# Package 'networkDynamic'

February 16, 2023

**Version** 0.11.3

**Date** 2023-02-15

**Title** Dynamic Extensions for Network Objects

**Type** Package

**Depends** R (>= 3.0.0), network (>= 1.17.0)

**Imports** statnet.common, methods, networkLite

**Suggests** testthat

**LinkingTo** network

**Description** Simple interface routines to facilitate the handling of network objects with complex intertemporal data. This is a part of the ``statnet'' suite of packages for network analysis.

**License** GPL-3

**LazyLoad** yes

**URL** https://statnet.org/

**NeedsCompilation** yes

**Author** Carter T. Butts [aut],
Ayn Leslie-Cook [aut],
Pavel N. Krivitsky [aut],
Skye Bender-deMoll [aut, cre],
Zack Almquist [ctb],
David R. Hunter [ctb],
Li Wang [ctb],
Kirk Li [ctb],
Steven M. Goodreau [ctb],
Jeffrey Horner [ctb],
Martina Morris [ctb]

**Maintainer** Skye Bender-deMoll <skyebend@uw.edu>

**Repository** CRAN

**Date/Publication** 2023-02-16 08:20:02 UTC

# R topics documented:



---

`networkDynamic-package`

*Dynamic Extensions for Network Objects*

---

## Description

Simple interface routines to facilitate the handling of dynamic network objects with different types of temporal data. This allows the user to create, store and query networks that change over time. Changes include edges that form and dissolve over time, and vertices that enter or leave the network. The package also includes support for defining and querying temporally changing attributes of vertices, edges, and network properties.

## Details

| | |
|---|---|
| Package: | networkDynamic |
| Type: | Package |
| Version: | 0.7 |
| Date: | 2014-09-25 |
| Depends: | network |
| License: | GPL-3 |
| LazyLoad: | yes |

The networkDynamic package provides support for a simple family of dynamic extensions to the network class; these employ the standard network attribute functionality (and hence the resulting objects are still compatible with all conventional routines), but greatly facilitate the practical storage, manipulation and query of dynamic network data.

The basis for the dynamic extensions is described in activity.attribute. Dynamic attributese are explained in attribute.activity.functions. Id systems which are persistent throughout manipulations and changes in network size are explained in persistent.ids. Other core routines can be found in the links below.

Additional example data sets can be found in the networkDynamicData package.

## Author(s)

Originally created by Carter T. Butts <buttsc@uci.edu>,

Current Maintainer: Skye Bender-deMoll skyebend@uw.edu

Contributions from: Pavel Krivitsky <pavel@uow.edu.au>, Ayn Leslie-Cook <aynlc3@uw.edu>, David Hunter <dhunter@stat.psu.edu>, Li Wang <lxwang@gmail.com>, Kirk Li <kirkli@uw.edu>, StevenGoodreau <goodreau@uw.edu>, Zack Almquist <almquist@uci.edu>, Jeffrey Horner <jeffrey.horner@gmail.com Martina Morris <morrism@u.washington.edu>

With support from the statnet team https://statnet.org

## See Also

activity.attribute, activate, is.active, network.extract, network.extensions network

---

activate *Dynamic Activation and Inactivation of Edges and Vertices*

---

## Description

activate and deactivate set spells of activity and inactivity respectively for elements (edges and vertices) in a dynamic network.

## Usage

```
activate.edges(x, onset = NULL, terminus = NULL, length = NULL, at = NULL,
    e = seq_along(x$mel))
activate.vertices(x, onset = NULL, terminus = NULL, length = NULL, at = NULL,
    v = seq_len(network.size(x)))

deactivate.edges(x, onset = NULL, terminus = NULL, length = NULL, at = NULL,
    e = seq_along(x$mel))
deactivate.vertices(x, onset = NULL, terminus = NULL, length = NULL, at = NULL,
    v = seq_len(network.size(x)), deactivate.edges = FALSE)
```

## Arguments

| | |
|---|---|
| x | an object of class network. |
| onset | an optional vector of timepoints that specifies the starts of the interval(s). This must be accompanied by one of terminus or length. |
| terminus | an optional vector of timepoints that specifies the ends of the interval(s). This must be accompanied by one of onset or length. |
| length | an optional vector of interval lengths for the interval(s). This must be accompanied by one of onset or terminus. |
| at | optional, one or more time points to be activated. |
| e | optional, one or more IDs indicating edges to be modified. Ids can be repeated to indicate multiple time values per element. |
| v | optional, one or more IDs indicating vertices to be modified. Ids can be repeated to indicate multiple time values per element. |
| deactivate.edges | optional, TRUE setting will automatically deactivate all the edges associated with the vertex being deactivated |

## Details

An element in a dynamically extended network class is considered to be either "active" or "inactive" at any point in time, with the substantive meaning of "activity" determined by the specific application context. The activate and deactivate functions provide an interface for controlling the state of the elements in a dynamic network.

activate.edges and activate.vertices have identical behavior, except for the elements they modify (the same is true for the deactivate.* functions).

There are several ways to specify the activity spell, and the general syntax rules are described at activity.attribute. Activity can be set for a single time point, using either at, or by setting onset=terminus. Activity can be set for an interval [onset,terminus), using a valid combination of the onset, terminus and length attributes.

This allows for a wide range of options when specifying spells, but a correct specification must use only one of these forms:

```
at
```

```
onset and terminus
onset and length
terminus and length
or, you can provide no timing information
```

If provided with no timing information, the element is activated/deactivated from `-Inf` to `Inf`. The specified interval spans the period from the onset (inclusive) to the terminus (exclusive), so [onset,terminus).

There are some special behaviors associated with the arguments `Inf` and `-Inf`.

- The open-ended interval `c(-Inf,x)` includes `-Inf`. For consistency, we also allow the open-ended interval `c(x, Inf)` to include `Inf`. Thus [onset, terminus) will be interpreted as [onset, terminus] when terminus = `Inf`.
- The arguments `Inf` or `-Inf` are only valid when used to specify an interval, they can not be used to specify status at a time point using `at`. In addition, they cannot be paired with themselves in a call. That is, both `(Inf,Inf)` and `(-Inf,-Inf)` are not valid specifications for any spell.
- Both `deactivate.*(x)` and `deactivate.*(x, -Inf, Inf)` create the null spell – specifying inactivity over the entire time span. Note that by convention the null spell is stored as `(Inf,Inf)`.

Calling the activation/deactivation functions with a vector of spell modifiers and a vector of elements to be modified (for example, `at=c(1,3,5 7), v=c(1:4)`) allows multiple elements in the network to be activated/deactivated simultaneously (note, not multiple spells for a single element). The spell modifiers are applied sequentially to the selected elements. If the length of the spell vector is less than the number of elements, the spell modifiers are recycled as needed. When multiple network elements are activated in a single call, the spell modifiers must all be of one type, either `at`, or a valid mix of `onset`, `terminus` and `length`.

The `activate.*` and `deactivate.*` functions in general modify spells in similar, if opposite, ways. However, there are some behaviors that are specific to each function.

- Effects on an element that has no existing activity attributes.
  - For activate: the element is marked as being inactive before the onset time, then active for the specified spell, then inactive from the terminus on.
  - For deactivate: the element is marked as being active before the onset time, then inactive for the specified spell, then active from the terminus on (i.e., the opposite of `activate.*`.
- Effects of the `at` specification.
  - For activate: the element is activated at that time point.
  - For deactivate: the element is deactivated at the time point, but only if the time point is currently activated as a 0-length spell.

Currently, there is no support for activating multiple spells for a single element in a single call. To activate 2 spells for a node, for example, one must call activate.vertices twice. It is advisable to remove duplicate edges or vertices from the `e` or `v` input vectors.

Edge/vertex activity is tracked through an attribute called (eponymously) "active", and which is explained in more detail under activity.attribute. This may be modified or otherwise accessed using standard class methods (e.g., `get.edge.attribute`), as well as the specialized methods described here.

## Value

None. (But all four commands modify their arguments as a side-effect.)

## Note

This function may not be entirely stable in the long run, since it makes explicit reference to internal elements of the network object; some syntactic changes could occur in the future.

## Author(s)

Ayn Leslie-Cook <aynlc3@uw.edu>, Carter T. Butts <buttsc@uci.edu>

## References

~put references to the literature/web site here ~

## See Also

is.active

## Examples

```
triangle <- network.initialize(3)  # create a toy network
add.edge(triangle,1,2)    # add an edge between vertices 1 and 2
add.edge(triangle,2,3)    # add more edges
add.edge(triangle,3,1)

# turn on all edges at time 1 only (0 length spell)
activate.edges(triangle,at=1)

# activate edge (1,2) from t=2 to t=3
activate.edges(triangle,onset=2, terminus=3,
               e=get.edgeIDs(triangle,v=1,alter=2))

# activate edge (2,3) from t=4 for 2 time lengths
activate.edges(triangle,onset=4, length=2,
               e=get.edgeIDs(triangle,v=2,alter=3))

deactivate.edges(triangle, at=2, e=1) # does not work since the spell is not 0-length
is.active(triangle, at=2, e=1:3)

deactivate.edges(triangle, e=1, onset=2, length=0.1)  # this deactivates the vertex
is.active(triangle, at=2, e=1:3)

# this activates edges 2 and 3 at time 5
activate.edges(triangle, e=2:3, at=5)

# this activates edge 1 at multiple times;
deactivate.edges(triangle)
activate.edges(triangle, e=c(1,1,1), at=6:8)
is.active(triangle, at=6, e=1:3)
```

```
is.active(triangle, at=7, e=1:3)
is.active(triangle, at=8, e=1:3)

# this activates vertex 1 for two spells, (0,1) and (3,4)
test <- network.initialize(3)
activate.vertices(test,onset=0:3,terminus=1:4,v=c(1,2,3,1))
get.vertex.activity(test, as.spellList=TRUE)
```

---

`activity.attribute` *Activity Attributes for Dynamically Extended Networks*

---

## Description

Dynamically extended networks are `networkDynamic` class objects (extensions of network objects) whose vertex and edge elements include timing information. This page describes the types of timing information currently available, and the general syntax for setting or querying this information.

