* they did, and whether the pattern held or decayed.

The grouping variable comes from the vertex attributes supplied to `dynet()` through its `nodes` argument.

Usage

```

mixing(
  dn,
  attribute,
  sessions = c("bounded", "collapse", "separate"),
  sample = NULL,
  start = NULL,
  end = NULL,
  step = NULL,
  window = NULL,
  plot = FALSE
)

```

Arguments

<code>dn</code>	A temporal network from <code>dynet()</code> built with vertex attributes.
<code>attribute</code>	Name of a column in the vertex table. A name the network does not carry raises an error of class <code>dynet_unknown_attribute</code> that lists the attributes it does have.
<code>sessions</code>	How to treat sessions, as in <code>centrality_series()</code> : "bounded" (the default), "collapse" or "separate". "separate" needs a network built with a session column and raises <code>dynet_no_sessions</code> otherwise.
<code>sample</code>	Deprecated. "instant" is equivalent to <code>window = 0</code> ; "window" uses the current positive/default window.
<code>start, end</code>	First and last time at which to measure. Default to the observed range. A network built from dates may be addressed with dates.
<code>step</code>	How often to measure. Defaults to the interval the network was built with.
<code>window</code>	How much time each measurement covers. Defaults to <code>step</code> , which tiles the period into disjoint bins. A larger value slides an overlapping window; 0 samples the network at each point in time. "all" measures the whole observed period as one window, closed on the right so an event at the final instant is inside it; it cannot be combined with <code>step</code> , and under <code>sessions = "separate"</code> or discontinuous observation it gives one window per session or observed component.
<code>plot</code>	Whether to draw the result as well as return it. Drawing is a side effect in the manner of <code>graphics::hist()</code> : the verb still returns its tidy table, invisibly when it has drawn, so <code>plot = TRUE</code> saves the wrapping <code>plot()</code> call without changing what comes back. Use <code>plot()</code> on the result when the figure needs arguments of its own.

Details

Each cell is a raw count of distinct active binary vertex dyads. Repeated, overlapping, or split spells and edge weights do not multiply a dyad. Retained self-loops count once. For directed networks, every ordered group pair is reported and

$$M_{ab} = \sum_{u:g(u)=a} \sum_{v:g(v)=b} Y_{uv}.$$

The row and column margins are grouped outdegree and indegree, and the table sum is the active directed edge count including retained loops.

Undirected networks report one lexicographically canonical cell for each unordered group pair, with display labels such as "A -- B". A within-group edge or loop contributes once to its diagonal cell. The group stub margin is

$$d_a = 2M_{aa} + \sum_{b \neq a} M_{\min(a,b), \max(a,b)},$$

so the margins sum to twice the table total. These are unnormalised counts, not Newman's mixing proportions.

Missing attribute values are retained as a collision-safe explicit group ordered after observed labels. Bounded and collapsed modes both use the binary calendar union: a dyad active in two sessions at the same time counts once. Separate mode returns session-local tables over the fixed group universe. Every supported cell is emitted, including zeros. Declared vertex activity first induces the endpoint-valid snapshot. The complete group-cell universe remains fixed, but inactive vertices and eligible isolates contribute no dyad.

Value

A `dynet_metric` at graph level with one row per time point and group pair. The columns are session (only under `sessions = "separate"`, the one mode that keeps session labels apart), time, measure, value, `from_group` and `to_group`. Directed measure labels use "A -> B"; undirected labels use "A -- B". value is the active binary-dyad count, and the authoritative `from_group` and `to_group` columns identify the cell. Attributes record unit, pair-domain, normalisation, weight, loop, missing-group, and session-aggregation conventions.

Conditions

Errors: `dynet_unknown_attribute` (no such vertex attribute), `dynet_no_sessions` (`sessions = "separate"` without a session column), `dynet_outside_observation` (the requested range misses observed support; it also carries `dynet_bad_input`), and `dynet_bad_input` for every other broken contract – `dn` not a `dynet`, an attribute that is not a single column name, and an out-of-range start, end, step or window.

Warning: `dynet_deprecated` for the retired `sample` argument.

References

- Newman, M. E. J. (2003). Mixing patterns in networks. *Physical Review E*, 67, 026126. doi:10.1103/PhysRevE.67.026126
- Morris, M., Handcock, M. S., & Hunter, D. R. (2008). Specification of exponential-family random graph models: terms and computational aspects. *Journal of Statistical Software*, 24(4). doi:10.18637/jss.v024.i04

Examples

```
dn <- dynet(forum_posts, thread = "thread", nodes = forum_people)
role_mixing <- mixing(dn, attribute = "role")
role_mixing
plot(role_mixing)
```

mooc_people	<i>Participants in a MOOC discussion forum</i>
-------------	--

Description

The participant table accompanying [mooc_posts](#): every person who appears as a sender or a receiver there, with the self-reported experience level as the chapter’s integer code and as the label it recodes the code into and uses as the mixing attribute.

Usage

mooc_people

Format

A data frame with 445 rows and 3 columns:

name Character. Participant identifier, matching the sender and receiver columns of [mooc_posts](#).

experience Integer. Self-reported experience level: 1 expert, 2 student, 3 teacher.

expert_level Character. The same level as Expert, Student or Teacher.

Source

As [mooc_posts](#): Saqr (2024), [doi:10.1007/9783031544644_17](#).

See Also

[mooc_posts](#); vignette("ch17-temporal-networks").

Examples

```
participants <- dynet(mooc_posts, from = "sender", to = "receiver",
                      time = "timestamp", thread = "discussion",
                      nodes = mooc_people)
as.data.frame(participants, what = "nodes")
```

mooc_posts	<i>Posts in a MOOC discussion forum</i>
------------	---

Description

The discussion log of chapter 17 of *Learning Analytics Methods and Tutorials* (Saqr, 2024), one row per post in the Digital Learning Transition MOOC, April to June 2013. A post names the participant who wrote it and the participant it answers, so a tie runs from sender to receiver; discussion is the thread the post belongs to, which is what makes the log threaded in the sense [dynet\(\)](#) means by thread =.

Usage

```
mooc_posts
```

Format

A data frame with 2529 rows and 4 columns:

sender Character. The participant who wrote the post.

receiver Character. The participant the post answers.

timestamp POSIXct (UTC). When the post was made, from 2013-04-04 16:32 to 2013-06-16 17:12.

discussion Character. Thread title; 338 distinct threads.

Details

Only the four columns the chapter's analysis reads are kept; the category hierarchy and comment identifiers of the published file are dropped.

Source

Saqr, M. (2024). Temporal network analysis: Introduction, methods and analysis with R. In M. Saqr & S. López-Pernas (Eds.), *Learning Analytics Methods and Tutorials*. Springer. doi:10.1007/9783031544644_17. Data from <https://github.com/lamethods/data>, directory 6_snaMOOC, prepared by data-raw/mooc_forum.R.

See Also

[mooc_people](#) for the participants, and `vignette("ch17-temporal-networks")` for the chapter's analysis.

Examples

```
dn <- dynet(mooc_posts, from = "sender", to = "receiver",
           time = "timestamp", thread = "discussion")
dn
```

paths

Time-respecting paths from a vertex

Description

Follows every time-respecting path out of (or into) one vertex and reports where it gets to, when, and through whom. A path may only use edges whose timing runs forward, so unlike a path in a flattened network it can never travel back in time.

The source vertex is named, not numbered. `paths(dn, from = "Ana")` works; there is no vertex index to look up first.

At the default zero traversal duration, forward paths use nondecreasing hop times, so relations active at the same instant may form a multi-hop chain. Waiting is allowed. Interval spells are onset-inclusive and terminus-exclusive; point events trigger at their exact timestamp through a distinct event rule. A positive duration separates a hop's trigger or entry from its completion, as detailed below. Reach and arrival do not depend on edge-row order or duplicate spell rows.

Usage

```
paths(
  dn,
  from,
  at = NULL,
  direction = c("forward", "backward"),
  sessions = c("bounded", "collapse", "separate"),
  start = NULL,
  end = NULL,
  traversal_time = 0,
  plot = FALSE
)
```

Arguments

<code>dn</code>	A temporal network from <code>dynet()</code> .
<code>from</code>	Name of the one vertex the search is anchored on: the source of a forward search, the target of a backward one.
<code>at</code>	Forward source-availability time or backward arrival deadline. An explicit <code>at</code> is used exactly: a source that is not present at that instant reaches nothing. The default, <code>NULL</code> , lets the vertex supply its own anchor. A vertex with declared spells (see <code>set_vertex_spells()</code>) starts at the first instant it is present inside the window, or at the last instant searching backward; a vertex with no declared spells starts at the window bound, which is <code>start</code> for a forward search and <code>end</code> for a backward one, each defaulting in turn to the matching end of the observation window. Date and date-time values use the network's time scale. It cannot be combined with <code>start</code> or <code>end</code> .
<code>direction</code>	"forward" traces where the vertex can reach; "backward" traces who could have reached it.
<code>sessions</code>	How to treat sessions, as in <code>path_centrality()</code> .
<code>start, end</code>	Inclusive lower and upper traversal-time bounds. Interval spells remain terminus-exclusive. When these are supplied, use them instead of <code>at</code> .
<code>traversal_time</code>	Nonnegative duration charged for every hop, in the network's time unit. A calendar network also accepts a scalar <code>difftime</code> .
<code>plot</code>	Whether to draw the result as well as return it. Drawing is a side effect in the manner of <code>graphics::hist()</code> : the verb still returns its tidy table, invisibly when it has drawn, so <code>plot = TRUE</code> saves the wrapping <code>plot()</code> call without changing what comes back. Use <code>plot()</code> on the result when the figure needs arguments of its own.

Details

A valid forward journey has distinct vertices, hop-entry times x , and completion times $y = x + \text{traversal_time}$. The source is ready at the resolved origin, each later entry is no earlier than the preceding completion, and final completion is at or before end. At zero duration, entry and completion coincide, recovering the nondecreasing hop times described above. The empty journey reaches the source at the origin. With at , that value is both the origin and the window bound: `start` for forward paths or `end` for backward paths. Cycles are unnecessary for reach and earliest arrival because deleting a repeated-vertex section and waiting at that vertex preserves every later hop.

The origin is anchored at the source's own presence. Without at , a vertex with declared spells starts at the first instant it is present inside the window, or at the last instant when searching backward, so a vertex that enters the network late is never scored from a time before it existed; a vertex with no declared spells starts at the window bound. A vertex that is never present inside the window has no valid anchor, so every row of its result, the source row included, is unreachable. The resolved origin is reported in the printed header; under `sessions = "separate"`, where every session resolves its own, it is reported in the `origin` column instead.

`start` and `end` form a closed bound on the complete journey: entry may equal `start` and completion may equal `end`. This does not close interval activity on the right. At zero duration, an event or interval onset at `end` is eligible while an interval terminating there cannot be entered. With positive duration, no nonempty hop can both enter and complete at `end`; `start = end` therefore leaves only the empty journey.

Declared vertex activity gates traversal appearances. The anchor must be valid: the forward source must be active exactly at the resolved origin, and the backward target either active there or leaving exactly there, since a spell's terminus is the last instant that vertex exists even though presence is half-open. An invalid anchor – which an explicit at outside the source's own spells produces – leaves every fixed-universe row, including the anchor row itself, unreachable. After a valid anchor, waiting may cross inactive periods. A zero-duration hop requires both endpoints at its time. A positive-duration interval hop requires both endpoints continuously on the closed traversal from entry through completion. A delayed point contact requires both endpoints at its trigger and the receiver again at completion, but creates no continuous edge or tail occupancy. Several activity-created timing domains of one canonical contact remain one path atom and cannot multiply `n_paths`.

For backward paths, `arrival_time` is the latest-departure supremum for a journey ending at the named target by the resolved end, and `latency` is `end` minus that value. A supremum at an interval's excluded terminus need not itself be an attainable departure. Such an endpoint is still reachable and still reports its route family: `n_hops`, `n_paths` and the steps of the routes that approach the supremum are those of the family, and `attained = FALSE` records that the instant itself is not realised.

With `sessions = "bounded"`, each endpoint is optimised across complete session-specific searches. A unique winner is named in `path_session`; ties leave it missing and are counted in `n_best_sessions`. No merged predecessor tree is exposed. The steps accessor retains a complete route from every tied best session, so each route stays inside one session. With `sessions = "separate"`, every session contributes a complete vertex block and resolves its own default origin. In the steps table, `time` is the optimal search label at that route vertex. For backward interval paths it can be an unattained supremum, as indicated by `attained = FALSE`.

With positive `traversal_time`, an interval hop entered at x arrives at $x + \text{traversal_time}$ and must fit within continuous activity for that pair; overlapping or touching interval spells form one

component. Completion exactly at the component terminus is allowed. A point event triggers at its timestamp and arrives after the same duration; it does not represent continued edge activity. The query end bounds completion, not only entry.