## Details

Currently, each edge and vertex in a dynamically extended network is presumed to be in one of two states at any given point in time: ("active" or "inactive"). The state of a network element (i.e., edge or vertex) is governed by an attribute with the name "active", which is considered a reserved term for purposes of this package.

The activity attribute consists of a two-column numeric matrix. Each row contains an activity spell, and the two columns encode onset and terminus times respectively. Elements are presumed inactive until the onset of their first activity spell, and are likewise presumed inactive after the termination of their last active spell.

Spells are taken to span the period from the onset (inclusive) to the terminus (exclusive), so [onset,terminus). They must be consecutive, ordered forward in time, and strictly non-overlapping.

The syntax for defining or querying spells can specify either an instantaneous time point or an interval. The commands for this include `at=`, `onset=`, `terminus=` and `length=`. Any numeric values may be used in the interval specifications, including `Inf` and `-Inf`, (with some restrictions, see below). A spell with `onset=-Inf` represents onset censoring. A spell with `terminus=Inf` represents terminus censoring. `Inf` and `-Inf` cannot be used with the `at` specification. Similarly, `onset` can not be `Inf` and `terminus` can not be -Inf.

The general syntax rules for specifying spells are as follows:

To specify a spell as a single time point:

- use the `at` argument, or
- use `onset=terminus`.

To specify a spell as a particular interval, one of the following combinations of `onset`, `terminus` and `length` is required:

- `onset` and `terminus`

- onset and length
- terminus and length

The special "null" spell is used to designate elements that are never active (i.e., have no valid activity spells). These can only be set by the deactivate function (see activity.attribute) and by convention are stored as c(Inf,Inf). "Null" spells are incompatible with other spells, and should be replaced whenever an activation enters the element's event history.

Although it is possible to access and modify the activity spells using network attribute methods (e.g., get.edge.attribute) it is not recommended, and extreme care should be taken to preserve the structure of the spell matrix. The preferred way to modify the spells of a network element is with the activate related methods. Vertices and edges with specific activity ranges can be selected using network.extensions.

In addition, a number of special functions are also provided to simplify common tasks related to the active attribute (noted below).

## Author(s)

Ayn Leslie-Cook <aynlc3@uw.edu>, Carter T. Butts <buttsc@uci.edu>

## See Also

is.active, activate, activate.vertices,activate.edges, deactivate.vertices,deactivate.edges,network.exte

## Examples

```
triangle <- network.initialize(3)  # create a toy network

activate.vertices(triangle,onset=1,terminus=5,v=1)
activate.vertices(triangle,onset=1,terminus=10,v=2)
activate.vertices(triangle,onset=4,terminus=10,v=3)
deactivate.vertices(triangle,onset=2, length=2, v=1)

get.vertex.activity(triangle) # vertex spells
```

---

# add.methods *PID-aware versions of functions for adding edges and vertices to networkDynamic objects*

---

## Description

The methods (add.vertices.networkDynamic, add.edge.networkDynamic, add.edges.networkDynamic) override their network-package counterparts in order to permit assigning persistent.ids to newly-added elements if the pid is defined. They can be defined by the user with the vertex.pids argument, or, if not specified, a unique random id will be generated. (Note that any new values added by default to a vertex.names pid will not be numeric.)

## Usage

```
## S3 method for class 'networkDynamic'
add.vertices(x, nv, vattr = NULL, last.mode = TRUE,
    vertex.pid = NULL, ...)

## S3 method for class 'networkDynamic'
add.edges(x, tail, head, names.eval = NULL, vals.eval = NULL,
    edge.pid = NULL, ...)

## S3 method for class 'networkDynamic'
add.edge(x, tail, head, names.eval = NULL, vals.eval = NULL,
    edge.check = FALSE, edge.pid = NULL, ...)
```

## Arguments

| | |
|---|---|
| x | an object of class network or networkDynamic (see add.vertices in network) |
| nv | number of vertices to be added. (see add.vertices in network) |
| vattr | optionally, a list of attributes with one entry per new vertex. (see add.vertices in network), not interpreted as a dynamic attribute |
| last.mode | logical; should the new vertices be added to the last (rather than the first) mode of a bipartite network? (see add.vertices in network) |
| vertex.pid | a vector of vertex.pids to be assigned to the newly-added vertices |
| edge.pid | a vector of edge.pids to be assigned to the newly-added edges |
| names.eval | see add.edges in network), not interpreted as a dynamic attribute |
| vals.eval | see add.edges in network), not interpreted as a dynamic attribute |
| edge.check | see add.edges in network) |
| head | see add.edges in network) |
| tail | see add.edges in network) |
| ... | possible additional arguments |

## Details

The networkDynamic versions of these methods call the network versions internally. See persistent.ids for additional details about the PID mechanism.

## Value

- add.verticies (invisibly) returns a reference to the network with the new vertices added. Pids will be added/created if a vertex.pid attribute is set.
- add.edge and add.edges (invisibly) return a reference to the network with the new edges added. Pids will be added/created if a vertex.pid attribute is set.

## Note

Adding edges via the extraction/replacement operators `[,]` bypasses the pid code and will break the edge pids defined for the network. Similarly, `add.vertices.active` and `add.edges.active` do not yet support including pids.

## Author(s)

lxwang, skyebend, the statnet team

## See Also

See also `persistent.ids`, `add.vertices`, `add.edge`, `add.edges` in network.

## Examples

```
# add vertices while using vertex.names as pids
nd <- as.networkDynamic(network.initialize(3))
set.network.attribute(nd,"vertex.pid","vertex.names")
add.vertices(nd,nv=2,vertex.pid=c(4,5)) # specify pids for two new vertices
network.vertex.names(nd) # peek at pids

# add vertices and edges w/ auto-generated pids
nd <- as.networkDynamic(network.initialize(3))
initialize.pids(nd)
get.vertex.attribute(nd,'vertex.pid') # peek at pids
add.vertices(nd,nv=2) # silently generate pids for vertices
get.vertex.attribute(nd,'vertex.pid') # peek at pids
add.edges(nd,1,2) # silently generate a pid for the edge
get.edge.attribute(nd,'edge.pid')
```

---

# add.vertices.active *add.verticies.active, add.edges.active*

---

## Description

Convenience functions for adding a set of verticies (or edges) and setting them to be active in a single call

## Usage

```
## S3 method for class 'active'
add.vertices(x, nv, vattr = NULL, last.mode = TRUE,
            onset = NULL, terminus = NULL, length = NULL, at = NULL, ...)

## S3 method for class 'active'
add.edges(x, tail, head, names.eval = NULL, vals.eval = NULL,
         onset = NULL, terminus = NULL, length = NULL, at = NULL, ...)
```

## Arguments

| | |
|---|---|
| x | an object of class network or networkDynamic. |
| nv | the number of vertices to add |
| tail | a vector of vertex IDs corresponding to the tail (source, ego) of each edge to be added |
| head | a vector of vertex IDs corresponding to the head (target, alter) of each edge to be added |
| onset | an optional vector of time points that specifies the starts of the interval(s). This must be accompanied by one of terminus or length. |
| terminus | an optional vector of time points that specifies the ends of the interval(s). This must be accompanied by one of onset or length. |
| length | an optional vector of interval lengths for the interval(s). This must be accompanied by one of onset or terminus. |
| at | optional, one or more time points to be activated. |
| names.eval | optional list of length equal to the number of edges, with each element containing a list of names for the attributes of the corresponding edge. not currently interpreted in a dynamic context, but passed directly to add.edges |
| vals.eval | an optional list of lists of edge attribute values (matching names.eval). Not currently interpreted in a dynamic context, but passed directly to add.edges |
| vattr | optionally, a list of attributes with one entry per new vertex. not currently interpreted in a dynamic context, but passed directly to add.vertices |
| last.mode | logical; should the new vertices be added to the last (rather than the first) mode of a bipartite network? |
| ... | possible future additional arguments |

## Details

Essentially a wrapper for a call to add.vertices and activate.vertices or add.edges and activate.edges when setting up a network object. These are not the S3 methods that their name appears to imply, since there is no "active" class. See add.edges.networkDynamic, etc.

## Value

The passed in network object with class set to networkDynamic and the specified number of new vertices or edges added and activated

## Note

Order of arguments was changed in version 1.9 for S3 method consistency. Does not currently support the multiple-vertex head- and tail-sets of add.edges or add.edge.

## Author(s)

Ayn Leslie-Cook <aynlc3@uw.edu>

## See Also

See Also as activate.vertices, activate.edges,add.vertices,add.edges

## Examples

```
nw <- network.initialize(5)
activate.vertices(nw,onset=0,terminus=10)
network.size(nw)  # just 5 nodes
# add some new nodes with times
add.vertices.active(nw,2,onset=10,terminus=12)
network.size(nw)  # now we have 7 nodes

# add 2 edges edges, and activate them
add.edges(nw, tail=c(1,2),head=c(2,3))
activate.edges(nw,onset=0,terminus=10,e=1:2)

# instead add and activate at the same time
add.edges.active(nw, tail=c(3,4),head=c(4,5),onset=10,terminus=12)
```

---

# adjust.activity *Adjust the activity ranges in all of the spells of a networkDynamic object*

---

## Description

Transforms the values of all the activity spells of vertices and edges, as well as spells of dynamic TEA attributes (and the net.obs.period if it exists).

## Usage

```
adjust.activity(nd, offset = 0, factor = 1)
```

## Arguments

| | |
|---|---|
| nd | networkDynamic object to be modified |
| offset | numeric value to be added (or subtracted) to all spell values |
| factor | numeric factor for multiplying all spell values |

## Details

Spell values are modified by first adding the value of offset and then multiplying by factor. If a net.obs.period attribute exists, it's observation spell values are transformed, and the time.increment value is multiplied by factor.

## Value

The nd argument is modified in place and returned invisibly.

### Author(s)

skyebend@uw.edu

### See Also

See also activity.attribute, attribute.activity.functions and net.obs.period

### Examples

```
# convert steps of an imaginary discrete sim
# with each steps corresponding to 1 day
# into decimal 'years' since 1990
sim<-network.initialize(5)
activate.vertices(sim,onset=0,terminus=2546)
add.edges.active(sim,head=1:4,tail=2:5,onset=0,terminus=2546)
set.network.attribute(sim,'net.obs.period',list(
   observations=list(c(0,2546)),mode="discrete",
   time.increment=1,time.unit="step"))

# do the transformation
adjust.activity(sim,offset=1990*365.25,factor=1/365.25)

# modify the 'units' of net.obs.period
obs <-get.network.attribute(sim,'net.obs.period')
obs$time.unit<-'year'
set.network.attribute(sim,'net.obs.period',obs)

# peek at the new values
as.data.frame(sim)
get.network.attribute(sim,'net.obs.period')
```

---

## age.at *Age of active edges or vertices at a query time point*

---

### Description

Reports the age of edges or vertices at a specific time point. More precisely, the duration of time between the query time point and the onset of the activity spell active at that point.

### Usage

```
edges.age.at(nD, at, e = seq_along(nD$mel), active.default = TRUE)
dyads.age.at(nD, at, tails, heads, active.default=TRUE,
                     format.out = c("vector", "edgelist", "matrix"))
vertices.age.at(nD, at, v=seq_len(network.size(nD)), active.default=TRUE)
```

## Arguments

| | |
|---|---|
| nD | a networkDynamic object to be evaluated |
| at | numeric query time point which edge or vertex ages should be evaluated at |
| e | numeric vector of edges ids for which ages / durations should be reported |
| v | numeric vector of vertex ids for which ages / durations should be reported |
| tails | numeric vector of vertex ids incident on the 'tail' of edges for which ages / durations should be reported |
| heads | numeric vector of vertex ids incident on the 'head' of edges for which ages / durations should be reported |
| active.default | logical, if TRUE edges or vertices with no activity specified will be considered always active (see is.active) |
| format.out | character value indicating out the output should be structured. vector returns output as a vector, edgelist returns a 3-column matrix in which the first columns are the tail and head vertex ids of the edge and the 3rd column is the age, matrix returns an adjacency matrix with the edge ages as values. |

## Details

Edges or vertices that are not active at time at will return NA. For edges or vertices with multiple activity spells, this function *does not* report the total duration of activity across all spells, only the duration from the start of the spell with which the at point intersects.

dyads.age.at reports the age of edges corresponding to each dyad (tail,head). It cannot be used with hypergraphic or multiplex networks because a pair of vertex ids may not uniquely correspond to an edge. If tails and heads are not specified, they will default to the tails and heads of all existing (but not necessarily active) edges in the network. Ordering and index position should correspond to valid.eids.

## Value

By default, a numeric vector indicating the age of the network element at the query time point, or NA if the element is not active or (in the case of edges) deleted. Elements of the vector return correspond to the values of e or v or (tails,heads) respectively.

## Author(s)

skyebend

## See Also

See also get.edge.activity, is.active

## Examples

```
# set up an example network for testing
test<-network.initialize(5)
add.edges(test,tail = 1:4,head=2:5 )
activate.edges(test,onset=0:2,terminus=c(2,3,3),e=1:3)
```

```
activate.vertices(test,v = 1:4,onset=-3:0,terminus=5:8)

# how old are each of the edges at time point 2.0 ?
edges.age.at(test,at=2.0)

# how old is vertex id 2 time point 3
vertices.age.at(test,at=3, v=2)

# how old are the edges [1,2] and [3,4] at time point 2
dyads.age.at(test,at=2,tails=c(1,3),heads=c(2,4))

dyads.age.at(test,at=2,format.out='matrix')
```

---

`as.data.frame.networkDynamic` *Extracts the edge timing information from a networkDynamic object and represent as a data.frame.*

---

## Description

Creates a data.frame giving lists of edges and their activity spells corresponding to the `networkDynamic` object. An observation window may be specified, and edge spells with values outside the window will be omitted or truncated/censored to the observation window bounds.

## Usage

```
## S3 method for class 'networkDynamic'
as.data.frame(x, row.names = NULL, optional = FALSE, e =seq_along(x$mel),
              start = NULL, end = NULL, active.default=TRUE,...)
```

## Arguments

| | |
|---|---|
| `x` | a networkDynamic object |
| `row.names` | ignored |
| `optional` | ignored |
| `e` | vector of edge ids that should be included in the output. |
| `start` | optional, numeric onset-censoring time: time at which the network was first observed. Edge spells with earlier onset times are given this as their onset time (and marked as left censored). See Details for the default values. |
| `end` | optional right-censoring time: time at which the network was last observed. Edge spells with larger terminus are given this as their terminus time (and marked as right censored). See Details for the default values. |
| `active.default` | logical indicating if edges with no defined activity spells should be considered active by default (and appear as a row in output) |
| `...` | possible additional arguments |

## Details

Exports the edge dynamics of a networkDynamic object as a data.frame giving a table of edges with their activity spells, including whether an activity spell was "censored" (onset or termination was not observed within the observation window). If a net.obs.period attribute exists, the maximal and minimal values will be used as censoring times by default. Censoring times can also be provided using the start and end arguments (which will override net.obs.period), and censoring can be prevented by setting start=-Inf, end=Inf. Edge spells that exceed the censoring bounds will have their onset and/or terminus time appropriately truncated to match and will be marked as censored. Spells which are fully outside the censoring bounds will be deleted and the corresponding rows will not be returned.

Note that the spell overlap rules mean that when edge spells are modeled as momentary events (onset==terminus), any events have onset==terminus==end of the the observation window will not be returned. This will likely be the case for networks created by the networkDynamic converter function.

## Value

A data.frame containing timed relational information describing the networkDynamic object, having the following columns:

onset, terminus
: Onset and terminus times of an edge, respectively, if not censored, and left and right censoring times, if censored.

tail, head
: Tail and head of the edge.

onset.censored, terminus.censored
: TRUE if onset/terminus time is censored.

duration
: the duration (terminus-onset) of each spell, evaluated after censoring

edge.id
: the id of the edge the row corresponds to

The output data.frame will be sorted by edge.id, onset,terminus. This means that for most (non-multiplex) networks the group of spells for a specific edge will appear in sequential lines of output, but the overall output is not in a temporal order.

## Note

Vertex and attribute activity are not included in the returned data.frame. If an edge has multiple activity spells, it will have multiple rows in the output. When output in censored, it may not match values returned by get.edge.activity

## See Also

See Also as get.edge.activity,get.vertex.activity

## Examples

```
net <-network.initialize(3)
net[1,2]<-1;
net[2,3]<-1;
```

```
## censoring
activate.edges(net,onset=1,terminus=Inf,e=1)
activate.edges(net,onset=2,terminus=3,e=2)
activate.vertices(net, onset=1, terminus=Inf, v=1)
as.data.frame(net)
```

---

`as.network.networkDynamic` *Remove 'networkDynamic' class name from networkDynamic object.*

---

## Description

Changes the class of a `networkDynamic` object to a plain `network` object while leaving all attributes (including dynamic attributes) intact.

## Usage

```
 ## S3 method for class 'networkDynamic'
as.network(x, ...)
```

## Arguments

| | |
|---|---|
| `x` | A `networkDynamic` object that will have its class name stripped. |
| `...` | Possible additional arguments |

## Details

The primary use-case for this method is to force some other S3 method (like `simulate` in the `tergm` package, plot, or print) to use 'network' instead of 'networkDynamic' in method dispatching. Dynamic features data structures be left intact but not recognized by corresponding methods, so effectively broken.

## Value

Returns the original network with the `networkDynamic` class name removed but all other attributes unchanged

## Author(s)

Skye Bender-deMoll

## See Also

See Also as `network.extract` for extracting parts of a networkDynamic, and `network.collapse` for extracting part of a networkDynamic as a static network. `as.networkDynamic` for the inverse operation (adding the networkDynamic class to a static network).

## Examples

```
test<-network.initialize(2)
add.edges.active(test,tail=1,head=2,onset=1,terminus=2)
is.networkDynamic(test)
test<-as.network(test)
is.networkDynamic(test)
```

---

`as.networkDynamic` *as.networkDynamic*

---

## Description

The `as.networkDynamic` generic provides a very basic conversion to `networkDynamic` from other network types. It is generally recommended to use the `networkDynamic` function instead of `as.networkDynamic`, because `networkDynamic` provides a number of additional features for handling temporal information.

## Usage

```
## S3 method for class 'networkDynamic'
as.networkDynamic(object,...)

## S3 method for class 'network'
as.networkDynamic(object,...)

## S3 method for class 'networkLite'
as.networkDynamic(object,...)
```

## Arguments

| | |
|---|---|
| `object` | a `network`, `networkLite`, or `networkDynamic` object |
| `...` | ignored |

## Details

`as.networkDynamic.network` converts a `network` object into a `networkDynamic` object by adding a `networkDynamic` class.

`as.networkDynamic.networkLite` converts a `networkLite` object into a `networkDynamic` object by first converting it to a `network` object and then adding a `networkDynamic` class. (`networkLites` are a backend data structure used in some `EpiModel` and `statnet` packages, providing improved performance for certain applications, especially when working with vertex and edge attributes that can be stored as atomic vectors. Currently, `networkLites` come with the restriction that the network attributes `hyper`, `multiple`, and `loops` must be `FALSE`. See `networkLite-package` for more information.)

Such conversions between network types are used when starting a dynamic simulation from a cross-sectional network and returning the simulation history as a dynamic network, as done in the tergm package for example.

## Value

For as.networkDynamic.network the input object is returned with a networkDynamic class added. For as.networkDynamic.networkLite, the input object is converted to a network object and then the networkDynamic class is added. For as.networkDynamic.networkDynamic the input object is returned unchanged.