Optimal forward journeys are shortest foremost: final completion is minimised first (foremost) and hop count second (shortest). Backward journeys mirror it, maximising the departure time first and minimising hop count second. There is no criterion argument: this is the only criterion `paths()` offers, and it is recorded on the result as "foremost_then_shortest". A fastest journey, which minimises elapsed time rather than arrival time, is a different optimum and is not computed here. Journey identity is the ordered sequence of canonical oriented contacts. Duplicate points, overlapping or touching interval segmentation, weights, and waiting schedules do not multiply paths; genuinely recurrent contacts do. `n_paths` is exact through 2^{53} , after which a `dynet_path_overflow` condition is raised. The empty journey has one path and an unreachable endpoint has none.

Failures are classed. An unknown from raises `dynet_unknown_vertex`; a from that is not one name, a negative `traversal_time`, combining at with start or end, or a window that cannot hold a journey, raises `dynet_bad_input`; a window disjoint from explicit observation raises `dynet_outside_observation`; a count beyond 2^{53} raises `dynet_path_overflow`; and expanding more than a million routes through `as.data.frame(x, what = "steps")` raises `dynet_path_expansion_too_large`, which the compact `n_paths` column answers instead.

Value

An object of class "dynet_paths": a tidy data frame with one row per vertex and columns `node`, `reachable`, `arrival_time`, `attained` (whether that optimum itself is realised), `latency` (elapsed time between the origin and `arrival_time`, in either direction), `n_hops`, and the exact count `n_paths`. Bounded mode adds `path_session` and `n_best_sessions`; separate mode adds `session` and `origin`, one complete vertex block per session. Use `as.data.frame(x, what = "steps")` for every reconstructed optimal route: one row per vertex visited, with `endpoint`, `path_id` (endpoint-local, distinguishing tied atom sequences), `path_session`, `step`, `node`, `time` and `attained`, preceded by `session` in separate mode.

References

- Kempe, D., Kleinberg, J., & Kumar, A. (2002). Connectivity and inference problems for temporal networks. *Journal of Computer and System Sciences*, 64(4), 820-842.
- Bui-Xuan, B., Ferreira, A., & Jarry, A. (2003). Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, 14(2), 267-285.
- Holme, P., & Saramaki, J. (2012). Temporal networks. *Physics Reports*, 519(3), 97-125.
- Casteigts, A., Corsini, A., & Sarkar, W. (2024). Simple, strict, proper, happy: A study of reachability in temporal graphs. *Theoretical Computer Science*, 991, 114434.

Examples

```
dn <- dynet(school_contacts)
routes <- paths(dn, from = "Ana")
routes
summary(routes)
paths(dn, from = "Ana", start = 0, end = 10)
paths(dn, from = "Ana", direction = "backward")
```

pathways	<i>Most frequent time-respecting routes</i>
----------	---

Description

`pathways()` reports whole journeys rather than per-vertex summaries: one row per distinct route, ranked by how many optimal routes follow it. It answers "which pathways does this network actually use", where `path_trajectories()` answers "where do the routes diverge" and `paths()` answers "who is reachable".

Usage

```
pathways(dn, from = NULL, top = NULL, min_hops = 1L, ..., plot = FALSE)
```

Arguments

<code>dn</code>	A temporal network from <code>dynet()</code> .
<code>from</code>	Optional source vertex. A name gives the routes leaving that vertex, and several names give the routes leaving each of them. The default, <code>NULL</code> , pools every vertex, which is the network-wide question, and is the only case that adds a <code>from</code> column naming each route's source; a named source is already the first step of every route string.
<code>top</code>	Optional number of routes to keep, most frequent first. The default keeps all of them.
<code>min_hops</code>	Shortest route to report. Defaults to one, which drops the zero-hop route from a vertex to itself.
<code>...</code>	Passed to <code>paths()</code> , so <code>start</code> , <code>end</code> , <code>at</code> , <code>direction</code> , <code>sessions</code> and <code>traversal_time</code> all apply.
<code>plot</code>	Whether to draw the result as well as return it. Drawing is a side effect in the manner of <code>graphics::hist()</code> : the verb still returns its tidy table, invisibly when it has drawn, so <code>plot = TRUE</code> saves the wrapping <code>plot()</code> call without changing what comes back. Use <code>plot()</code> on the result when the figure needs arguments of its own.

Details

The result is already ordered and already carries the share of the total, so a caller never sorts or subsets it; `top` limits it in the call.

Routes are keyed on their **vertex sequence**. The trajectory tree keys a node on vertex *and* time, so one sequence realised through different contacts appears there as several branches; those are one pathway here and their counts are added. Under the foremost criterion this loses nothing: only earliest-arrival routes survive to be counted, so duplicates of a sequence necessarily share an arrival time, and a test asserts it.

Value

An object of class `dynet_pathways`, a data frame with one row per distinct route, most frequent first: `route`, the vertex sequence joined by arrows; `endpoint`, where it lands; `count`, how many optimal routes follow it; `share`, its fraction of every counted route, so the shares of a result limited by `top` do not sum to one; `n_hops`; and `arrival_time`, the earliest time the route lands. Pooling over every source adds `from` as the first column. Use `as.data.frame()` for a plain frame and `as.data.frame(x, what = "steps")` for the per-hop timing of the routes that were kept.

An unknown `from` raises `dynet_unknown_node`; a `top` or `min_hops` that is not one finite number in range raises `dynet_bad_input`; and a query that leaves no route of at least `min_hops` hops raises `dynet_empty_result`. Conditions raised by `paths()` on the arguments passed through `...` reach the caller unchanged.

See Also

`paths()` for reachability, `path_trajectories()` for the prefix tree those routes share.

Examples

```
dn <- dynet(school_contacts)
pathways(dn, from = "Ana")
pathways(dn, from = c("Ana", "Ben", "Kira"), top = 5)
```

path_centrality

Closeness and betweenness on time-respecting paths

Description

Centrality computed from the time-respecting paths that `paths()` finds, taken across the whole observation period (or the start-to-end window). A path may only continue along a tie that is available after it arrives, so these values cannot be inflated by ties that occur in the wrong order, as a flattened network is. The result is one value per vertex, not a series: for centrality that changes from window to window, use `centrality_series()`; for the number of vertices a vertex can reach, use `reachability()`.

Usage

```
path_centrality(
  dn,
  measure = "closeness",
  sessions = c("bounded", "collapse", "separate"),
  start = NULL,
  end = NULL,
  traversal_time = 0,
  plot = FALSE
)
```

Arguments

dn	A temporal network from <code>dynet()</code> .
measure	One or both of "closeness" (the default) and "betweenness". Any other name raises <code>dynet_unknown_measure</code> .
sessions	How to treat sessions: "bounded" (the default) keeps paths inside a session, "collapse" ignores sessions, "separate" reports each session on its own rows. "separate" on a network built without a session column raises <code>dynet_no_sessions</code> .
start, end	Inclusive path-traversal bounds. Default to the observed range. A network built from dates may be addressed with dates.
traversal_time	Nonnegative duration charged for every hop, in the network's time unit; 0 by default. A calendar network also accepts a scalar <code>difftime</code> .
plot	Whether to draw the result as well as return it. Drawing is a side effect in the manner of <code>graphics::hist()</code> : the verb still returns its tidy table, invisibly when it has drawn.

Details

Betweenness is the raw dependency sum over reachable forward ordered pairs. For each source-target pair, its unit dependency is divided equally over every canonical shortest-foremost journey, and an internal vertex receives the fraction of those journeys that contain it. Sources and targets receive no endpoint credit. This ordered-pair convention also applies to undirected contacts because temporal reach is generally asymmetric. The result is not normalised; its fixed range is $[\emptyset, (n - 1) * (n - 2)]$.

Closeness is inverse mean forward latency over reachable vertices: if R_s is the set of reachable vertices other than source s ,

$$C(s) = |R_s| / \sum_{z \in R_s} (a_z - o_s),$$

where a_z is the foremost arrival time and o_s is the source's resolved origin: the traversal window's lower bound, or – when vertex activity was declared – the source's first presence inside that window. Every reachable endpoint is included once, regardless of how many optimal paths reach it. A source with no reachable nonself endpoints has value zero. If all reachable endpoints have zero latency, the value is `Inf`; zero-latency endpoints remain in the numerator when mixed with positive latencies. The measure therefore has inverse-time units, is invariant to translating the time axis, and scales inversely when time is rescaled.

Both measures use `paths()` traversal semantics: nondecreasing times, unlimited waiting, half-open interval spells, and a separate exact timestamp rule for point events. Positive `traversal_time` requires an interval traversal to finish within continuous pair activity; a point event triggers at its timestamp and reaches its endpoint after that duration. `start` and `end` bound every measure. In separate-session output, a session outside a one-sided bound contributes zero rows.

Declared vertex activity gates the exact source anchor and every hop. Waiting after a valid anchor may cross inactivity; interval traversal requires both endpoints through completion, while a point trigger requires the receiver again after any traversal delay. Fixed node rows and full-network denominators are retained.

Value

A node-level `dynet_metric`: a tidy data frame with one row per vertex and measure, columns `node`, `measure` and `value`, preceded by `session` under `sessions = "separate"`. There is no time column. A single-measure result stores its mathematical choices as direct attributes; a two-measure result stores named records under `measure_metadata`.

Conditions

Errors: `dynet_unknown_measure` (a measure other than "closeness" or "betweenness"), `dynet_no_sessions` (`sessions = "separate"` without a session column), `dynet_outside_observation` (the requested range misses observed support; it also carries `dynet_bad_input`), and `dynet_bad_input` for every other broken contract – `dn` not a `dynet`, a malformed measure, an out-of-range start, end or `traversal_time`.

References

- Pan, R. K., & Saramaki, J. (2011). Path lengths, correlations, and centrality in temporal networks. *Physical Review E*, 84(1), 016105.
- Tang, J., Musolesi, M., Mascolo, C., Latora, V., & Nicosia, V. (2010). Analysing information flows and key mediators through temporal centrality metrics. *Proceedings of SNS '10*.
- Buss, S., Molter, H., Niedermeier, R., & Rymar, M. (2024). Algorithmic aspects of temporal betweenness. *Network Science*, 12(2), 160-188.
- Nicosia, V., Tang, J., Mascolo, C., Musolesi, M., Russo, G., & Latora, V. (2013). Graph metrics for temporal networks. In *Temporal Networks* (pp. 15-40). Springer.

See Also

[paths\(\)](#), [reachability\(\)](#), [centrality_series\(\)](#).

Examples

```
# Every ordered pair is searched, so the cost grows steeply with the
# vertex count; a subgraph keeps the example quick.
dn <- dynet(school_contacts)
few <- induce_subgraph(dn, nodes = c("Ana", "Ben", "Cara", "Dan", "Eve",
                                   "Finn", "Gita", "Hugo"))

path_centrality(few)
path_centrality(few, measure = c("closeness", "betweenness"),
               start = 0, end = 10)
```

Description

`paths()` uses an endpoint-local foremost-then-shortest criterion, so its routes need not form one predecessor tree. This function therefore returns the honest union of all expanded optimal route hops. Edge weight is the number of endpoint/path families using the hop; `first_time` and `last_time` retain its temporal range.

Usage

```
path_network(x)
```

Arguments

`x` A result from `paths()`.

Value

A static `dynet_path_network` cograph netobject, whose two tidy tables are reached with `as.data.frame(x, what = "edges")` and `as.data.frame(x, what = "nodes")`. The edge table has one row per hop used by at least one optimal route, with `from`, `to`, `weight` (how many endpoint/path families use the hop), `first_time` and `last_time` (the hop's temporal range) and `n_endpoints` (how many distinct endpoints it serves). The node table has one row per vertex the source actually reaches, the source included, with `name`, `arrival_time`, `latency`, `n_hops`, `n_paths` and `groups` (hop count as a grouping label for plotting). Unreachable vertices are absent, not present with NA. The network is always directed, because a route hop has an orientation even when the temporal network does not; hops of a backward path result still point the way time runs, from the sender towards the queried target, and its `arrival_time` is that vertex's latest-departure supremum, as in `paths()`.

A result that is not from `paths()` raises `dynet_bad_input`; a path result with no reachable vertex raises `dynet_empty_result`.

Examples

```
dn <- dynet(school_contacts)
routes <- paths(dn, from = "Ana")
union_network <- path_network(routes)
as.data.frame(union_network)
as.data.frame(union_network, what = "nodes")
```

path_trajectories

Optimal temporal routes as a counted trajectory tree

Description

Turns the optimal route family returned by `paths()` into a tidy prefix tree. Every row is one tree node: a route prefix reaching vertex at time, used by count optimal routes. A named vertex reached through a different temporal history is a separate row, so branches never create the misleading crossings of a path-union graph and two routes that differ only in when a hop fires stay separate.

Forward routes grow away from the queried source. Backward routes are reversed, so the queried target is the root and possible senders branch away from it.

Usage

```
path_trajectories(x, min_count = 1L, plot = FALSE)
```

Arguments

x	A result from <code>paths()</code> .
min_count	Keep only branches used by at least this many optimal routes. The default of 1 keeps the complete family; a higher value is the caller's explicit pruning.
plot	Whether to draw the result as well as return it. Drawing is a side effect in the manner of <code>graphics::hist()</code> : the verb still returns its tidy table, invisibly when it has drawn, so <code>plot = TRUE</code> saves the wrapping <code>plot()</code> call without changing what comes back. Use <code>plot()</code> on the result when the figure needs arguments of its own.