## Author(s)

Pavel, Zack W Almquist <almquist@uci.edu>

## See Also

For the inverse (removing the networkDynamic class) see as.network.networkDynamic and as.networkLite.network (which applies to networkDynamics). For extracting cross-sectional information from a networkDynamic (which is often more appropriate than simply removing the networkDynamic class), see network.collapse and network.extract. For more general construction of networkDynamics, see networkDynamic.

## Examples

```
nd <- as.networkDynamic(network.initialize(3))
class(nd)
is.networkDynamic(nd)

nwL <- networkLite::networkLite(3)
nwD <- as.networkDynamic(nwL)
class(nwD)
is.networkDynamic(nwD)
```

---

# attribute.activity.functions

*Functions to activate and query dynamic attributes on vertices, edges, and networks*

---

## Description

These functions set, query and modify attributes of networkDynamic objects used to represent values which change over time. These dynamic or "temporally extended attributes" are stored as standard attributes with a specially defined structure.

## Usage

```
activate.vertex.attribute(x, prefix, value, onset = NULL, terminus = NULL,
   length = NULL, at = NULL, v = seq_len(network.size(x)), dynamic.only = FALSE)

activate.edge.attribute(x, prefix, value, onset = NULL, terminus = NULL,
    length = NULL, at = NULL, e = seq_along(x$mel), dynamic.only = FALSE)

activate.edge.value(x, prefix, value, onset = NULL, terminus = NULL,
    length = NULL, at = NULL, e = seq_along(x$mel), dynamic.only = FALSE)

activate.network.attribute(x, prefix, value, onset = NULL, terminus = NULL,
    length = NULL, at = NULL, dynamic.only = FALSE)

## S3 method for class 'active'
get.vertex.attribute(x, prefix, onset = NULL, terminus = NULL,
          length = NULL, at = NULL, rule = c("any", "all","earliest","latest"),
          na.omit = FALSE, null.na = TRUE, active.default = TRUE,
          dynamic.only = FALSE, require.active = FALSE,
          return.tea = FALSE, unlist = TRUE, ...)

## S3 method for class 'active'
get.edge.attribute(x, prefix, onset = NULL, terminus = NULL,
          length = NULL, at = NULL, rule = c("any", "all","earliest","latest"),
          active.default = TRUE, dynamic.only = FALSE, require.active = FALSE,
          return.tea = FALSE, unlist = TRUE, ..., el)

## S3 method for class 'active'
get.edge.value(x, prefix, onset = NULL, terminus = NULL,
          length = NULL, at = NULL, rule = c("any", "all","earliest","latest"),
          active.default = TRUE, dynamic.only = FALSE, require.active = FALSE,
          return.tea = FALSE, unlist = TRUE, ...)

## S3 method for class 'active'
get.network.attribute(x, prefix, onset = NULL, terminus = NULL,length = NULL,
                at = NULL, rule = c("any", "all","earliest","latest"),
                dynamic.only = FALSE, return.tea=FALSE, unlist=TRUE, ...)

deactivate.vertex.attribute (x, prefix, onset=NULL, terminus=NULL,
       length = NULL, at = NULL, v = seq_len(network.size(x)), dynamic.only = FALSE)

deactivate.edge.attribute (x, prefix, onset = NULL, terminus = NULL,
          length = NULL, at = NULL, e = seq_along(x$mel), dynamic.only = FALSE)

deactivate.network.attribute(x, prefix, onset = NULL, terminus = NULL,
          length = NULL, at = NULL, dynamic.only = FALSE)

## S3 method for class 'active'
list.vertex.attributes(x, onset = NULL, terminus = NULL,length = NULL,
```

```
                at = NULL, na.omit = FALSE, rule = c("any", "all"),
                v = seq_len(network.size(x)), require.active = FALSE,
                active.default = TRUE, dynamic.only = FALSE, ...)

## S3 method for class 'active'
list.edge.attributes(x, onset = NULL, terminus = NULL,length = NULL,
                at = NULL, na.omit = FALSE, rule = c("any", "all"),
                e = seq_along(x$mel), require.active = FALSE,
                active.default = TRUE, dynamic.only = FALSE, ...)

## S3 method for class 'active'
list.network.attributes(x, onset = NULL, terminus = NULL,length = NULL,
                at = NULL, na.omit = FALSE, rule = c("any", "all"),
                dynamic.only = FALSE, ...)
```

## Arguments

| | |
|---|---|
| x | a networkDynamic object |
| prefix | character string for the name of the attribute to be stored (will be stored as prefix.active) |
| value | an object that is the value of the attribute to be stored. If value is a list, it is assumed that each element corresponds to an index in v. |
| onset | an optional vector of time-points that specifies the starts of the interval(s) at which the value argument should be considered active for each vertex/edge. |
| terminus | an optional vector of time-points that specifies the ends of the interval(s) at which the value argument should be considered active for each vertex/edge. |
| length | an optional vector of interval lengths for the interval(s) which should be activated for each vertex/edge. This must be accompanied by one of onset or terminus. |
| at | optional, one or more time points (corresponding to vertices/edges) at which values should be activated for each vertex/edge |
| v | optional, one or more IDs indicating which vertices should have the TEA attribute added (default is all vertices) |
| e | optional, one or more edge IDs indicating which edge should have the TEA attribute added. Default is all edges. Note: Currently values of e correspond to elements of x$mel, including null elements from deleted edges. So if some edges have been deleted from the network, e may need to be padded out appropriately and can be longer than network.edgecount(x). |
| dynamic.only | If dynamic.only==TRUE, only TEAs are considered; otherwise, non-TEA attributes are also employed (with activity as defined by active.default). |
| require.active | logical. If TRUE, NA will be returned instead of an attribute value if the associated vertex or edge is inactive for the query period. When vertex or edge activity has not be explicitly set, the state is controlled by active.default. |
| active.default | logical; should underlying vertices or edges without an activity attribute be regarded as active by default? |

| | |
|---|---|
| na.omit | logical; should values from missing vertices/edges be removed? (implemented for vertices, but removal means that the index may not correspond to vertex.id for returned values) |
| null.na | logical; should NULL values be replaced with NAs? |
| rule | a character string indicating how the attribute activity spells should be matched against the query spell for this query:<br>• any (default) match attribute values if they overlap with any part of the query interval.<br>• all match an attribute value only if it overlaps with the entire query interval,<br>• earliest match a single value corresponding to the earliest attribute overlaping any part of the query spell,<br>• latest match a single value corresponding to the earliest attribute overlaping any part of the query spell. |
| return.tea | logical; if return.tea=TRUE, will return an (appropriately truncated) activity attribute- a list with the first element a list of values and the second element the spell matrix of corresponding spell(s) matching according to rule. Also sets unlist=FALSE and dynamic.only=TRUE. |
| unlist | logical; should retrieved attributes be unlisted prior to being returned? Setting to FALSE is necessary to correctly fetch list objects stored as attributes. |
| ... | possible additional arguments |
| el | unused argument that exists solely for consistency with network; do not use this argument! |

## Details

These functions provide easy ways to work with temporally extended attributes, making it possible to store attributes on a dynamic network that should be represented as changing values and different points in time. Because the serve as wrappers for get.*.attribute and set.*.attribute, many of the arguments are directly passed through.

The activate.*.attribute methods act as a cross between activate.* set.*.attribute methods. They are used to activate an attribute for a given spell, and in so doing set a value for that spell. The corresponding deactivate methods are more straightforward, deactivating the attribute over a set interval (and removing any spells/values as needed).

A dynamic or Temporally Extended Attribute (TEA) is an edge, vertex, or network attribute satisfying the following properties:

1. Its name consists of an arbitrary prefix, together with the suffix ".active".
2. Its value consists of a two-element list, whose respective contents must be maintained in order as follows:
   - a. A list of value entries, such that the ith list element is the value of the attribute for the ith spell in the associated activity matrix. An attribute not active at any given point in time is defined as having a value of NA (but the reverse is not true – an active attribute can also take a value of NA).

- b. A two-column numeric matrix, conforming to the specifications for activity attributes in the networkDynamic package. Every active spell indicated in the activity matrix must correspond to exactly one entry of the associated value list, and these must be maintained in order (i.e., the value of the attribute for the ith active spell is the ith element in the value list).

There are several ways to specify the activity spell, and the general syntax rules are described at `activity.attribute`. Activity can be set for a single time point, using either `at`, or by setting onset=terminus. Activity can be set for an interval [onset,terminus), using a valid combination of the `onset`, `terminus` and `length` attributes.

This allows for a wide range of options when specifying spells, but a correct specification must use only one of these forms:

```
at
onset and terminus
onset and length
terminus and length
```

If the `set` and `deactivate` functions are provided with no timing information, the element is activated/deactivated from `-Inf` to `Inf`. The specified interval spans the period from the onset (inclusive) to the terminus (exclusive), so `[onset,terminus)`. For more details, see `activate` The `get` query functions will give an error if no timing information is provided.

When the activity arguments have a length greater than 1, the elements are assumed to correspond to the vertex or edge ids that should have that activity specified, not to the elements of `value`. In other words, it is possible to use one function call to activate multiple values on multiple vertices with a different activity time on each vertex, but it is not possible to activate multiple values at multiple times on a single vertex with one call.

`activate.edge.value` is a wrapper for `activate.edge.attribute`.

When the attribute query has a non-zero duration (i.e. `length` > 0, or `onset!=terminus`) it is possible that the query spell will match against multiple attribute values for each network element. If multiple values match and `return.tea=FALSE`, a warning will be generated and only the earliest value will be returned. To avoid the warning, this behavior (or its opposite) can be specified by setting `rule='earliest'` or `rule='latest'`.

## Value

The set methods modify their network argument internally and also invisibly return a reference to the modified network. The get methods return the attribute object appropriate for the time range specified query spell parameters. If query results in multiple values and `return.tea=FALSE` only the first (earliest) is returned along with a warning. Note that `get.edge.value.active` returns `NULL` if no attributes names match, where `get.vertex.attribute.active` returns `NA`s. Also of `null.na=FALSE` and `unlist=TRUE`, the index position for returned values may not correspond to the vertex.id because NULL values will be omited.

## Note

Attribute activity is only checked against vertex and edge activity during fetching if `require.active=TRUE`. Otherwise the attributes for a vertex can take values (be active) when the vertex itself is inactive. Likewise for edges. The because they must check all attributes on all network elements,

the `list.*.active` functions are generally more computationally expensive that a `get` call for checking if a specific attribute exists. Note that `get.edge.attribute.active` currently calls `get.edge.attribute.active` and does not support a list of edges as input.

## Author(s)

skyebend, ayn, carter, pavel, martina, steve

## See Also

See Also as `set.vertex.attribute`, `get.vertex.attribute`,`list.vertex.attributes`,`activate.vertices`, `activity.attribute`, `activate`

## Examples

```
#initialize network
test<-network.initialize(5)

#activate vertex attribute
test<-activate.vertex.attribute(test,"letter","a",onset=0,terminus=1)
test<-activate.vertex.attribute(test,"number","4",onset=1,terminus=2)
test<-activate.vertex.attribute(test,"number","5",onset=2,terminus=3)

#list active/all vertex attributes
list.vertex.attributes.active(test, onset=0,terminus=3,dynamic.only=TRUE)
list.vertex.attributes.active(test, onset=1,terminus=3,dynamic.only=FALSE)

#get values for specific vertex attribute
get.vertex.attribute.active(test,"letter",onset=2,terminus=3)

#deactive vertex attribute
test <- deactivate.vertex.attribute(test, "letter", onset=0, terminus=3)
list.vertex.attributes.active(test, onset=0,terminus=3,dynamic.only=TRUE)

#initialize edges
test[1,2]<-1
test[2,3]<-1

#activate edge attribute
test<-activate.edge.attribute(test,"number",1,onset=0,terminus=2)
test<-activate.edge.attribute(test,"number",5,onset=2,terminus=5)
test<-activate.edge.attribute(test,"letter","a",onset=1,terminus=4)

#list edge attributes
list.edge.attributes.active(test, onset=0,terminus=4,dynamic.only=TRUE)
list.edge.attributes.active(test, onset=0,terminus=4,dynamic.only=FALSE)

#get values for specific edge attribute
get.edge.value.active(test,"number",onset=3,terminus=4)

#deactive edge attribute
test <- deactivate.edge.attribute(test, "letter", onset=0, terminus=3)
```

```
list.edge.attributes.active(test, onset=0,terminus=3,dynamic.only=TRUE)

#activate network attribute
test <- activate.network.attribute(test,"alist",list("a","b"),onset=1,terminus=2)
test <- activate.network.attribute(test,"alist",list("c","d"),onset=2,terminus=3)
test <- activate.network.attribute(test,"aspace",list("1","2"),onset=1,terminus=2)
test <- activate.network.attribute(test,"aspace",list("3","4"),onset=2,terminus=3)

#list network attributes
list.network.attributes.active(test, onset=0,terminus=3,dynamic.only=TRUE)
list.network.attributes.active(test, onset=0,terminus=3,dynamic.only=FALSE)

#get values for specific network attribute
get.network.attribute.active(test,"alist",at=2.5,unlist=FALSE)

#deactive network attribute
test <- deactivate.network.attribute(test, "alist", onset=0, terminus=3)
list.network.attributes.active(test, onset=0,terminus=3,dynamic.only=TRUE)

# activate multiple values on multiple vertices at multiple times
test<-network.initialize(3)
activate.vertex.attribute(test,"letters",c("a","b","c"),onset=c(0,1,2),terminus=c(1,2,3))

# peek at TEA structure using non-TEA attribute query
get.vertex.attribute(test,"letters.active",unlist=FALSE)

# compare different 'rules'
test<-network.initialize(1)
activate.vertex.attribute(test,'color','red',onset=0,terminus=1)
activate.vertex.attribute(test,'color','green',onset=1,terminus=2)
activate.vertex.attribute(test,'color','blue',onset=2,terminus=3)

get.vertex.attribute.active(test,'color',onset=0,terminus=3)
get.vertex.attribute.active(test,'color',onset=0,terminus=3,return.tea=TRUE)
get.vertex.attribute.active(test,'color',onset=0,terminus=3,rule='earliest')
get.vertex.attribute.active(test,'color',onset=0,terminus=3,rule='latest')
```

---

`cls33_10_16_96` *Daniel McFarland's Streaming Classroom Interactions Dataset*

---

## Description

A dynamic network object describing continuous time streams of social interactions between teachers and students in a classroom observed by Daniel McFarland in 1996.

## Usage

```
data(McFarland_cls33_10_16_96)
```

## Format

The format is is a networkDynamic object named `cls33_10_16_96` with node and edge activity.attributes defined by spell matricies and associated vertex attributes and dynamic edge attributes.

## Details

The dynamic network object has 20 vertices and directed edges and covers a time range of 0.0 to 49.0 minutes of class time. The vertex attribute `gender` gives gender ('M' or 'F') of each person and `type` gives a crude role ('grade11','grade12','instructor'). The original dataset id for the vertices is included as `data_id` and defined as a persistent.id. The edges include a dynamic attribute (TEA) `interaction_type` with values 'saction', 'social' and 'task' (see explanation below) and a `weight` assigned by the reacher to that interaction (direct comunication is given more weight than indirect 'broadcast' communication).

A .son formatted input file with the same data as well as seperate tsv files for vertices and edges are located in the /inst/extdata directory.

The data for these classroom interactions consists of streaming observations of conversation turns. The conversation turns were recorded as pairs of senders and receivers and for types of content. Speakers were viewed as directing their communication in one of two fashions: (1) indirect soundings, such as lectures (where a teacher addresses all students); and (2) direct interactions that are focused on particular others. Each type of directional speech is viewed as having different forms of network reception - indirect speech reaches all bystanders as passive hearers and directed speech reaches the focal person as an active co-author of the conversation (Goffman 1981).

Two types of interaction are found to prevail in classroom contexts: task and sociable (McFarland, in press). Task interactions are those behaviors that pertain to the ongoing teacher prescribed task (content is academic). In contrast, sociable interactions concern everyday concerns of adolescents' social lives, such as parties, dating, social outings, plans, etc. While the content is the key distinction, it is often the case that these speech acts are distinguishable in style as well, where sociable behaviors are more play-like, fast-paced, and free than the more constrained academic styles of speech during lessons (Cazden 1988). Last, observations also recorded when task and sociable forms of interaction were laminated with evaluative meaning. Such evaluations were seen as being either positive or negative - either giving praise or attempting a reprimand (Ridgeway and Johnson).

The class (#33) is an economics class composed of 11th and 12th graders at a magnet high school. On this day, economics has two teachers. The first is the usual teacher and the second is a businessman who donates his time a couple days a month to assist the class with their lesson on stock investments. After a minute of undefined class time, the two teachers prescribe collaborative group work and assist students in conducting it. The students are assigned groups within which they are to study the stock market and make mock investments they can follow over time. The groups compete with each other over the course of the semester to see who can make the greatest profit.

The network narrative for the class opens with the teacher trying to focus student attention and then lecturing while the visiting professional assists each group. The teacher prescribes group work but continues to lecture, thereby preventing the desired social routine from fully taking effect. Eventually the students are doing group projects while the adults move from group to group, facilitating their progress, and stabilizing interaction patterns. The routine of group work is basically characterized by dyadic task and social interactions that persist in multiple clusters. Not all persons engage in these groups, and a couple students sit quietly by themselves. The group work routine breaks down as social activity increases within the groups', and then the teacher emits broadcast sanctions

in an effort to redirect student attention back on task (16 minutes). The task breaks down again at the end of class, but this time because the adults make closing announcements.

So the network illustrates that teachers are involved in this task engaging their students as they monitor interaction. When students become too social, a teacher usually arrives, disperses the group, and then reforms it via task interactions (revolution in type of behavior, McFarland 2004). Hence, the "dance" here entails relatively bounded groups of individuals that free-associate over tasks and drift into social affairs, and teachers who refocus affairs by indirect means of broadcasts or by direct means of directed speech.

## Source

Dr. Daniel McFarland, Stanford University https://ed.stanford.edu/faculty/mcfarland

## References

McFarland, Daniel A. 2005. "Why Work When You Can Play? Dynamics of Formal and Informal Organization in Classrooms." Chapter 8 in The Social Organization of Schooling, edited by Larry Hedges and Barbara Schneider (pp. 147-174). New York: Russell Sage Foundation.

McFarland, Daniel A. 2004. "Resistance as a Social Drama - A Study of Change-Oriented Encounters." American Journal of Sociology 109 (6): 1249-1318.

McFarland, Daniel A. 2001. "Student Resistance: How the Formal and Informal Organization of Classrooms Facilitate Everyday Forms of Student Defiance." American Journal of Sociology 107 (3): 612-78.

McFarland, Daniel A. and Skye Bender-deMoll. 2003. "Classroom Structuration: A Study of Network Stabilization." Working paper, Stanford University.

## Examples

```
data(McFarland_cls33_10_16_96)
```

---

# delete.activity *Remove Timing Information in a Dynamically Extended Network Object*

---

## Description

Removes the activity spells of a given set of vertices or edges.

## Usage

```
delete.edge.activity(x, e=seq_along(x$mel))

delete.vertex.activity(x, v=seq_len(network.size(x)))
```

## Arguments

| | |
|---|---|
| x | an object, of class network or networkDynamic. |
| e | the edges of x that will have their timing information deleted; default = all. |
| v | the vertices of x that will have their timing information deleted; default = all. |

## Details

Though the timing information of the edges and/or vertices may be removed, other networkDynamic methods will assume activity or inactivity across all time points, based on the argument active.default.

## Value

An object with the same class as x, equivalent to x without the activity spells of the edges e or the vertices v.