Value

A `dynet_path_trajectories` data frame with one row per tree node and columns `node` (the route prefix, written as `vertex@time` steps joined by arrows), `parent`, `depth`, `count`, `probability`, `vertex`, `time`, `session` and `branch`. `depth` is the hop number from the queried vertex, `probability` is the branching fraction of the parent's routes that continue along this branch and is missing at the root, which has no parent, and `branch` is the node's placement across the tree. The synthetic (start) root is dropped when it has a single child, which is the usual case; it is kept when it genuinely branches, as under `sessions = "separate"`, where it carries one subtree per session, has no vertex or time, a missing probability, and pushes every other node one hop deeper.

A result that is not from `paths()`, or a `min_count` that is not one positive whole number, raises `dynet_bad_input`; a path result with no route step, or a `min_count` no prefix reaches, raises `dynet_empty_result`.

See Also

`plot_path_trajectories()` to draw the tree, `path_network()` for the route union as a network.

Examples

```
dn <- dynet(school_contacts)
routes <- paths(dn, from = "Ana")
path_trajectories(routes)
```

plot.dynet

*Draw a temporal network***Description**

Nine views, each answering a different question.

"events" Every contact as a link drawn at the moment it fires, with actors on the vertical axis. A link leaves its source in the source's colour and arrives in the target's.

"timeline" Edge activity as an intensity heatmap, one row per pair. This is the view a static network cannot give you: it shows at a glance whether the network was busy throughout or concentrated in a few bursts.

"activity" Edges forming and dissolving over time.

"network" The network as a node-link diagram, drawn by `cograph::splot()`. With no `at`, the whole window is flattened into one picture – useful as a reference point, and as a reminder of how much it overstates, since every tie appears simultaneous. With `at`, only that time bin is drawn.

"snapshots" Small multiples, one `cograph::splot()` per time bin, laid out on shared coordinates so positions are comparable across panels.

"layers" The multilayer view: one network per time slice, drawn as a stack of layers in which each vertex appears once per slice and is joined to its own copy in the next by an identity arc of weight ω .

"heatmap" The matrix counterpart of "layers": each slice is a tilted heatmap plane rather than a node-link diagram.

"stack" The same slices projected as a node-link stack, each vertex keeping one colour through the whole stack so it can be followed between planes.

"proximity" Vertices placed on a vertical line at each time point according to how close they are in the network, and joined through time. Clusters appear as bands of lines travelling together.

All node-link rendering is `cograph`'s. A `dynet` object is a `cograph` `netobject`, so `cograph::splot(dn)` works directly and every one of its rendering arguments is available here through ...

Usage

```
## S3 method for class 'dynet'
plot(
  x,
  type = c("timeline", "events", "activity", "network", "snapshots", "proximity",
    "layers", "heatmap", "stack"),
  at = NULL,
  start = NULL,
  end = NULL,
  top = 40L,
  step = NULL,
  omega = 1,
```

```

bins = NULL,
link = c("hook", "arc", "chevron", "wave", "bracket"),
time = c("bin", "event", "clock"),
aggregate = TRUE,
nest = c("pair", "column"),
split = 0.8,
blend = FALSE,
weight = TRUE,
node_size = NULL,
node_shape = NULL,
node_fill = NULL,
node_border_color = NULL,
node_border_width = NULL,
node_alpha = NULL,
edge_color = NULL,
edge_alpha = NULL,
edge_width = NULL,
edge_width_range = NULL,
edge_style = NULL,
edge_start_style = NULL,
edge_start_length = NULL,
curvature = NULL,
curve_pivot = NULL,
label_size = NULL,
label_color = NULL,
label_fontface = NULL,
panels = 9L,
measure = "degree",
phases = NULL,
networks = TRUE,
events = TRUE,
labels = TRUE,
highlight = NULL,
slices = 120L,
window = NULL,
flow = 2L,
palette = "okabe",
default_dist = 2,
base_size = 12,
style = .dyn_style(),
...
)

```

Arguments

x	A temporal network from <code>dynet()</code> .
type	One of "timeline" (the default), "events", "activity", "network", "snapshots", "layers", "heatmap", "stack" or "proximity".

at	For "network", the time to draw. NULL draws the whole window flattened.
start, end	Window the plot to [start, end] before drawing. Either may be NULL, which keeps that side of the observed range. Every view is windowed, and an empty window is an error rather than an empty panel.
top	For the timeline, draw only the top busiest vertex pairs. Defaults to 40.
step	Width of one time bin, in the network's time unit. For "timeline" and "events" it is the bin the activity is counted in (1/24 on a network measured in days is hourly); for "layers", "heatmap" and "stack" it is the width of each slice, and at least two slices are needed, so too wide a step is an error rather than a single panel. NULL uses the construction interval.
omega	For "layers", the weight on the identity arcs carrying a vertex between adjacent slices, that is, the interlayer coupling. One non-negative number, 1 by default.
bins	Number of equal time bins for "timeline" and "events". NULL uses the network's own interval. step, a width, takes precedence when both are given.
link	Link glyph for "events": "hook" (the default), "arc", "chevron", "wave" or "bracket".
time	Time axis for "events". "bin", the default, groups onsets into equal windows and keeps duration honest, "event" gives one evenly spaced column per distinct onset, "clock" uses true positions.
aggregate	For "events", fold repeat firings of one pair inside one column into a single link, TRUE by default. Binning merges distinct onsets, and without this they stack as parallel bows carrying no extra reading.
nest	For "events", which links are fanned apart. "pair", the default, fans only links joining the same two rows in the same column; "column" fans every link sharing a column.
split	For "events", the share of each link that keeps its source colour before switching to its target's, so direction reads without arrowheads. One number between 0 and 1, 0.8 by default.
blend	For "events", fade between the two endpoint colours instead of switching at a boundary. FALSE by default.
weight	For "events", scale alpha and width by how often the pair occurs across the network, so one-off links recede and habitual ones stand out. TRUE by default.
node_size, node_shape, node_fill, node_border_color, node_border_width, node_alpha	Node aesthetics, named as in <code>cograph::splot()</code> . NULL uses the view's own default. They are honoured by the "network", "snapshots" and "events" views; the "layers", "heatmap", "stack" and "proximity" views take their renderer's own arguments through ...
edge_color, edge_alpha, edge_width, edge_width_range, edge_style	Link aesthetics, named as in <code>cograph::splot()</code> . An <code>edge_color</code> overrides the source-to-target colour run with one colour.
edge_start_style, edge_start_length	How the origin of each link is marked, named as in <code>cograph::splot()</code> . For "events" the defaults follow <code>cograph</code> 's TNA styling: the first 0.2 of every link, from its source, is "dotted"; "dashed" is also accepted and "solid" turns the mark off. <code>edge_start_length</code> is a share between 0 and 0.5.

curvature, curve_pivot	Bow geometry, as in <code>cograph::splot()</code> . <code>curvature</code> is the base bow as a fraction of the column gap and <code>0</code> draws straight links; <code>curve_pivot</code> slides where the bow peaks.
label_size, label_color, label_fontface	Axis label aesthetics, named as in <code>cograph::splot()</code> .
panels	For snapshots, the maximum number of panels to draw, 9 by default. Bins are sampled evenly across the window and the choice is reported.
measure	For the proximity view, the node-level measure that line thickness follows, "degree" by default. Any measure <code>centrality_series()</code> accepts at snapshot scope; the temporal-scope-only measures "reach" and "reach_count" are not available here, because the view redraws the measure over many short slices.
phases	For the proximity view, how many phases to split the window into for the network panels. <code>NULL</code> uses the network's sessions when it has them and three phases otherwise.
networks	Whether the proximity view draws a network panel per phase, <code>TRUE</code> by default.
events	Whether the proximity view marks the times edges formed, <code>TRUE</code> by default.
labels	Whether vertices are named, <code>TRUE</code> by default: beside each node in the "network", "snapshots", "layers" and "stack" views, and at the right-hand end of each line in the proximity view in place of a legend. The "timeline", "events", "activity" and "heatmap" views name their axes rather than their vertices and ignore it. <code>FALSE</code> is the readable choice for a network of more than a few dozen vertices.
highlight	Vertex names to draw in colour in the proximity view, with the rest in grey. <code>NULL</code> , the default, colours every vertex.
slices	How many times the proximity view measures the network across the window, 120 by default. Smoothness comes from measuring often, never from interpolation. <code>NULL</code> measures once per time bin, and anything else must be at least two.
window	Width of each proximity slice. <code>NULL</code> uses a sixth of the observation window, or the bin width if that is wider: scaling is only meaningful on a slice whose network is connected, and over one narrow bin most vertices are isolated.
flow	How many corner-cutting passes round each proximity line, 2 by default. Rounding only ever takes convex combinations of measurements, so it softens the joints without letting the curve overshoot one. <code>0</code> leaves them sharp.
palette	Colours for vertices and lines: "okabe" (the default, nine colour-blind safe colours, recycled), "extended" (hue varied with lightness, about twelve distinct), "many" (packed for separation, any number, not colour-blind safe), your own vector of colours, or a function of <code>n</code> returning <code>n</code> colours.
default_dist	Distance assumed between vertices with no path between them, in the proximity view. 2 by default.
base_size	Base font size for the "timeline", "events" and "activity" views, 12 by default. The "heatmap" view is also a ggplot but is sized by its own renderer.

style	Style constants for the proximity view's base-graphics panel: a list holding cex, grid, background, grid_color, axis_color, text_color and frame_color. The default is the package's own.
...	Passed to the renderer the chosen view uses: <code>cograph::splot()</code> for "network", "snapshots" and "proximity", <code>cograph::plot_mlna()</code> for "layers", <code>cograph::plot_ml_heatmap()</code> for "heatmap" and <code>cograph::plot_temporal()</code> for "stack". The remaining views take no further drawing arguments, and a name no view can read is an error rather than a silently ignored argument.

Details

The node-link views set a few of `cograph::splot()`'s defaults before handing over: they draw no edge labels and no edge-colour legend (`legend_edge_colors = FALSE`, against `cograph::splot()`'s own `TRUE`), colour edges a neutral grey, and size nodes and arrowheads from the vertex count. Naming any of those through ... overrides it.

Failures are classed conditions. `dynet_bad_input` covers every malformed argument, `dynet_unknown_plot_arg` a name in ... no view can read, `dynet_bad_palette` an unusable palette, and `dynet_empty_result` a window, an at or a step that leaves nothing to draw. The proximity view adds `dynet_unknown_measure` and `dynet_needs_directed`. With `cograph` absent, the node-link views raise `dynet_needs_cograph` and the "layers", "heatmap" and "stack" views `dynet_missing_package`.

Value

For "timeline", "events", "activity" and "heatmap", a ggplot object, which prints itself when the call is not assigned. For "network", "snapshots", "layers", "stack" and "proximity", the figure is drawn on the current device and the network is returned invisibly – `x` itself, or the windowed network when `start` or `end` was given.

References

Okabe, M., & Ito, K. (2008). Color universal design: how to make figures and presentations that are friendly to colorblind people.

Chaikin, G. M. (1974). An algorithm for high-speed curve generation. *Computer Graphics and Image Processing*, 3(4), 346-349.

Examples

```
dn <- dynet(school_contacts)
plot(dn)
plot(dn, type = "proximity")
plot(dn, type = "proximity", measure = "betweenness", phases = 4)
plot(dn, type = "network", node_fill = "#56B4E9")
```

plot.dynet_collapsed *Draw a collapsed temporal network*

Description

The union of a network's ties over a window, as a node-link diagram with Dynet's rendering defaults; any `cograph::splot()` argument overrides them.

Usage

```
## S3 method for class 'dynet_collapsed'
plot(x, palette = "okabe", ...)
```

Arguments

<code>x</code>	A result from <code>collapse_network()</code> .
<code>palette</code>	Palette specification, as in <code>plot.dynet()</code> .
<code>...</code>	Passed to <code>cograph::splot()</code> .

Value

`x`, invisibly.

Examples

```
dn <- dynet(school_contacts)
flat <- collapse_network(dn)
plot(flat, layout = "oval")
```

plot.dynet_metric *Plot a temporal measure*

Description

Draws the quantity against time. Node-level measures are drawn as one line per vertex; graph-level measures as one line per measure. Distinctions are carried by colour and line type together, never by colour alone.

Usage

```
## S3 method for class 'dynet_metric'
plot(
  x,
  type = c("line", "heatmap", "ridge"),
  highlight = NULL,
  top = NULL,
  palette = "okabe",
  base_size = 12,
  ...
)
```

Arguments

x	A dynet_metric.
type	"line" for trajectories over time, "heatmap" for a vertex-by-time tile plot, "ridge" for small multiples per measure. Ignored for a measure with no time axis.
highlight	Optional character vector naming the series to draw in colour, with everything else in grey: vertex names for a node-level measure, measure names for a graph-level one. For a mixing() result a group name selects every flow into or out of that group, so highlight = "Teacher" colours the teacher rows and columns of the mixing table. A name that matches nothing raises an error of class dynet_unknown_highlight. Ignored for a measure with no time axis.
top	How many rows to draw. For a measure taken over time, the top vertices with the largest mean value; NULL, the default, draws every vertex. For a measure with no time axis, the top rows with the largest absolute value, defaulting to 30, with a subtitle naming how many of how many are shown.
palette	Colours for the series: "okabe" (the default), "extended", "many", your own vector of colours, or a function of n.
base_size	Base font size.
...	Ignored.

Details

A measure with no time axis, such as reachability or [durations\(\)](#), has no trajectory to draw and is shown as a bar panel instead, one bar per vertex or pair and one facet per measure. type and highlight have nothing to act on there and are ignored.

Value

A ggplot object. Drawing happens when that object is printed, so the plot is the return value here rather than a side effect.

Examples

```
dn <- dynet(school_contacts)
degree <- centrality_series(dn, measure = "degree")
plot(degree, top = 5)
plot(degree, palette = "extended")
```

plot.dynet_paths

Plot time-respecting paths when a valid renderer exists

Description

A result from `paths()` records the optimality criterion it was found under, and such a result is drawn as a trajectory tree by `plot_path_trajectories()`. That is the right picture for an endpoint-local family: the routes need not share prefix-optimal subpaths, so a vertex reached under two different temporal histories appears twice rather than being forced into one predecessor tree.

Older serialised results carry no criterion. Those are drawn by the legacy predecessor-tree renderer, which only defines a tree when the sessions were collapsed; a bounded or separate-session result of that vintage raises `dynet_unsupported_plot` instead of implying a tree the criterion never promised.

Usage

```
## S3 method for class 'dynet_paths'
plot(x, palette = "okabe", ...)
```

Arguments

<code>x</code>	A <code>dynet_paths</code> from <code>paths()</code> .
<code>palette</code>	Palette specification, as in <code>plot.dynet()</code> . Vertices are coloured by how many hops they are from the source. Read only by the legacy tree renderer.
<code>...</code>	Passed to <code>plot_path_trajectories()</code> , or to <code>cograph::splot()</code> for a legacy result.

Value

A `ggplot` object for a result carrying criterion metadata, which is every result `paths()` returns. A legacy result without it is drawn on the current device by `cograph::splot()` and `x` is returned invisibly.

A legacy result raises `dynet_unsupported_plot` when its sessions were not collapsed, `dynet_empty_result` when the source reaches no other vertex, `dynet_bad_palette` for an unusable palette, and `dynet_needs_cograph` when `cograph` is not installed.

Examples

```
dn <- dynet(school_contacts)
journeys <- paths(dn, from = "Ana")
plot(journeys)
```

plot.dynet_pathways *Plot pathways on a time axis*

Description

One row per route, drawn along real time: a point at each vertex placed at the moment the route reaches it, joined by a segment. The horizontal gap between two points is the waiting time at that vertex, so the drawing is time-respecting in the way a bar of a route string is not – a route that waits is visibly slower than one that does not, and where routes arrive relative to each other is read off the axis.

Usage

```
## S3 method for class 'dynet_pathways'
plot(x, top = 12L, labels = TRUE, base_size = 12, ...)
```

Arguments

x	A dynet_pathways result.
top	Number of routes to draw, most frequent first. Defaults to twelve, which keeps the vertex labels legible.
labels	Whether to name the vertex at each step. TRUE by default; turn it off for a dense figure where the shape is the point.
base_size	Base font size, as in <code>ggplot2::theme_minimal()</code> . Defaults to twelve.
...	Ignored.

Details

Routes are ordered by frequency, most used at the top, with the count and share to the right of each. Fill marks the endpoint, which the row already names, so colour never carries a distinction alone.

Value

A ggplot object. A top, labels or base_size that is not one valid value raises dynet_bad_input.

Examples

```
dn <- dynet(school_contacts)
routes <- pathways(dn, from = "Ana")
plot(routes)
```

```
plot.dynet_path_network
```

Draw a path network

Description

The hops used by a set of optimal temporal paths, as a node-link diagram with Dynet's rendering defaults; any `cograph::splot()` argument overrides them.

Usage

```
## S3 method for class 'dynet_path_network'
plot(x, palette = "okabe", ...)
```

Arguments

<code>x</code>	A result from <code>path_network()</code> .
<code>palette</code>	Palette specification, as in <code>plot.dynet()</code> .
<code>...</code>	Passed to <code>cograph::splot()</code> .

Value

`x`, invisibly.

Examples

```
dn <- dynet(school_contacts)
routes <- paths(dn, from = "Ana")
route_network <- path_network(routes)
plot(route_network, layout = "oval")
```

```
plot.dynet_path_trajectories
```

Plot path trajectories

Description

A method wrapper so `plot()` works on the result directly. It draws exactly what `plot_path_trajectories()` draws; that function remains the place where the appearance arguments are documented.

Usage

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

Arguments

x A dynet_path_trajectories result.
 ... Passed to [plot_path_trajectories\(\)](#).

Value

A ggplot object.

Examples

```
dn <- dynet(school_contacts)
routes <- paths(dn, from = "Ana")
trajectories <- path_trajectories(routes)
plot(trajectories)
```

plot.dynet_pshifts	<i>Plot participation shift counts</i>
--------------------	--

Description

One horizontal bar per shift type, grouped and coloured by family. Families are distinguished by a direct axis grouping as well as by fill, so the figure does not rely on colour alone.

Usage

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

Arguments

x A dynet_pshifts result.
 ... Ignored.

Value

A ggplot object.

Examples

```
dn <- dynet(school_contacts)
shifts <- pshifts(dn)
plot(shifts)
```

plot.dynet_similarity *Draw time-bin similarity as a heatmap*

Description

Draw time-bin similarity as a heatmap

Usage

```
## S3 method for class 'dynet_similarity'
plot(x, base_size = 12, ...)
```

Arguments

x	A result from similarity() .
base_size	Base text size. Defaults to twelve.
...	Ignored.

Value

A ggplot object. Drawing happens when that object is printed, so the plot is the return value here rather than a side effect.

Examples

```
dn <- dynet(school_contacts)
bin_similarity <- similarity(dn, step = 5, window = 5)
plot(bin_similarity)
```

plot.dynet_snapshot *Plot how many ties each snapshot holds*

Description

Plot how many ties each snapshot holds

Usage

```
## S3 method for class 'dynet_snapshot'
plot(x, base_size = 12, palette = "okabe", ...)
```

Arguments

x	A dynet_snapshot from snapshots() .
base_size	Base font size.
palette	Palette specification, as in plot.dynet() .
...	Ignored.

Value

A ggplot object, faceted by session when the result carries one. A result in which no tie is active raises an error of class `dynet_empty_result` rather than drawing an empty panel.

Examples

```
dn <- dynet(school_contacts)
bins <- snapshots(dn)
plot(bins)
```

plot_path_trajectories

Draw optimal temporal paths as a trajectory tree

Description

Draws the optimal route family returned by `paths()` using the trajectory-tree grammar ported from the `transitiontrees` package: leaves stacked in depth-first order, parents centred on their children, and branches carried by a cosine smoothstep. Nodes follow that package's horizontal phylogram rather than its capsule style – a count-sized filled circle with its label set below it. Node size and branch width always show how many optimal routes use a branch. Every node also prints the value of the chosen measure beside its vertex name, so nothing is encoded by colour alone: the default frequency view is free to fill each node with its vertex's own colour, while the "time" and "predictability" views fill from a ramp with a colour bar.

Forward paths grow away from the queried source. Backward paths are flipped so the queried target is the root and possible senders branch away from it. A named vertex repeats whenever it is reached under a different temporal history.

Usage

```
plot_path_trajectories(
  x,
  measure = c("frequency", "time", "predictability"),
  orientation = c("horizontal", "vertical"),
  min_count = 1L,
  base_size = 11,
  palette = "okabe"
)
```

Arguments

<code>x</code>	A result from <code>paths()</code> or from <code>path_trajectories()</code> .
<code>measure</code>	What each node reports, and what fills it. "frequency", the default, is the number of optimal routes through the branch and fills by vertex, since node size already carries the count; "time" is the attained time at the node and "predictability" the branching fraction of the parent's routes that continue along the branch, both filled from a ramp.

orientation	"horizontal" grows the tree left to right with hop number on the x axis; "vertical" grows it top to bottom.
min_count	Draw only branches used by at least this many optimal routes. Defaults to 1, the complete family. Ignored when x is already a <code>path_trajectories()</code> result.
base_size	Base text size, as in <code>ggplot2::theme_minimal()</code> . Defaults to eleven.
palette	Palette for the vertex colours of the frequency view, as in <code>plot.dynet()</code> . Defaults to "okabe".

Value

A ggplot object. A tree with no branch to draw raises `dynet_empty_result`; an x that is neither a `paths()` nor a `path_trajectories()` result, or a `base_size` that is not one positive number, raises `dynet_bad_input`.

See Also

`path_trajectories()` for the tidy tree behind the plot.

Examples

```
dn <- dynet(school_contacts)
routes <- paths(dn, from = "Ana")
plot_path_trajectories(routes)
plot_path_trajectories(routes, measure = "time", orientation = "vertical")
```

print.dynet	<i>Print a temporal network</i>
-------------	---------------------------------

Description

Print a temporal network

Usage

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

Arguments

x	A temporal network from <code>dynet()</code> .
...	Ignored.

Value

x, invisibly.

Examples

```
dn <- dynet(school_contacts)
dn
```

```
print.dynet_animation Print an animation's bin table
```

Description

Print an animation's bin table

Usage

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

Arguments

x	A dynet_animation from animate() .
n	Largest number of bins to list, 12 by default.
...	Ignored.

Value

x, invisibly.

Examples

```
if (requireNamespace("gifski", quietly = TRUE) &&
    requireNamespace("cograph", quietly = TRUE)) {
  dn <- dynet(school_contacts)
  frames <- animate(dn, step = 6, window = 6, tween = 2)
  print(frames, n = 3)
}
```

```
print.dynet_collapsed
```

Print a collapsed temporal network

Description

Print a collapsed temporal network

Usage

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

Arguments

`x` A network returned by [collapse_network\(\)](#).
`...` Ignored.

Value

`x`, invisibly.

Examples

```
dn <- dynet(school_contacts)
collapsed <- collapse_network(dn)
collapsed
```

```
print.dynet_collapsed_list
```

Print session-specific collapsed networks

Description

Print session-specific collapsed networks

Usage

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

Arguments

`x` A `dynet_collapsed_list`.
`...` Ignored.

Value

x, invisibly.

Examples

```
dn <- dynet(data.frame(
  from = c("A", "A"), to = c("B", "B"), start = c(0, 0), end = c(2, 3),
  session = c("s1", "s2")
), session = "session")
collapse_network(dn, sessions = "separate")
```

print.dynet_metric	<i>Print a temporal measure</i>
--------------------	---------------------------------

Description

Print a temporal measure

Usage

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

Arguments

x	A dynet_metric.
n	Number of rows to show. Defaults to twelve.
...	Ignored.

Value

x, invisibly.

Examples

```
dn <- dynet(school_contacts)
degree <- centrality_series(dn, step = 4, window = 4)
degree
print(degree, n = 4)
```

print.dynet_paths	<i>Print time-respecting paths</i>
-------------------	------------------------------------

Description

Print time-respecting paths

Usage

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

Arguments

x	A dynet_paths from paths() .
n	Number of rows to show. Defaults to twelve.
...	Ignored.

Value

x, invisibly.

Examples

```
dn <- dynet(school_contacts)  
routes <- paths(dn, from = "Ana")  
routes  
print(routes, n = 4)
```

print.dynet_pathways	<i>Print ranked pathways</i>
----------------------	------------------------------

Description

Print ranked pathways

Usage

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

Arguments

x	A dynet_pathways result.
n	Number of routes to show. Defaults to twelve.
...	Ignored.

Value

x, invisibly. Called for the side effect of printing a header giving the number of distinct routes and of optimal routes counted, followed by the first n rows.

Examples

```
dn <- dynet(school_contacts)
routes <- pathways(dn, from = "Ana")
print(routes)
```

```
print.dynet_path_trajectories
      Print a temporal trajectory tree
```

Description

Print a temporal trajectory tree

Usage

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

Arguments

x	A result from <code>path_trajectories()</code> .
...	Passed to the data frame print method.

Value

x, invisibly. Called for the side effect of printing a header naming the direction, anchor, node count, depth and number of routes, followed by the tidy table.

Examples

```
dn <- dynet(school_contacts)
routes <- paths(dn, from = "Ana")
trajectories <- path_trajectories(routes)
print(trajectories)
```

```
print.dynet_projection
```

Print a time-projected network

Description

Print a time-projected network

Usage

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

Arguments

x	A projection returned by projection() .
...	Ignored.

Value

x, invisibly.

Examples

```
dn <- dynet(school_contacts)
projected <- projection(dn, step = 4, window = 4)
projected
```

```
print.dynet_pshifts
```

Print participation shift counts

Description

Print participation shift counts

Usage

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

Arguments

x	A dynet_pshifts result.
...	Ignored.