## Author(s)

Ayn Leslie-Cook <aynlc3@uw.edu>, Carter T. Butts <buttsc@uci.edu>

## See Also

activate

## Examples

```
library(networkDynamic)
data(flo)
net1 <- network(flo)
activate.edges(net1)
activate.vertices(net1)
net2 <- net1
delete.edge.activity(net1, e=seq(2,40,2))
delete.edge.activity(net2)
delete.vertex.activity(net2)
is.active(net1, at=0, e=c(1,2), active.default=FALSE)
is.active(net1, at=0, e=c(1,2), active.default=TRUE)
is.active(net2, at=0, e=1:16, active.default=FALSE)
is.active(net2, at=0, e=1:16, active.default=TRUE)
```

---

# duration.matrix *Construct a edge spells list from base network and toggle list.*

---

## Description

Given a network object and a toggle list, it will construct a list of edges and activity spells and return that as a data frame. The network object does not need to have edges; it only needs to have enough vertices to cover all the edges specified in the toggle list.

The toggle list should be a three column matrix or data frame, where the first column is the toggle time, the second column is the tail vertex id of the edge, and the third column is the head vertex id of the edge. A toggle represents a switch from active state to inactive, or vice-versa.

## Usage

```
duration.matrix(nw0, changes, start, end)
```

## Arguments

| | |
|---|---|
| nw0 | an object of class network |
| changes | a matrix or data frame, listing the toggles for edges as specified in the above description |
| start | integer, specifies the starting time of the network's dynamic activity |
| end | integer, specifies the ending time of the network's dynamic activity |

## Value

Returns a data frame that lists the edge spells specified by the toggles. The columns in the data frame are start end tail head left.censored right.censored duration. Left and right censoring and duration are calculated automatically from the toggle list and the start and end times.

## Author(s)

Ayn Leslie-Cook <aynlc3@uw.edu>, Carter T. Butts <buttsc@uci.edu>

## See Also

as.networkDynamic

## Examples

```
library(networkDynamic)
# duration matrix test
net <-network.initialize(3)
net[1,2]<-1;
net[2,3]<-1;
net[1,3]<-1;
class(net)
# toggle list: time, tail, head
tog=matrix(c(1,1,2, 1,2,3, 2,1,2, 4,1,3, 4,1,2), ncol=3, byrow=TRUE)
networkDynamic:::duration.matrix(net, tog, 0, 5)
```

`get.dyads.active` *Return the set of vertex ids of edges active at a given time*

## Description

Constructs a matrix, essentially an edgelist, of the pairs of vertices connected by edges active within the query spell.

## Usage

```
get.dyads.active(nD, onset = NULL, terminus = NULL, length = NULL, at = NULL,
             rule = c("any", "all", "earliest", "latest"), active.default = TRUE)
```

## Arguments

| | |
|---|---|
| `nD` | networkDynamic object to be queried |
| `onset` | optional numeric value giving onset time of query spell |
| `terminus` | optional numeric value giving terminus time of query spell |
| `length` | optional numeric value giving length (duration) of query spell |
| `at` | optional numeric value giving single time point to query |
| `rule` | a character string indicating the definition of "active" for this query: `any` (default) to define active as any active spell during the interval, or `all` to define active as being active over the entire interval. For this functions, the values of `earliest` and `latest` are equivalent to `any` and are only included for consistency. |
| `active.default` | logical; should elements without an activity attribute be regarded as active by default? Default to `TRUE` |

## Details

Uses `is.active` internally. The ordering of matrix rows is arbitrary. The ordering of vertices in non-directed networks corresponds to how they were set in the data structure (i.e. does not enforce i > j) If no edges are active or exist, a matrix with zero rows will be returned.

## Value

Returns a two-column numeric matrix in which the first column gives a tail vertex.id and the second column gives the head vertex.id and each row corresponds to the existence of an active relationship between the vertices for the given query spell.

## Note

This function does not support hypergraphic networks.

## Author(s)

skyebend

## See Also

See also is.active

## Examples

```
data(windsurfers)
get.dyads.active(windsurfers,onset=2,terminus=3)
```

---

`get.networks` *Obtain a list of collapsed networks sampled periodically from a networkDynamic object*

---

## Description

Given a start time, end time, and increment (or vectors of onsets and termini) to express sampling intervals, return a list of collapsed networks from a networkDynamic object.

## Usage

```
get.networks(dnet, start = NULL, end = NULL, time.increment = NULL,
             onsets=NULL, termini=NULL,...)
```

## Arguments

| | |
|---|---|
| `dnet` | A `networkDynamic` object with possible vertex, edge, network, and attribute spell information. |
| `start` | numeric value giving the start of the sampling interval |
| `end` | numeric value giving the end of the sampling interval |
| `time.increment` | value for the offset (and duration) between sucessive samples. Will default to 1 if not otherwise specified |
| `onsets` | A numeric vector containing the onset times of the networks to be extracted. This must be accompanied by `termini` of the same length. |
| `termini` | A numeric vector containing the terminus times of the networks to be extracted. This must be accompanied by `onsets` of the same length. |
| `...` | Additional arguments to `network.collapse` (such as `rule`,`active.default`,`retain.all.vertices`) |

## Details

The sampling ("slicing") intervals may be defined using either the `start`, `end`, and `time.increment` parameters, or by providing parallel vectors of `onsets` and `termini`. If values are not specefied but a `net.obs.period` attribute exists to describe the 'natural' sampling parameters, `start` and `end` will defult to the max an min of the observations element, with `time.increment` set to its corresponding value in the `net.obs.period`.

## Value

A `list` of static `network` objects corresponding to the specified sampling intervals of the `networkDynamic`

## Note

See Note in `network.collapse`.

## Author(s)

Kirk Li, Skye Bender-deMoll

## See Also

See Also as `network.collapse` for obtaining a slice of static network, `network.extract` for extracting sub-ranges of a networkDynamic, `get.vertex.attribute.active` for more on TEA attributes, and `as.network.networkDynamic` for stripping the the networkDynamic class from an object.

## Examples

```
# create a networkDynamic with some basic activity and time extended attributes (TEA)
test <- network.initialize(5)
add.edges.active(test, tail=c(1,2,3), head=c(2,3,4),onset=0,terminus=1)
activate.edges(test,onset=3,terminus=5)
activate.edges(test,onset=-2,terminus=-1)
activate.edge.attribute(test,'weight',5,onset=3,terminus=4)
activate.edge.attribute(test,'weight',3,onset=4,terminus=5,e=1:2)

# obtain the list of networks
list <- get.networks(test,start=0, end=5)

# aggregate over a longer time period with specified rule
list <- get.networks(test,start=0, end=6,time.increment=2,rule='latest')

# use 'at' style extraction of momentary slices via onsets & termini
list <- get.networks(test,onsets=0:5,termini=0:5)

# ensure that all networks returned will be the same size
list <- get.networks(test,onsets=0:5,termini=0:5,retain.all.vertices=TRUE)


# find out how many edges in each sampling point with apply
sapply(get.networks(test,start=0,end=5),network.edgecount)

# generate a list of matrices
lapply(get.networks(test,start=0,end=5),as.matrix)
```

`get.timing.info` *Retrieve the timing information present in a networkDynamic.*

## Description

These methods return either the activity spells of each vertex or edge or the unique time points at which edges and vertices become active or inactive.

Finds all of the unique time points at which the edges and/or vertices become active or inactive.

## Usage

```
get.change.times(x, vertex.activity=TRUE, edge.activity=TRUE, ignore.inf=TRUE,
             vertex.attribute.activity = TRUE, edge.attribute.activity = TRUE,
             network.attribute.activity = TRUE)
get.vertex.activity(x, v=seq_len(network.size(x)),
                    as.spellList=FALSE, active.default=TRUE)
get.edge.activity(x, e=seq_along(x$mel), as.spellList=FALSE, active.default=TRUE)
```

## Arguments

| | |
|---|---|
| `x` | an object, of class `network` or `networkDynamic`. |
| `vertex.activity` | logical, whether the times at which vertices become active or inactive is to be returned. |
| `edge.activity` | logical, whether the times at which edges become active or inactive is to be returned. |
| `vertex.attribute.activity` | logical, whether the times attached to any dynamic vertex attributes should be returned |
| `edge.attribute.activity` | logical, whether the times attached to any dynamic edge attributes should be returned |
| `network.attribute.activity` | logical, whether the times attached to any dynamic network attributes should be returned |
| `ignore.inf` | logical, whether Inf value will be ignored or not. |
| `e` | IDs for the edges whose activity spells are wanted. |
| `v` | IDs for the vertices whose activity spells are wanted. |
| `as.spellList` | should data be returned in spell matrix form? TODO: should this be callsed as.spellmatrix instead? |
| `active.default` | logical, should edges / vertices with no spell activity defined be treated as always active (`TRUE`) and included in output, or never active (`FALSE`) and skipped. |

## Value

For the get.change.times function, the default settings return a vector of all of the finite time points at which any edge or any vertex becomes active or inactive, or any of their dynamic attributes change. If only one of the logical arguments, say vertex.activity, is set to FALSE, then the time points for the corresponding parts of the data structure will not be returned. If both are FALSE, numeric(0) is returned.

For the get.vertex.activity and get.edge.activity functions, a list of activity spells is returned with each list element corresponding to the id of the vertex or edge. Deleted edges will appear as NULL entries, as will any spell marked as inactive with the 'null' (Inf,Inf) spell. If the argument as.spellList=TRUE the activity spells are instead formatted as a data.frame and include additional columns to indicate possible onset- and terminus-censoring. Any 'null' spells and deleted edges will be omited.

For get.vertex.activity the columns are [onset,terminus,vertex.id,onset.censored,terminus.censored,durat

For get.edge.activity the columns are [onset, terminus, tail, head, onset.censored, terminus.censored, duration, edge.id] .

The data.frames will be sorted by id, onset, terminus.

If a net.obs.period network attribute is set, the range of its $observations element is substituted in for the -Inf and Inf values that otherwise indicate the onset- and terminus-censored observations.