Value

x, invisibly.

Examples

```
dn <- dynet(school_contacts)
shifts <- pshifts(dn)
shifts
```

```
print.dynet_similarity
```

Print time-bin similarity

Description

Print time-bin similarity

Usage

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

Arguments

<i>x</i>	A result from similarity() .
<i>...</i>	Passed to the data frame print method.

Value

x, invisibly.

Examples

```
dn <- dynet(school_contacts)
resemblance <- similarity(dn, step = 4, window = 4)
resemblance
```

```
print.dynet_snapshot
```

Print snapshot edges

Description

Print snapshot edges

Usage

```
## S3 method for class 'dynet_snapshot'
print(x, n = 10L, ...)
```

Arguments

<code>x</code>	A <code>dynet_snapshot</code> from snapshots() .
<code>n</code>	Number of rows to show; ten by default.
<code>...</code>	Ignored.

Value

`x`, invisibly.

Examples

```
dn <- dynet(school_contacts)
bins <- snapshots(dn, at = 3)
bins
```

```
projection
```

Project a temporal network into directed vertex-time states

Description

`projection()` discretises a temporal network into snapshot slices and connects each vertex state to its realisation in the next slice. Within a slice it uses the same independently aggregated, endpoint-induced snapshot as [snapshots\(\)](#). Identity arcs always point forward and carry the coupling weight ω . The result is a tidy projection object rather than a bare matrix.

Usage

```
projection(
  dn,
  sessions = c("bounded", "collapse", "separate"),
  start = NULL,
  end = NULL,
  step = NULL,
  window = NULL,
  omega = 1
)
```

Arguments

<code>dn</code>	A temporal network from <code>dynet()</code> .
<code>sessions</code>	Session handling: "bounded" (the default), "collapse" or "separate". "collapse" erases labels. For a sessioned network, "bounded" and "separate" both preserve disjoint session-local projection blocks so identity arcs never cross a wall.
<code>start, end</code>	First and last slice times. Defaults to observed support.
<code>step</code>	Spacing between slice starts. NULL uses the construction interval.
<code>window</code>	Width represented by each slice. NULL uses step; zero samples an exact point; "all" represents the whole observed period as a single slice, closed on the right.
<code>omega</code>	Weight on the identity arcs that carry a vertex from one slice to the next, that is, the interlayer coupling of the time-expanded network. One, the default, keeps an identity arc as heavy as a unit contact; zero leaves the slices uncoupled. Must be a single finite non-negative number, or a <code>dynet_bad_input</code> error is raised.

Details

Every fixed-universe vertex receives one state in every emitted slice. `active` records whether the vertex was eligible in that slice. Identity arcs are retained through inactive slices because `Dynet` permits waiting through vertex inactivity; inactive states have no incident endpoint-induced within-slice edge. Consecutive observed slices are also linked across an observation gap, matching `Dynet`'s calendar-time waiting convention.

Directed source edges produce one within-slice arc. An undirected nonloop edge produces reciprocal arcs, while an undirected loop is emitted once. Parallel active spells are one within-slice pair whose weight is their summed weight and whose `n_spells` records their count. Identity arcs have `weight = omega` and `n_spells = 0`, and `meta$identity_weight` reports the same value.

Value

An object of class `dynet_projection`. Use `as.data.frame(x, what = "vertices")` for vertex states and `as.data.frame(x, what = "edges")` for directed projected arcs.

References

Bender-deMoll, S., & Moody, J. `timeProjectedNetwork()` in the `tsna` package, version 0.3.6.
 Butts, C. T., Leslie-Cook, A., Krivitsky, P. N., & Bender-deMoll, S. (2024). *networkDynamic: Dynamic Extensions for Network Objects*, version 0.11.5. doi:10.32614/CRAN.package.networkDynamic

Examples

```
dn <- dynet(data.frame(
  from = c("A", "B", "C"), to = c("B", "C", "A"),
  start = 0:2, end = 1:3
), observation_start = 0, observation_end = 3)
projected <- projection(dn, step = 1, window = 1)
as.data.frame(projected, what = "vertices")
as.data.frame(projected, what = "edges")
```

pshifts

Gibson participation shifts from raw temporal turns

Description

Gibson participation shifts from raw temporal turns

Usage

```
pshifts(
  dn,
  sessions = c("bounded", "collapse", "separate"),
  output = c("final", "cumulative"),
  start = NULL,
  end = NULL,
  group_events = c("simultaneous", "none"),
  plot = FALSE
)
```

Arguments

dn	A directed temporal network from <code>dynet()</code> . An undirected network raises an error of class <code>dynet_needs_directed</code> .
sessions	Session aggregation policy: "bounded" (the default) reads each session as its own turn sequence and pools the counts, "collapse" erases session labels and reads one calendar-ordered sequence, and "separate" reports each session on its own rows. "separate" needs a network built with a session column and raises <code>dynet_no_sessions</code> otherwise.
output	"final" (the default) for one row per shift class, "cumulative" for the running class vector at every turn.
start, end	Optional inclusive query limits; each query is a fresh sequence and never uses a predecessor outside the range. A network built from dates may be addressed with dates.
group_events	Infer one group-directed turn from simultaneous distinct recipients ("simultaneous", the default), or retain every dyadic row ("none"). Several turns at one instant are ordered by speaker, then group turn before dyadic turn, then target, each in the

network's vertex order; the classification of consecutive turns depends on that order, so under "none" a batch of simultaneous replies is read in vertex order.

plot Whether to draw the result as well as return it. Drawing is a side effect in the manner of `graphics::hist()`: the verb still returns its tidy table, invisibly when it has drawn, so `plot = TRUE` saves the wrapping `plot()` call without changing what comes back. Use `plot()` on the result when the figure needs arguments of its own.

Details

Only uncensored raw spell onsets inside the observed query and observation components are turns; duration, weights, fragments and terminus censoring are ignored. Consecutive turns are classified using Gibson's fixed thirteen labels. Session and component walls, loops, ties, duplicate multiplicity, and simultaneous-recipient group inference are retained in metadata. `output = "final"` emits one typed row per class; `output = "cumulative"` emits the running class vector for each turn.

Value

A `dynet_pshifts` data frame whose shape follows `output`. "final" gives one row per shift class – thirteen rows, always all thirteen even when a class never occurred – with columns `shift` (the Gibson label), `family` (the label's group) and `count`. "cumulative" gives one row per turn and class, that is thirteen rows per classified turn, with sequence and event locating the turn in its sequence, time, speaker, target and group describing the turn, and `shift`, `family` and `count` carrying the running total of that class up to and including the turn. Either shape gains a leading session column under `sessions = "separate"`, which reports each session on its own rows; "bounded" and "collapse" carry no session column. Print it, `summary()` it, `plot()` it, or take the plain frame with `as.data.frame()`.

Conditions

Errors: `dynet_needs_directed` (an undirected network; the class vector is `c("dynet_needs_directed", "dynet_bad_input")`), `dynet_no_sessions` (`sessions = "separate"` without a session column), `dynet_outside_observation` (the requested range misses observed support; it also carries `dynet_bad_input`), and `dynet_bad_input` for every other broken contract – `dn` not a `dynet`, and a `start` or `end` that is not a single finite time. An unmatched `sessions`, `output` or `group_events` is rejected by `match.arg()` and is a plain error, not a classed one.

References

Gibson, D. R. (2003). Participation shifts: Order and differentiation in group conversation. *Social Forces*, 81, 1335–1380. doi:10.1353/sof.2003.0055

Examples

```
dn <- dynet(data.frame(
  from = c("A", "B"), to = c("B", "A"), start = c(1, 2), end = c(1, 2)
))
pshifts(dn)
```

reachability

*Reachability of every vertex***Description**

The number or share of other vertices each vertex can reach along time-respecting paths, and the number or share that can reach it. Reachability is the temporal replacement for component membership: in a static network two vertices in the same component reach each other by definition, whereas in a temporal network reach depends on whether the timing lines up.

Usage

```
reachability(
  dn,
  direction = c("both", "forward", "backward"),
  at = NULL,
  sessions = c("bounded", "collapse", "separate"),
  start = NULL,
  end = NULL,
  traversal_time = 0,
  measure = "reach",
  plot = FALSE
)
```

Arguments

<code>dn</code>	A temporal network from dynet() .
<code>direction</code>	"both" (the default, reporting each vertex's forward and backward reach side by side), "forward" or "backward".
<code>at</code>	Forward source-availability time or backward arrival deadline, defaulting to the beginning or end of each observed period respectively. Unlike in paths() it sets only the traversal window, because every vertex is then anchored at its own presence inside that window: a vertex with declared spells starts at the first instant it is present there, or at the last instant searching backward, and one with no declared spells starts at the window bound. Date and date-time values use the network's time scale. It cannot be combined with <code>start</code> or <code>end</code> .
<code>sessions</code>	How to treat sessions, as in path_centrality() .
<code>start, end</code>	Inclusive lower and upper traversal-time bounds. Interval spells remain terminus-exclusive.
<code>traversal_time</code>	Nonnegative duration charged for every hop, in the network's time unit. A calendar network also accepts a scalar <code>difftime</code> .
<code>measure</code>	One or both of "reach", the proportion of other vertices, and "reach_count", their number. The source vertex is excluded from both. Defaults to "reach".

`plot` Whether to draw the result as well as return it. Drawing is a side effect in the manner of `graphics::hist()`: the verb still returns its tidy table, invisibly when it has drawn, so `plot = TRUE` saves the wrapping `plot()` call without changing what comes back. Use `plot()` on the result when the figure needs arguments of its own.

Details

Reachability uses `paths()` traversal semantics: nondecreasing times, unlimited waiting, half-open interval spells, and a separate exact timestamp rule for point events. Positive `traversal_time` requires interval occupancy to finish within continuous pair activity and delays a point-trigger arrival. Declared vertex activity additionally requires active hop endpoints and a valid anchor, and every vertex is anchored at its own presence: each search starts at that vertex's first instant inside the window, or its last instant searching backward, rather than at the window bound. A vertex never present inside the window reaches nothing, which is reported as zero rather than as a missing row. Waiting after a valid anchor may cross vertex inactivity; interval traversal requires both endpoints continuously through completion, while a delayed point requires the receiver again at completion. For backward reachability, the resolved end is a common deadline and latest-departure suprema determine whether a vertex can reach the target. The canonical start and end bounds apply one closed traversal-time window to both forward and backward queries.

The source is excluded: a count is the number of distinct other vertices in the reachable set, not the number of journeys. A proportion divides that count by the full network size minus one. It is defined as zero for a singleton network. In separate-session output the same full-network denominator is retained in every session block.

In separate-session output, a session entirely outside a one-sided bound contributes zero-reach rows rather than aborting the complete result. Its missing implicit bound is clamped to the supplied bound, producing the empty journey at that boundary and no eligible hop.

Failures are classed. An unrecognised measure raises `dynet_unknown_measure`; a malformed measure, a negative `traversal_time`, at combined with start or end, or a window that cannot hold a journey raises `dynet_bad_input`; and a window disjoint from explicit observation raises `dynet_outside_observation`.

Value

A `dynet_metric` at node level: a tidy data frame with one row per vertex per requested measure, columns `node`, `measure` and `value`, preceded by `session` under `sessions = "separate"`. Proportion measures are named `forward_reach` and `backward_reach`; counts are named `forward_reach_count` and `backward_reach_count`. `as.data.frame()` returns the plain frame.

References

- Holme, P. (2005). Network reachability of real-world contact sequences. *Physical Review E*, 71(4), 046119.
- Holme, P., & Saramaki, J. (2012). Temporal networks. *Physics Reports*, 519(3), 97-125.

Examples

```
# Reachability searches every ordered pair, so the example uses a small
```

```
# inline network to stay fast. The verb takes any `dynet`.
dn <- dynet(data.frame(
  from = c("A", "B", "C", "A"),
  to   = c("B", "C", "D", "D"),
  start = c(0, 1, 2, 3),
  end   = c(1, 2, 3, 4)
))
reachability(dn)
reachability(dn, direction = "forward")
reachability(dn, start = 0, end = 2)
```

remove_arcs

Remove directed temporal arcs

Description

The same operation as [remove_ties\(\)](#), with one extra guarantee: the network must already be directed, so a from/to pair names one arc and not both orientations. Removing an arc from an undirected network raises a condition of class `dynet_needs_directed`.

Usage

```
remove_arcs(
  dn,
  ties = NULL,
  from = NULL,
  to = NULL,
  start = NULL,
  end = NULL,
  session = NULL
)
```

Arguments

dn	A temporal network from dynet() .
ties	Which ties to remove: a condition on the spell table, evaluated the way subset() evaluates <code>one - duration > 2, course == "g1"</code> – over the columns as <code>data.frame(dn)</code> returns, tie attributes included; or integer positions or a logical mask over that table.
from, to, start, end, session	Optional selectors combined by conjunction. <code>start</code> and <code>end</code> match a spell's own boundary, compared with the package's magnitude-relative time tolerance rather than exactly, so a selector written <code>0.3</code> still matches a spell that accumulated as <code>0.1 + 0.1 + 0.1</code> . When <code>ties</code> is supplied, these selectors must be omitted. On undirected networks <code>from</code> and <code>to</code> must be supplied together and their order is ignored.

Value

A new directed dynet object without the matched spells, with the same structure as the input.

See Also

`remove_ties()`, which does not require a directed network.

Examples

```
dn <- dynet(data.frame(
  from = c("A", "B"), to = c("B", "C"),
  start = c(0, 1), end = c(1, 2)
))
remove_arcs(dn, ties = 1)
```

remove_nodes	<i>Remove nodes from a temporal network</i>
--------------	---

Description

Remove nodes from a temporal network

Usage

```
remove_nodes(dn, nodes, cascade = FALSE)
```

Arguments

dn	A temporal network from <code>dynet()</code> .
nodes	Character node names.
cascade	Whether to remove every incident temporal tie and vertex activity spell. The safe default, FALSE, rejects nodes that are not isolates with a condition of class <code>dynet_node_not_isolate</code> .

Value

A new internally consistent dynet object, of the same class and structure as the input, without the named vertices and – under `cascade = TRUE` – without their ties and vertex-activity spells. At least one temporal tie must remain.

Examples

```
dn <- dynet(data.frame(from = "A", to = "B", start = 0, end = 1))
dn <- add_nodes(dn, "C")
remove_nodes(dn, "C")
```

remove_ties	<i>Remove temporal ties</i>
-------------	-----------------------------

Description

Remove temporal ties

Usage

```
remove_ties(
  dn,
  ties = NULL,
  from = NULL,
  to = NULL,
  start = NULL,
  end = NULL,
  session = NULL
)
```

Arguments

dn	A temporal network from <code>dynet()</code> .
ties	Which ties to remove: a condition on the spell table, evaluated the way <code>subset()</code> evaluates <code>one - duration > 2</code> , <code>course == "g1"</code> – over the columns as <code>data.frame(dn)</code> returns, tie attributes included; or integer positions or a logical mask over that table.
from, to, start, end, session	Optional selectors combined by conjunction. <code>start</code> and <code>end</code> match a spell's own boundary, compared with the package's magnitude-relative time tolerance rather than exactly, so a selector written <code>0.3</code> still matches a spell that accumulated as <code>0.1 + 0.1 + 0.1</code> . When <code>ties</code> is supplied, these selectors must be omitted. On undirected networks <code>from</code> and <code>to</code> must be supplied together and their order is ignored.

Value

A new internally consistent `dynet` object, of the same class and structure as the input, without the matched spells. At least one temporal tie must remain. A request that matches nothing raises a condition of class `dynet_tie_not_found`.

Examples

```
dn <- dynet(data.frame(
  from = c("A", "B"), to = c("B", "C"),
  start = c(0, 1), end = c(1, 2)
))
remove_ties(dn, ties = 1)
```

remove_vertex_spells	<i>Remove declared vertex-activity components</i>
----------------------	---

Description

Remove declared vertex-activity components

Usage

```
remove_vertex_spells(dn, spells)
```

Arguments

dn	A temporal network.
spells	Integer positions or a logical mask over <code>as.data.frame(dn, what = "vertex_spells")</code> . A logical mask must have one element per declared component and no NA.

Value

A new dynet object, class `c("dynet", "netobject", "cograph_network")`, with the selected activity components dropped and the rest canonicalised again, so the remaining spell identifiers renumber. A node with no remaining declaration becomes implicitly always active over observation support. Raises `dynet_bad_input` when `spells` is not a valid selection.

Examples

```
dn <- dynet(school_contacts)
present <- data.frame(node = c("Ana", "Ben"), start = 0, end = 10)
enrolled <- set_vertex_spells(dn, present)
trimmed <- remove_vertex_spells(enrolled, spells = 1)
as.data.frame(trimmed, what = "vertex_spells")
```

rename_nodes	<i>Rename nodes everywhere in a temporal network</i>
--------------	--

Description

Rename nodes everywhere in a temporal network

Usage

```
rename_nodes(dn, mapping)
```

Arguments

dn	A temporal network.
mapping	A named character vector whose names are old node names and values are replacements, a two-column data frame named old and new, or the name of one vertex attribute (given through <code>dynet(nodes =)</code>) whose values become the node names. The attribute must be complete and unique.

Value

A new `dynet` object, class `c("dynet", "netobject", "cograph_network")`, with edge endpoints, node attributes, vertex activity, cograph labels, and groups renamed together. Raises `dynet_unknown_node` when an old name is not a vertex, `dynet_duplicate_node` when a replacement collides with a name that is being kept, `dynet_unknown_attribute` when mapping names a column that is not a vertex attribute, and `dynet_bad_input` otherwise.

Examples

```
dn <- dynet(school_contacts)
renamed <- rename_nodes(dn, c(Ana = "Anna", Ben = "Benjamin"))
renamed
```

rename_sessions	<i>Rename session walls</i>
-----------------	-----------------------------

Description

Rename session walls

Usage

```
rename_sessions(dn, mapping)
```

Arguments

dn	A sessioned temporal network.
mapping	A named character vector from old to new labels, or an old/new data frame.

Value

A new `dynet` object, class `c("dynet", "netobject", "cograph_network")`, with edge and vertex session labels renamed together and the session scheme in its metadata updated. Labels absent from mapping are left alone. Raises `dynet_unknown_session` when an old label is not a session, and `dynet_bad_input` when the network has no session scheme, when mapping is malformed, or when the renaming would produce duplicate labels.

Examples

```
dn <- dynet(school_contacts)
weeks <- with(school_contacts, ifelse(start < 7, "week_1", "later"))
labelled <- set_tie_sessions(dn, session = weeks)
renamed <- rename_sessions(labelled, c(week_1 = "opening"))
renamed
```

school_contacts

Student contacts recorded as intervals

Description

Face-to-face contacts among fourteen students over three weeks. Each row is one contact with an explicit start and end, which makes this an interval log: duration carries information, and two students who met once at length are distinguishable from two who met briefly many times. The students fall into three loosely-connected friendship clusters and overall activity rises through the second week before falling away.

Usage

```
school_contacts
```

Format

A data.frame with 240 rows and 4 columns:

from Character. The student initiating the contact.

to Character. The other student.

start Numeric. Day the contact began, counted from day zero.

end Numeric. Day the contact ended.

The fourteen students are Ana, Ben, Cara, Dan, Eve, Finn, Gita, Hugo, Iris, Jonas, Kira, Leo, Mira and Nils; times run from day 0 to day 21.52.

Source

Simulated, not observed. Generated deterministically under a fixed seed by `data-raw/make-data.R`.

Examples

```
dynet(school_contacts)
```

seminar_attendance	<i>Seminar attendance</i>
--------------------	---------------------------

Description

Which students attended which weekly seminar over one term. This is two-mode data: students are not linked to each other directly, only to the seminars they turned up to. `dynet()` projects it, connecting every pair of students who shared a room in the week they shared it.

Usage

seminar_attendance

Format

A data.frame with 104 rows and 3 columns:

student Character. Student identifier, s01 to s24.

seminar Character. Which weekly seminar, week_01 to week_12.

date Date. When the seminar was held, 2024-09-03 to 2024-11-19.

Source

Simulated, not observed. Generated deterministically under a fixed seed by `data-raw/make-data.R`.

Examples

```
dynet(seminar_attendance, actor = "student", group = "seminar")
```

set_observations	<i>Replace observation support</i>
------------------	------------------------------------

Description

Replace observation support

Usage

```
set_observations(dn, data = NULL, start = NULL, end = NULL)
```

Arguments

dn	A temporal network.
data	Optional data frame with start and end observation components. Overlapping and adjacent positive components are merged. Default NULL.
start, end	Optional scalar continuous bounds used instead of data, each defaulting to NULL. Supply exactly one of data or the start/end pair; supplying both, or neither, is an error.

Value

A new dynet object, class `c("dynet", "netobject", "cograph_network")`. Raw edge and vertex spells are unchanged – `as.data.frame(x)` still returns the originals – and only the non-destructive measurement view is replaced, so every verb now clips exposure and path horizons to this support. Read the components back with `as.data.frame(x, what = "observations")`, one row per component with observation, start, end, duration and instant. Raises `dynet_bad_input` when neither or both of data and the bounds are given.

Examples

```
dn <- dynet(school_contacts)
first_week <- set_observations(dn, start = 0, end = 7)
as.data.frame(first_week, what = "observations")
```

set_tie_sessions	<i>Assign or remove tie sessions</i>
------------------	--------------------------------------

Description

Assign or remove tie sessions

Usage

```
set_tie_sessions(dn, session = NULL, breaks = NULL, labels = NULL)
```

Arguments

dn	A temporal network.
session	A complete, nonempty character vector of length one or the raw tie count; a length-one value labels every spell. Default NULL, which, when breaks is also NULL, removes all tie-session walls and erases session labels on vertex activity. A vector of the full length is matched positionally against the spell table, not against the data frame the network was built from. <code>dynet()</code> sorts spells by start, end, from and to, so the two orders coincide only when the input was already in that order. To cut sessions by time, use breaks instead.
breaks	Optional increasing numeric vector of cut points on the network's time axis. A spell belongs to the session of the interval its start falls in: before the first break, between two breaks, or from the last break on, so k breaks give k + 1 sessions. Mutually exclusive with session.
labels	Optional character vector naming the k + 1 sessions that breaks defines, in time order. The default is session_1, session_2 and so on.

Value

A new dynet object, class `c("dynet", "netobject", "cograph_network")`, with a session column on the spell table and the session scheme recorded in its metadata, or with both removed when neither session nor breaks is given. Raises `dynet_bad_input` when session has neither length one nor the raw tie count, or carries NA or blank labels; when session and breaks are both given; when breaks is not increasing and finite; or when labels does not have one more element than breaks.

Examples

```
dn <- dynet(school_contacts)
weeks <- set_tie_sessions(dn, breaks = c(7, 14),
                          labels = c("week_1", "week_2", "week_3"))
weeks
```

set_vertex_spells	<i>Replace declared vertex activity</i>
-------------------	---

Description

Replace declared vertex activity

Usage

```
set_vertex_spells(dn, data = NULL)
```

Arguments

dn	A temporal network.
data	A vertex-spell data frame with node, start, and end, plus optional session, onset_censored, and terminus_censored; or the string "ties", which declares each vertex present from the start of its first tie spell to the end of its last, so that a vertex is absent before it has had a tie and after it has had its last. Default NULL, which clears explicit activity, making every retained node implicitly active over observation support.

Value

A new dynet object, class `c("dynet", "netobject", "cograph_network")`, whose declared vertex activity is exactly data and whose edge spells, node attributes and metadata are those of dn. Overlapping or adjacent spells are canonicalised exactly as in `dynet()`, so canonical spell identifiers may change. Read the result back with `as.data.frame(x, what = "vertex_spells")`. Raises `dynet_unknown_node` when data names a vertex the network does not have, and `dynet_bad_input` for a string other than "ties".

Examples

```
dn <- dynet(school_contacts)
present <- data.frame(node = c("Ana", "Ben"), start = 0, end = 10)
enrolled <- set_vertex_spells(dn, present)
as.data.frame(enrolled, what = "vertex_spells")

# Present from the first contact to the last, vertex by vertex.
spanned <- set_vertex_spells(dn, "ties")
as.data.frame(spanned, what = "vertex_spells")
```

similarity

Similarity between the networks at each pair of time points

Description

Compares the edge set at every time bin with the edge set at every other, giving the pairwise similarity matrix as a tidy frame. This answers how much the network at one moment resembles the network at another, which no single-bin measure reports and which the formation and dissolution quantities in [events\(\)](#) only address between neighbouring bins.

Coefficients are computed by `cograph::layer_similarity()`.

Usage

```
similarity(
  dn,
  method = c("jaccard", "overlap", "hamming", "cosine", "pearson"),
  sessions = c("bounded", "collapse", "separate"),
  start = NULL,
  end = NULL,
  step = NULL,
  window = NULL,
  plot = FALSE
)
```

Arguments

dn	A temporal network from dynet() .
method	One of "jaccard" (the default), "overlap", "hamming", "cosine" or "pearson".
sessions	How to treat sessions when the layers are built: "bounded" (the default) and "collapse" differ in whether a session wall gates a tie into its bin. Unlike centrality_series() , "separate" adds no session column here: layers are keyed on time alone, so two session-local bins sharing a time are compared as one layer.
start, end	First and last time to measure. Default to the observed range.
step	How often to measure. Defaults to the interval the network was built with.

window	How much time each measurement covers. Defaults to step. window = "all" measures the whole observed period as a single bin and so leaves nothing to compare; it raises <code>dynet_empty_result</code> , as does any other grid that yields fewer than two bins.
plot	Whether to draw the result as well as return it. Drawing is a side effect in the manner of <code>graphics::hist()</code> : the verb still returns its tidy table, invisibly when it has drawn, so plot = TRUE saves the wrapping <code>plot()</code> call without changing what comes back. Use <code>plot()</code> on the result when the figure needs arguments of its own.