## Author(s)

Ayn Leslie-Cook <aynlc3@uw.edu>, Carter T. Butts <buttsc@uci.edu>,skyebend

## See Also

See also activity.attribute,net.obs.period,as.data.frame.networkDynamic

## Examples

```
library(networkDynamic)
data(flo)
net1 <- network(flo)
activate.edges(net1, onset=1:20, terminus=101:120)
activate.vertices(net1, at=seq(2,32,2))
get.change.times(net1)
get.edge.activity(net1)
get.edge.activity(net1,as.spellList=TRUE)
get.vertex.activity(net1)

a <-network.initialize(5)
activate.vertices(a,onset=0,terminus=Inf)
get.change.times(a,ignore.inf=FALSE)
```

`is.active` *Determine Activation State of Edges and Vertices*

## Description

Query the activity state of one or more network elements (edges or vertices) for either a single time point or an extended interval.

## Usage

```
is.active(x, onset = NULL, terminus = NULL, length = NULL, at = NULL, e = NULL, v = NULL,
    rule = c("any", "all","earliest","latest"), active.default = TRUE)
```

## Arguments

| | |
|---|---|
| `x` | an object of class `network`. |
| `onset` | an optional vector of timepoints that specifies the start of the queried interval(s). This must be accompanied by one of `terminus` or `length`. |
| `terminus` | an optional vector of timepoints that specifies the end of the queried interval(s). This must be accompanied by one of `onset` or `length`. |
| `length` | an optional vector of interval lengths for the queried interval(s). This must be accompanied by one of `onset` or `terminus`. |
| `at` | an optional vector of single time point(s) to query. |
| `e` | a vector of edge IDs to evaluate. |
| `v` | a vector of vertex IDs to evaluate. |
| `rule` | a character string indicating the definition of "active" for this query: `any` (default) to define active as any active spell during the interval, or `all` to define active as being active over the entire interval. Because we are only testing for activity, the values of `earliest` and `latest` are equivilent to `any` and are only included for compatibility with calling functions. |
| `active.default` | logical; should elements without an activity attribute be regarded as active by default? |

## Details

This function can be used to query the activity status of an element at single time point, or over an interval. The specification of timepoints and intervals is the same as for the `activate` function. The general rules are discussed in `activity.attribute`.

To query status at a single time point, use one of the following:

- the `at` argument, or
- set `onset=terminus`.

To query status over a particular interval, one of the following valid combinations of `onset`, `terminus` and `length` is required:

- onset and terminus.
- onset and length.
- terminus and length.
- or, you can specify no information.

If the query is specified with no timing information the spell is defined as (-Inf,Inf). The query interval spans the period from the onset (inclusive) to the terminus (exclusive), so [onset,terminus).

There are some special behaviors associated with the arguments Inf and -Inf.

- The open-ended interval c(-Inf,x) includes -Inf. For consistency, we also allow the open-ended interval c(x, Inf) to include Inf. Thus [onset, terminus) will be interpreted as [onset, terminus] when terminus = Inf.
- Queries for intervals specified by c(Inf, Inf) or c(-Inf, -Inf) are ignored.

If the e argument includes edge ids corresponding to deleted edges, a warning will be generated because the length of the output vector will not match the vector of edge ids. In this case it is a good idea to use valid.eids to determine the edge ids.

## Value

A logical vector indicating the activity states of vertices or edges. In the case of vertices, the elements of the vector correspond to the vertex ids provided via the v paramter. In the edges case, if the network has deleted edges, they will be omited from the result so the elements of the vector may not correspond to the eids provided via the e parameter.

## Author(s)

Ayn Leslie-Cook <aynlc3@uw.edu>, Carter T. Butts <buttsc@uci.edu>

## See Also

activity.attribute, activate, valid.eids

## Examples

```
triangle <- network.initialize(3)  # create a toy network
add.edge(triangle,1,2)    # add an edge between vertices 1 and 2
add.edge(triangle,2,3)    # add more edges
add.edge(triangle,3,1)

# turn on all edges at time 1 only (0 length spell)
activate.edges(triangle,at=1)
# activate edge (1,2) from t=2 to t=3
activate.edges(triangle,onset=2, terminus=3,
               e=get.edgeIDs(triangle,v=1,alter=2))
# activate edge (2,3) from t=4 for 2 time lengths
activate.edges(triangle,onset=4, length=2,
               e=get.edgeIDs(triangle,v=2,alter=3))

# are the edges active at a particular time?
```

```
is.active(triangle, at=1, e=1:3)
is.active(triangle, at=4, e=seq_along(triangle$mel))

# delete an edge
delete.edges(triangle,e=2)
is.active(triangle, at=4, e=seq_along(triangle$mel))
# gives warning, and only two values returned, which edges are they?
# much safer to use
is.active(triangle, at=4, e=valid.eids(triangle))
```

---

nd_test_nets *networkDynamic testing networks*

---

## Description

A list of networkDynamic objects to be used for testing package functions. Each network exemplifies a possible extreme case for the data structure representation.

## Usage

```
data(nd_test_nets)
```

## Format

A list of networkDynamic objects. Each element is named with a short description

## Details

This list of networks is intended for testing purposes. Each element is a network with a different unusual configuration of the networkDynamic data structure. The idea is that the items on the list can be used when testing functions that use networkDynamic objects to make sure that they can properly handle (or explicitly reject) a wide range of network configurations and parameter settings. A short description of each test case can be printed with names(nd_test_nets). Note that these test cases are intended for checking that functions process the data structures without error, but in most cases the networks are too trivial to be useful in testing algorithm results.

## Examples

```
data(nd_test_nets)
# print the list of test cases
names(nd_test_nets)
# check that network.size.active works
# (or at least doesn't fail)
# for many types of networks
lapply(nd_test_nets,network.size.active,at=1)
```

net.obs.period *Network attribute describing the observation properties of a networkDynamic object*

## Description

net.obs.period is an optional network-level attribute that gives meta-information about the characteristics of the observations, time range and temporal model of a networkDynamic object.

## Format

If the attribute is present, it is required to have a specific structure. It is a 4-element list with named values.

- the observations element is a list of two-elment spell vectors that give the onset and terminus of the time period(s) during which the network was observed or simulated
- mode A string indicating the temporal model of the network: either discrete or continuous
- time.increment is an indicator of the 'natural' time unit for iterating over the network (primarily useful for discrete time models)
- time.unit is the name of the unit used by all of the spells in the object: often set to step for discrete simulations.

## Details

Some functions that work with networkDynamic objects need to know information about the observation design of the network. Is it discrete or continuous time model? When did the observation (or simulation) of the network processes start and end? What are the time units? This information may be stored in the *optional* net.obs.period network attribute.

Some examples of possible net.obs.period values for various networks:

- for an stergm output it might be: list(observations=list(c(0,100)),mode="discrete", time.increment=1,time.unit="step")
- For Lin's windsurfers , which is missing all observations on day 25: list(observations=list(c(0,25),c(26,31)),
  time.increment=1,time.unit="day")
- For McFarland's classroom data: list(observations=list(c(0,55),),mode="continuous", time.increment=NA,time.unit="minute")
- A fictitious instantaneous panel example : list(observations=list(c(0,0),c(21,21),c(56.5,56.5),c(68,68))
  time.increment=0,time.unit="seconds")

  The values of the net.obs.period object are not checked for consistency with activity spells of the network itself.

## References

Link to page on statnet wiki?

## See Also

Several of the networkDynamic converter functions set the net.obs.period attribute on their output object. See also activity.attribute

## Examples

```
nd<-network.initialize(5)
activate.vertices(nd,onset=0,terminus=100)
set.network.attribute(nd,'net.obs.period',list(observations=list(c(0,100)),
                      mode="discrete", time.increment=1,time.unit="step"))
# find the overall `expected' time range for the object
range(get.network.attribute(nd,'net.obs.period')$observations)
```

---

network.collapse *Convert a time range of a networkDynamic object into a static network object.*

---

## Description

This function provides an easy way to sensibly collapse the time-related information in a networkDynamic object and return a plain network object with a set of vertices, edges, and attributes that appropriately correspond to those appearing in the original networkDynamic object during the query interval.

## Usage

```
network.collapse(dnet, onset = NULL, terminus = NULL, at = NULL, length = NULL,
               rule = c("any", "all","earliest","latest"), active.default = TRUE,
                  retain.all.vertices=FALSE, rm.time.info=TRUE, ...)
dnet%k%at
```

## Arguments

| | |
|---|---|
| dnet | A networkDynamic object with possible vertex, edge, network, and attribute spell information. |
| onset | optionally, the start of the specified interval. This must be accompanied by one of terminus or length. |
| terminus | optionally, the end of the specified interval. This must be accompanied by one of onset or length. |
| length | optionally, the length of the specified interval. This must be accompanied by one of onset or terminus. |
| at | optionally, a single time point. |
| rule | a text string for defining "active" for this call: any if elements active at any time during the interval are to be used, or all if elements must be active over the entire interval. The value earliest behaves like any but specifies that when multiple attribute values are encountered, only the earliest will be returned. The |

value latest behaves like any but specifies that when multiple attribute values are encountered, only the latest will be returned.

active.default — logical; should elements without an activity attribute be regarded as active by default?

retain.all.vertices — logical; should the extracted network retain all vertices, ignoring the vertex activity spells of x in order to ensure that the network returned has the same size as dnet?

rm.time.info — logical; if TRUE, the net.obs.period attribute will be removed (if it exists), and the activity summary attributes activity.count and activity.duration will not be attached to edges and vertices

... — Possible additional arguments (not yet invented) to handle aggregation of attributes.

## Details

First performs a network.extract on the passed networkDynamic object with the specified time range to get the appropriate set of active edges. Extracts appropriate values from any dynamic attributes and stores them as static attributes and optionally (if rm.time.info=FALSE) computes some crude summary attributes for edge and vertex spells (activity.count, activity.duration),. Then removes all activity.attribute and dynamic.attributes information and returns a plain network without the networkDynamic class.

The %k% operator ('K' for kollapse) is a shortcut for the 'at' version of network.collapse.