Value

A `dynet_similarity` data frame with one row per ordered pair of time bins and columns time, other, measure and value. The diagonal is included and is one for every coefficient except "hamming", where identical layers differ in nothing and score zero. "pearson" reaches one only to floating-point accuracy, so compare it with a tolerance rather than with `==`. The frame is returned invisibly when plot = TRUE has drawn the figure.

The coefficients come from `cograph`, which is a hard dependency of `Dynet`; a namespace that cannot be loaded raises `dynet_needs_cograph`.

See Also

`snapshots()` for the networks being compared, `events()` for formation and dissolution between neighbouring bins.

Examples

```
dn <- dynet(school_contacts)
similarity(dn)
similarity(dn, method = "cosine")
```

snapshots

The network sliced into snapshots

Description

The edges alive in each time bin, as one tidy table. Useful for exporting a slice, for feeding a layout routine, or for checking by eye what the metric verbs are seeing.

Usage

```
snapshots(
  dn,
  at = NULL,
  sessions = c("bounded", "collapse", "separate"),
  sample = NULL,
  start = NULL,
```

```

    end = NULL,
    step = NULL,
    window = NULL,
    plot = FALSE
  )

```

Arguments

<code>dn</code>	A temporal network from <code>dynet()</code> .
<code>at</code>	Optional single time, narrowing the result to the bins that cover it. A network built from dates may be addressed with a date. With the default disjoint tiling that is one bin; with an overlapping window every bin containing the time is returned. A time outside every bin falls back to the nearest bin rather than failing, but that bin may itself hold no active tie, in which case the result is a zero-row frame with the documented columns.
<code>sessions</code>	How to treat sessions, as in <code>centrality_series()</code> : "bounded" (the default), "collapse" or "separate". "separate" needs a network built with a session column and raises <code>dynet_no_sessions</code> otherwise.
<code>sample</code>	Deprecated. "instant" is equivalent to <code>window = 0</code> ; "window" uses the current positive/default window.
<code>start, end</code>	First and last time at which to measure. Default to the observed range. A network built from dates may be addressed with dates.
<code>step</code>	How often to measure. Defaults to the interval the network was built with.
<code>window</code>	How much time each measurement covers. Defaults to <code>step</code> , which tiles the period into disjoint bins. A larger value slides an overlapping window; <code>0</code> samples the network at each point in time. "all" measures the whole observed period as one window, closed on the right so an event at the final instant is inside it; it cannot be combined with <code>step</code> , and under <code>sessions = "separate"</code> or discontinuous observation it gives one window per session or observed component.
<code>plot</code>	Whether to draw the result as well as return it. Drawing is a side effect in the manner of <code>graphics::hist()</code> : the verb still returns its tidy table, invisibly when it has drawn, so <code>plot = TRUE</code> saves the wrapping <code>plot()</code> call without changing what comes back. Use <code>plot()</code> on the result when the figure needs arguments of its own.

Value

A `dynet_snapshot` data frame with one row per active edge per bin: `session` (when the network has sessions), `observation` (when observation is discontinuous, naming the observed component the bin falls in), `time`, `from`, `to`, `weight` and `n_spells`. `print()` shows a header and the first rows, `summary()` collapses to one row per bin, `plot()` draws how many ties each bin holds, and `as.data.frame()` returns the plain table. A pair joined by more than one spell in the same bin is one edge, with `n_spells` recording how many spells were collapsed – so the edge counts here agree with those from `metrics()`. `weight` is the sum of those spells' full weights: a spell counts its whole weight in every bin it touches, as `networkDynamic::network.collapse()` does. Snapshot "strength" in `centrality_series()` instead splits a spell's weight by the share of its duration inside the bin. Eligible isolates have no synthetic edge row; use `centrality_series()` or `metrics()` when the eligible population itself is required.

Conditions

Errors: `dynet_no_sessions` (`sessions = "separate"` without a session column), `dynet_outside_observation` (the requested range misses observed support; it also carries `dynet_bad_input`), and `dynet_bad_input` for every other broken contract – `dn` not a `dynet`, an `at`, `start` or `end` that is not a single finite time, and an out-of-range step or window.

Warning: `dynet_deprecated` for the retired `sample` argument.

Examples

```
dn <- dynet(school_contacts)
snapshots(dn, at = 3)
```

summary.dynet	<i>Describe a temporal network</i>
---------------	------------------------------------

Description

A tidy description of the whole network, one row per property. Two densities are reported and they answer different questions. Snapshot density is the mean over time bins of realised against possible edges. Temporal density is the proportion of all possible relational exposure occupied during the observation window. Overlapping and duplicate spells for the same ordered pair, or dyad in an undirected network, are unioned before their duration is counted.

Usage

```
## S3 method for class 'dynet'
summary(object, temporal_density = FALSE, ...)
```

Arguments

<code>object</code>	A temporal network from <code>dynet()</code> .
<code>temporal_density</code>	Whether to compute the temporal-density row. FALSE, the default, reports "not computed" for it. The quantity integrates exact occupancy over every eligible ordered pair, so its cost grows with the square of the vertex count: on a 442-vertex forum network it takes about 32 seconds, while every other row in the table is immediate. Pass TRUE when the number is wanted.
<code>...</code>	Ignored.

Details

Let $Y_q(t)$ indicate that both endpoints of relational opportunity q are eligible at positive observed time t , and let $E_q(t)$ indicate binary union edge activity. Temporal density is

$$\rho = \frac{\sum_q \int Y_q(t) E_q(t) dt}{\sum_q \int Y_q(t) dt}.$$

Directed opportunities are ordered; undirected opportunities are unordered. The integrals are evaluated exactly over observation, vertex, and edge change points. Self-loops, weights, session labels, duplicate spells, genuine points, and observation gaps do not add exposure. A network with no positive time containing two coeligible distinct vertices has undefined temporal density and reports NA.

This is an occupancy definition. Unlike summing spell durations, it remains in $[0, 1]$ when the same relation has overlapping or duplicated spells.

Value

A data.frame with columns property and value, one row per property.

References

Bender-deMoll, S., & Morris, M. (2025). *tsna: Tools for Temporal Social Network Analysis*. R package version 0.3.6.

Holme, P., & Saramaki, J. (2012). Temporal networks. *Physics Reports*, 519(3), 97-125.

Latapy, M., Viard, T., & Magnien, C. (2018). Stream graphs and link streams for the modeling of interactions over time. *Social Network Analysis and Mining*, 8, 61.

Examples

```
dn <- dynet(school_contacts)
summary(dn)

# The temporal density is opt-in, since it is quadratic in the vertex count.
summary(dn, temporal_density = TRUE)
```

```
summary.dynet_animation
```

Summarise an animation

Description

Summarise an animation

Usage

```
## S3 method for class 'dynet_animation'
summary(object, ...)
```

Arguments

object	A dynet_animation from <code>animate()</code> .
...	Ignored.

Value

A one-row plain data.frame with bins, frames, fps, seconds, tween, ease, layout, format, measure (what node size follows, NA when it is constant), first_time, last_time, min_ties, max_ties, turnover (over the bins after the first that hold a tie, the median share of a bin's ties that were not active in the bin before: near 0 the film flows, near 1 every bin is a new picture; NA with a single bin) and file.

Examples

```
if (requireNamespace("gifski", quietly = TRUE) &&
    requireNamespace("cograph", quietly = TRUE)) {
  dn <- dynet(school_contacts)
  frames <- animate(dn, step = 6, window = 6, tween = 2)
  summary(frames)
}
```

summary.dynet_collapsed_list

Summarise session-specific collapsed networks

Description

Summarise session-specific collapsed networks

Usage

```
## S3 method for class 'dynet_collapsed_list'
summary(object, ...)
```

Arguments

```
object      A dynet_collapsed_list.
...         Ignored.
```

Value

A plain data.frame, one row per session: session, the number of collapsed pairs, the nodes those pairs span, and the summed union_duration and total_duration.

Examples

```
dn <- dynet(data.frame(
  from = c("A", "A"), to = c("B", "B"), start = c(0, 0), end = c(2, 3),
  session = c("s1", "s2")
), session = "session")
by_session <- collapse_network(dn, sessions = "separate")
summary(by_session)
```

summary.dynet_metric *Summarise a temporal measure*

Description

Collapses the time dimension. Node-level measures are summarised one row per vertex and measure; graph-level measures one row per measure. The peak time is reported alongside, because when a quantity peaked is usually the question a temporal network is being asked.

Usage

```
## S3 method for class 'dynet_metric'
summary(object, by = NULL, ...)
```

Arguments

object	A dynet_metric.
by	Grouping for the summary: "node", "time" or "measure". The default, NULL, groups by "node" when the measure has a node column and by "measure" otherwise. A session column, when the measure has one, and measure itself are always part of the grouping as well. A grouping the measure has no column for – "node" on a graph-level series, say – is dropped rather than raising, leaving the grouping the measure does carry.
...	Ignored.

Value

A data.frame with the grouping columns plus n, mean, sd, min, max and, when time is available, peak_time. n counts the measured values the statistics were computed from, so a vertex that was inactive for part of the calendar reports fewer than the number of time points.

Examples

```
dn <- dynet(school_contacts)
degree <- centrality_series(dn, measure = "degree")
summary(degree)
summary(degree, by = "time")
```

summary.dynet_paths	<i>Summarise time-respecting paths</i>
---------------------	--

Description

Summarise time-respecting paths

Usage

```
## S3 method for class 'dynet_paths'
summary(object, ...)
```

Arguments

object	A dynet_paths.
...	Ignored.

Value

A data.frame with columns property and value, one row per property, both character so the table prints as one block. The eight properties are source, direction, reachable, reachable share, median latency, max latency, median hops and max hops; the source is excluded from every count and share. Under sessions = "separate" a leading session column is added and the eight properties are repeated for each session.

Examples

```
dn <- dynet(school_contacts)
routes <- paths(dn, from = "Ana")
summary(routes)
```

summary.dynet_pathways	<i>Summarise ranked pathways</i>
------------------------	----------------------------------

Description

Summarise ranked pathways

Usage

```
## S3 method for class 'dynet_pathways'
summary(object, ...)
```

Arguments

object	A dynet_pathways result.
...	Ignored.

Value

A plain data.frame, one row per endpoint: endpoint, the number of distinct routes reaching it, their summed count and share, the min_hops of the shortest, and first_arrival, the earliest time any route lands there. Ordered by count.

Examples

```
dn <- dynet(school_contacts)
routes <- pathways(dn, from = "Ana")
summary(routes)
```

```
summary.dynet_path_trajectories
      Summarise path trajectories
```

Description

Summarise path trajectories

Usage

```
## S3 method for class 'dynet_path_trajectories'
summary(object, ...)
```

Arguments

object	A dynet_path_trajectories result.
...	Ignored.

Value

A plain data.frame, one row per depth in increasing order: depth, the number of distinct branches reaching it, the number of distinct vertices they land on, the summed count of routes through it, and mean_branching, the average branching fraction of those routes. Note probability in the underlying table is CONDITIONAL on each parent, so it is averaged rather than summed: adding conditional fractions across siblings would not be a probability at all. Depth zero is the queried vertex itself and has no parent, so its mean_branching is NA.

Examples

```
dn <- dynet(school_contacts)
routes <- paths(dn, from = "Ana")
trajectories <- path_trajectories(routes)
summary(trajectories)
```

```
summary.dynet_projection
```

Summarise a time projection

Description

Summarise a time projection

Usage

```
## S3 method for class 'dynet_projection'
summary(object, ...)
```

Arguments

object	A dynet_projection result.
...	Ignored.

Value

A plain data.frame, one row per slice: slice, its time, the number of active vertex states, the within_slice arcs induced in it, and the identity_arcs leaving it for the next slice. The final slice emits no identity arcs, so its count is zero.

Examples

```
dn <- dynet(school_contacts)
slices <- projection(dn, step = 5, window = 5)
summary(slices)
```

```
summary.dynet_pshifts
```

Summarise participation shifts by family

Description

Summarise participation shifts by family

Usage

```
## S3 method for class 'dynet_pshifts'
summary(object, ...)
```

Arguments

object	A dynet_pshifts result.
...	Ignored.

Value

A plain data.frame, one row per shift family, ordered by descending count: family, its count, the share of all classified transitions it accounts for, and top_shift, the single most frequent shift type within it. share is NaN when nothing was classified. The family totals are sums of the count column as it stands, so they are transition counts for an output = "final" result.

Examples

```
dn <- dynet(school_contacts)
shifts <- pshifts(dn)
summary(shifts)
```

```
summary.dynet_similarity
      Summarise snapshot similarity
```

Description

Summarise snapshot similarity

Usage

```
## S3 method for class 'dynet_similarity'
summary(object, ...)
```

Arguments

```
object      A dynet_similarity result.
...         Ignored.
```

Value

A plain data.frame, one row per bin: time, the mean, min and max similarity to every other bin, and nearest, the time of the most similar other bin. The self-comparison is excluded throughout, so a bin with no comparable neighbour reports NaN and NA.

Examples

```
dn <- dynet(school_contacts)
bin_similarity <- similarity(dn, step = 5, window = 5)
summary(bin_similarity)
```

```
summary.dynet_snapshot
```

Summarise snapshot edges by time bin

Description

Summarise snapshot edges by time bin

Usage

```
## S3 method for class 'dynet_snapshot'
summary(object, ...)
```

Arguments

object	A dynet_snapshot from <code>snapshots()</code> .
...	Ignored.

Value

A plain `data.frame` with one row per bin and columns `session` (when present), `time`, `ties`, `nodes` and `weight`.

Examples

```
dn <- dynet(school_contacts)
bins <- snapshots(dn)
summary(bins)
```

```
synthdata
```

Synthetic code-transition network (Trees of Thought stand-in)

Description

A synthetic temporal network of how one interaction code follows another in asynchronous discussion. Vertices are the ten codes a message can carry, so a vertex is a **category**, **never a person**, and a tie runs from the code of a message to the code of the message it replies to.

Usage

```
synthdata
```

Format

A data.frame with 101 rows and 5 columns:

from Character. Code of the replying message.

to Character. Code of the message replied to.

start Numeric. When the spell opens, 0 to 2.10.

end Numeric. When it closes, 0.03 to 5.80.

weight Numeric, whole-valued. Message pairs the spell represents, 1 to 5881.

The ten codes are Acceptance, Argument, Composing, Disagreement, Evaluation, Group_regulation, Question, Sharing, Socioemotional and T.Regulation; 14 rows are self-loops.

Details

This is a stand-in for the network analysed in the *Trees of Thought* study, whose own data is not redistributable. It was built by drawing a random 70% subset of that study's edge spells and resampling that subset with replacement to 90% of the original spell count, so it omits about a third of the real spells and repeats others. **No row of it should be read as a finding about the study**, and figures computed from it will not match the published ones. It exists so the analysis can be run and taught end to end.

Spells are weighted by how many message pairs they represent, overlap in time, and include self-loops, because a code following itself is a real and common transition. Build with loops = TRUE to keep them; the weight column is picked up as the tie weight.

Source

Synthesised from the *Trees of Thought* code-transition network by the resampling described above; see data-raw/synthdata.R.

Examples

```
dynet(synthdata, directed = TRUE, loops = TRUE, weight = "weight")
```

tail.dynet_metric	<i>Last rows of a temporal measure</i>
-------------------	--

Description

The counterpart of `head.dynet_metric()`; the header still describes the series and a last `n` of `N` rows line records the truncation.

Usage

```
## S3 method for class 'dyenet_metric'
tail(x, n = 6L, ...)
```

Arguments

x A dynet_metric.
n Number of rows to keep. Defaults to six.
... Passed to the default method.

Value

A dynet_metric with at most n rows, carrying the source counts so its header stays true to the series.

Examples

```
dn <- dynet(school_contacts)
degree <- centrality_series(dn, step = 4, window = 4)
tail(degree)
```

thought_chains	<i>Trees of Thought reply links, augmented by simulation</i>
----------------	--

Description

A table of reply links between the codes of messages in coded asynchronous discussions, based on the *Trees of Thought* study. Each row is one link from the code of a message to the code of the message it replies to. About 20 percent of the study's records were removed, dates and rates were changed and anonymised, and the data were augmented by simulation, so the table is not the study's data, and the participant, group, course and time values do not identify anyone.

Thursdays and Fridays do not occur. Some rows repeat exactly (8,122 duplicates); aggregating the log counts them as weight. A code answering itself is a self-link (9,452 rows); `dynet()` drops these unless `loops = TRUE`. The course column is recognised as the session column, so the five courses become sessions unless `session =` says otherwise.

Usage

```
thought_chains
```

Format

A data.frame with 23,017 rows and 7 columns:

from Character. Code of the replying message, one of the nine codes Approving, Arguing, Coordinating, Drafting, Inquiring, Objecting, Resourcing, Socialising, Tutoring.

to Character. Code of the message replied to, same set.

time POSIXct (UTC) time of the replying message, 2006-09-23 to 2011-11-02; changed and anonymised, not the study's dates.

participant Character. Author label, P001 to P240.

discussion Integer discussion (thread) id, 1 to 1169, all present.

group Character. Course group, A_01 style; 29 groups.

course Character. Course, A to E.

Source

Based on the *Trees of Thought* study of coded asynchronous discussions, with about 20 percent of the records removed, dates and rates changed and anonymised, and the data augmented by simulation: Saqr, M., López-Pernas, S. and Törmänen, T. (2026). A temporal network approach to reveal the longitudinal dynamics of CSCL group regulation and productive collaboration. *International Journal of Computer-Supported Collaborative Learning*, 21, 237-270. doi:10.1007/s11412025-094645

Examples

```
dynet(thought_chains, time = "time", loops = TRUE)
dynet(thought_chains, thread = "discussion")
```

update_nodes	Update static node attributes
--------------	-------------------------------

Description

Update static node attributes

Usage

```
update_nodes(dn, data)
```

Arguments

- dn A temporal network.
- data A nonempty data frame with a name key and one or more attributes to add or replace. Only named nodes are changed.

Value

A new dynet object, class c("dynet", "netobject", "cograph_network"), with the same spells, vertex activity and metadata as dn and the supplied attributes added to or replaced on the named vertices. Unnamed vertices keep their existing values, gaining NA in any column the network did not already have. Raises dynet_unknown_node when a name is not a vertex, and dynet_bad_input when data is malformed or names a cograph structural column (id, label, x, y).

Examples

```
dn <- dynet(data.frame(from = "A", to = "B", start = 0, end = 1))
update_nodes(dn, data.frame(name = "A", role = "initiator"))
```

update_ties	<i>Update temporal ties and their attributes</i>
-------------	--

Description

Update temporal ties and their attributes

Usage

```
update_ties(dn, ties, data, loops = FALSE)
```

Arguments

dn	A temporal network.
ties	Which ties to update: a condition on the spell table, evaluated the way <code>subset()</code> evaluates one, over the columns <code>as.data.frame(dn)</code> returns; or integer row positions or a logical mask over that table.
data	A data frame with one row or one row per selected tie. Columns may be canonical tie fields or arbitrary atomic spell attributes.
loops	Whether an endpoint update may introduce a new self-loop. Existing loops may always be retained. Default FALSE.

Value

A new dynet object, class `c("dynet", "netobject", "cograph_network")`, holding the unselected spells unchanged and the selected spells with the supplied values substituted. Because the edited spells are rebuilt together with the rest, spell order and the canonical spell identifiers may change. Raises `dynet_loop_not_allowed` when an endpoint update would create a new self-loop without `loops = TRUE`, and `dynet_bad_input` when the selection or data is malformed, or names the read-only derived columns `duration` or `.raw_spell`.

Examples

```
dn <- dynet(school_contacts)
marked <- update_ties(dn, ties = end - start > 1,
                      data = data.frame(kind = "long"))
marked
```

update_vertex_spells *Update declared vertex-activity components*

Description

Update declared vertex-activity components

Usage

```
update_vertex_spells(dn, spells, data)
```

Arguments

dn	A temporal network.
spells	Integer positions or a logical mask over canonical vertex activity, as returned by <code>as.data.frame(dn, what = "vertex_spells")</code> .
data	A data frame with one row, or one row per selected component, containing the fields to replace: <code>node</code> , <code>start</code> , <code>end</code> , <code>session</code> , <code>onset_censored</code> or <code>terminus_censored</code> . Any other column name raises <code>dynet_unknown_column</code> , so a misspelled field is refused rather than quietly doing nothing.

Value

A new dynet object, class `c("dynet", "netobject", "cograph_network")`. Updated components are canonicalised with the retained components, so overlaps can merge and spell identifiers can change. Raises `dynet_bad_input` when `spells` is not a valid selection, when `data` is malformed or of the wrong height, or when it names the read-only derived columns `vertex_spell`, `duration` or `instant`.

Examples

```
dn <- dynet(school_contacts)
present <- data.frame(node = c("Ana", "Ben"), start = 0, end = 10)
enrolled <- set_vertex_spells(dn, present)
extended <- update_vertex_spells(enrolled, spells = 1,
                                data = data.frame(end = 12))
as.data.frame(extended, what = "vertex_spells")
```

Index

* datasets

- forum_people, 46
- forum_posts, 47
- mooc_people, 56
- mooc_posts, 56
- school_contacts, 99
- seminar_attendance, 100
- synthdata, 114
- thought_chains, 116

add_arcs, 4

add_nodes, 5

add_ties, 5

add_ties(), 4

add_vertex_spells, 6

animate, 7

animate(), 14, 81, 107

as.data.frame(), 32, 42, 52, 91, 105

as.data.frame.dynet, 11

as.data.frame.dynet_animation, 13

as.data.frame.dynet_collapsed, 14

as.data.frame.dynet_collapsed_list, 15

as.data.frame.dynet_metric, 16

as.data.frame.dynet_path_network, 19

as.data.frame.dynet_path_trajectories, 20

as.data.frame.dynet_paths, 17

as.data.frame.dynet_pathways, 18

as.data.frame.dynet_projection, 21

as.data.frame.dynet_pshifts, 22

as.data.frame.dynet_similarity, 23

as.data.frame.dynet_snapshot, 23

as_dynet, 24

as_dynet(), 26, 36

as_dynet.networkDynamic, 25

as_dynet.networkDynamic(), 24

base::as.data.frame(), 14

burstiness, 27

centrality_series, 28

centrality_series(), 8, 11, 12, 27, 37, 44, 50, 54, 62, 64, 70, 103, 105

clear_observations, 35

collapse_network, 35

collapse_network(), 15, 72, 82

durations, 37

durations(), 17, 73

dynet, 39

dynet(), 4–6, 8, 12, 24, 26, 27, 29, 36, 37, 44, 46, 47, 50, 51, 53, 54, 56, 58, 61, 63, 68, 80, 89, 90, 92, 94–96, 100–103, 105, 106, 116

events, 43

events(), 10, 103, 104

forum_people, 46, 47

forum_posts, 46, 47

ggplot2::theme_minimal(), 75, 80

graphics::hist(), 27, 30, 38, 44, 51, 54, 58, 61, 63, 66, 91, 93, 104, 105

grDevices::png(), 8

head.dynet_metric, 48

head.dynet_metric(), 115

induce_subgraph, 48

match.arg(), 91

metrics, 49

metrics(), 105

mixing, 53

mixing(), 17, 46, 52, 73

mooc_people, 56, 57

mooc_posts, 56, 56

path centrality, 62

path centrality(), 28, 34, 58, 92

path_network, 64
 path_network(), 20, 66, 76
 path_trajectories, 65
 path_trajectories(), 21, 61, 62, 79, 80, 85
 paths, 57
 paths(), 18, 61–66, 74, 79, 80, 84, 92, 93
 pathways, 61
 pathways(), 19
 plot(), 32, 43, 52, 91, 105
 plot.dynet, 67
 plot.dynet(), 8, 9, 11, 72, 74, 76, 78, 80
 plot.dynet_collapsed, 72
 plot.dynet_metric, 72
 plot.dynet_path_network, 76
 plot.dynet_path_trajectories, 76
 plot.dynet_paths, 74
 plot.dynet_pathways, 75
 plot.dynet_pshifts, 77
 plot.dynet_similarity, 78
 plot.dynet_snapshot, 78
 plot_path_trajectories, 79
 plot_path_trajectories(), 66, 74, 76, 77
 print(), 105
 print.dynet, 80
 print.dynet_animation, 81
 print.dynet_collapsed, 82
 print.dynet_collapsed_list, 82
 print.dynet_metric, 83
 print.dynet_path_trajectories, 85
 print.dynet_paths, 84
 print.dynet_pathways, 84
 print.dynet_projection, 86
 print.dynet_pshifts, 86
 print.dynet_similarity, 87
 print.dynet_snapshot, 88
 projection, 88
 projection(), 21, 86
 pshifts, 90

 reachability, 92
 reachability(), 34, 62, 64
 remove_arcs, 94
 remove_nodes, 95
 remove_ties, 96
 remove_ties(), 94, 95
 remove_vertex_spells, 97
 rename_nodes, 97
 rename_sessions, 98

 school_contacts, 99
 seminar_attendance, 100
 set_observations, 100
 set_tie_sessions, 101
 set_vertex_spells, 102
 set_vertex_spells(), 6, 58
 similarity, 103
 similarity(), 23, 78, 87
 snapshots, 104
 snapshots(), 7, 8, 11, 24, 31, 78, 88, 104, 114
 subset(), 49, 94, 96, 118
 summary(), 32, 43, 51, 52, 91, 105
 summary.dynet, 106
 summary.dynet_animation, 107
 summary.dynet_collapsed_list, 108
 summary.dynet_metric, 109
 summary.dynet_path_trajectories, 111
 summary.dynet_paths, 110
 summary.dynet_pathways, 110
 summary.dynet_projection, 112
 summary.dynet_pshifts, 112
 summary.dynet_similarity, 113
 summary.dynet_snapshot, 114
 synthdata, 114

 tail.dynet_metric, 115
 thought_chains, 116

 update_nodes, 117
 update_ties, 118
 update_vertex_spells, 119
 update_vertex_spells(), 7