If no temporal arguments are specified, defaults to collapsing entire time range of the input network (onset=-Inf,terminus=Inf). Original network is not modified.

Arbitrary attribute values may be returned when query spells with a duration are used for continuous time networks (i.e. anything other than 'at') or query spells that don't line up with the discrete units for discrete time networks are used. If a query spell intersects with multiple attribute values, a warning will be produced and only the earliest value will be used (see get.vertex.attribute.active). To avoid ambiguity (and the warning), appropriate handling can be specified by setting rule='earliest' or rule='latest' to indicate which of the multiple values should be returned.

The duration values returned for edges do not account for any potential 'censoring' of observations in the original network.

## Value

A new 'static' network object corresponding to the specified time range of the networkDynamic argument. If the original network contained dynamic TEA attributes (i.e. 'weight.active'), the appropriate (if possible) value is queried and stored as a non-TEA attribute ('weight').

## Note

This function may be quite computationally expensive if the network contains lots of attributes. For many tasks it is possible to avoid collapsing the network by using is.active, the dynamic network.extensions, and the attribute.activity.functions.

## Author(s)

Skye Bender-deMoll

## See Also

See also network.extract for extracting sub-ranges of a networkDynamic, get.vertex.attribute.active for more on TEA attributes, and as.network.networkDynamic for stripping the the networkDynamic class from an object without the expense of modifying or removing the activity attributes.

## Examples

```
# create a network with some basic activity
test<-network.initialize(5)
add.edges.active(test, tail=c(1,2,3), head=c(2,3,4),onset=0,terminus=1)
activate.edges(test,onset=3,terminus=5)
activate.edges(test,onset=-2,terminus=-1)

# collapse the whole thing
net <-network.collapse(test)
is.networkDynamic(net)
get.vertex.attribute(net,'activity.duration')
get.edge.value(net,'activity.count')
get.edge.value(net,'activity.duration')

# add a dynamic edge attribute
activate.edge.attribute(test,'weight',5,onset=3,terminus=4)
activate.edge.attribute(test,'weight',3,onset=4,terminus=5)

# collapse with an interval query
net3<-network.collapse(test,onset=3,terminus=4)
get.edge.value(net3,'weight')

# note that if we use a query that intersects mutiple
# attribute values it will generate a warning.
# try commented line below:

# net3<-network.collapse(test,onset=3,terminus=5)

# but should be safe from attribute issues when
# collapsing with a point query
net3<-network.collapse(test,at=3)
get.edge.value(net3,'weight')

# can use operator version for point query instead
net3<-test%k%4.5
get.edge.value(net3,'weight')
```

`network.dynamic.check` *Verify a Dynamically Extended Network Object*

## Description

Checks a networkDynamic object for any activity information that is not correctly specified.

## Usage

```
network.dynamic.check(x, verbose = TRUE, complete = TRUE)
is.networkDynamic(x)
```

## Arguments

| | |
|---|---|
| `x` | an object, hopefully of class `networkDynamic`. |
| `verbose` | logical; should output messages be printed to the user? |
| `complete` | logical; should a complete test be ran, including checks of the activity matrices for edges and vertices, versus a check of only edge activity invovling non-active vertices |

## Details

Currently, `network.dynamic.check` runs three types of checks on an input object:

1. If a complete test is run, vertex spell matrices and edge spell matrices (if any) are checked for legality, specifically these are checked for correct dimensionality and proper spell ordering.
2. If vertex and edge spell matrices are correctly specified or assumed to be using `complete=FALSE`, incident vertices of active edges (if any) are checked to ensure that they are also active during the specified period(s).
3. If a complete test is run, and any TEA attributes are attached to the network, vertices or edges, they will be checked for correct structure.

If spell matrices are only acted upon by network or networkDynamic methods, there is no need to run a complete test - correct dimensionality and spell ordering are guaranteed.

`is.networkDynamic` checks if its argument has the class 'networkDynamic'.

## Value

For `network.dynamic.check`: A list of logical vectors, one for each network component checked:

- `$vertex.checks` : Whether the spell matrix of each vertex (if any) is correctly specified
- `$edge.checks` : Whether the spell matrix of each edge (if any) is correctly specified
- `$dyad.checks` : Corresponding to edges, whether the incident vertices of each edge are active when the edge is active

- $vertex.tea.checks : Corresponding to vertices, whether the vertex has correctly formed TEA attributes
- $edge.tea.checks : Corresponding to edges, whether the edge has correctly formed TEA attributes
- $network.tea.checks : Single boolean, whether the network TEA attributes are formed correctly
- $net.obs.period.check : NULL, if no net.obs.period network attribute is present, otherwise a single boolean, whether the attribute is formed correctly.

For is.networkDynamic: a boolean which is true if the class matches

## Author(s)

Ayn Leslie-Cook <aynlc3@uw.edu>, Carter T. Butts <buttsc@uci.edu>, Skye Bender-deMoll

## See Also

activity.attribute, net.obs.period

## Examples

```
test <-network.initialize(2) # make a network
# only activate one vertex
activate.vertices(test,onset=2,length=5,v=1)
test[1,2] <-1 # add an edge between the verticies
# activate the edge for a spell not including vertex spell
activate.edges(test,onset=0,terminus=7, e=1)
network.dynamic.check(test) # find the bad edge

# name a vertex attrible like a TEA but with the wrong structure
set.vertex.attribute(test,'letters','a')
network.dynamic.check(test)
```

---

# network.extensions *Dynamically Extended Core Network Functions*

---

## Description

Various core functions from the network package, with specialized extensions for handling dynamic data.

## Usage

```
get.edgeIDs.active(x, v, onset = NULL, terminus = NULL, length = NULL, at = NULL,
    alter = NULL, neighborhood = c("out", "in", "combined"),
   rule = c("any", "all", "earliest", "latest"), na.omit = TRUE, active.default = TRUE)
get.edges.active(x, v, onset = NULL, terminus = NULL, length = NULL, at = NULL,
    alter = NULL, neighborhood = c("out", "in", "combined"),
```

```
   rule = c("any", "all", "earliest", "latest"), na.omit = TRUE, active.default = TRUE)
get.neighborhood.active(x, v, onset = NULL, terminus = NULL, length = NULL, at = NULL,
   type = c("out", "in", "combined"), rule = c("any", "all", "earliest", "latest"),
    na.omit = TRUE, active.default = TRUE)

is.adjacent.active(x, vi, vj, onset = NULL, terminus = NULL, length = NULL, at = NULL,
   rule = c("any", "all", "earliest", "latest"), na.omit = FALSE, active.default = TRUE)

## S3 method for class 'active'
network.dyadcount(x, onset = NULL, terminus = NULL, length = NULL, at = NULL,
    rule = c("any", "all", "earliest", "latest"),
    na.omit = TRUE, active.default = TRUE,...)
## S3 method for class 'active'
network.edgecount(x, onset = NULL, terminus = NULL, length = NULL, at = NULL,
    rule = c("any", "all", "earliest", "latest"),
    na.omit = TRUE, active.default = TRUE,...)
## S3 method for class 'active'
network.naedgecount(x, onset = NULL, terminus = NULL, length = NULL, at = NULL,
    rule = c("any", "all", "earliest", "latest"),
    active.default = TRUE,...)
## S3 method for class 'active'
network.size(x, onset = NULL, terminus = NULL, length = NULL, at = NULL,
    rule = c("any", "all", "earliest", "latest"),
    active.default = TRUE,...)
```

## Arguments

| | |
|---|---|
| x | an object of class network |
| v | a required vertex ID. |
| vi | an optional vertex ID (tail). |
| vj | an optional vertex ID (head). |
| onset | an optional start time for the selection interval. This must be accompanied by one of terminus or length. |
| terminus | an optional end time for the selection interval. This must be accompanied by one of onset or length. |
| length | the length of the selection interval. This must be accompanied by one of onset or terminus |
| at | a time point for selecting edges/vertices. |
| alter | optionally, the ID of another vertex. |
| neighborhood | a text string that indicates whether we are interested in in-edges, out-edges, or both (relative to v). |
| type | a text string that functions as neighborhood, above, for vertex neighborhoods. |
| rule | a text string for defining "active" for this call: any (default) to select elements appearing at any time during the specified interval, or all to select only elements active for the entire interval. For these functions, the values of earliest and latest are equivalent to any and are only included for consistency. |

| | |
|---|---|
| na.omit | logical; should we omit missing edges? |
| active.default | logical; should elements without an activity attribute be regarded as active by default? |
| ... | possible additional argumenets |

## Details

These functions are "dynamically extended" versions of their standard network counterparts. As such, conventional aspects of their behavior can be found on their respective manual pages (see below). In all cases, the sole difference between the extended and non-extended versions lies in the ability of the user to specify a time point (or interval), such that only vertices and edges active at the indicated time (or during the indicated interval) are employed by the routine.

For elements (edges and vertices) lacking an activity attribute, the active.default argument determines appropriate treatment. By default, all such elements are considered active.

## Value

The return value matches that of the original function, suitably filtered.

## Note

The current temporal implementation of these routines will be slower compared to the non-temporal versions. Do not use them unless you have to.

## Author(s)

Carter T. Butts <buttsc@uci.edu>

## References

Butts, C. T. (2008). "network: a Package for Managing Relational Data in R." *Journal of Statistical Software*, 24(2). https://www.jstatsoft.org/v24/i02/

## See Also

network.extract, is.active, get.edgeIDs, get.edges, get.neighborhood, is.adjacent, network.dyadcount, network.edgecount, network.naedgecount, network.size

## Examples

```
## is.adjacent.active
triangle <- network.initialize(3)    # create a toy network
add.edge(triangle,1,2)               # add an edge between vertices 1 and 2
add.edge(triangle,2,3)               # add a more edges
add.edge(triangle,3,1)
is.adjacent.active(triangle,vi=1,vj=2,onset=2,length=1)

## get.edges.active
net <-network.initialize(5)
```

```
net[1,2]<-1;
net[2,3]<-1;
activate.edges(net,onset=1,terminus=Inf,e=1)
activate.edges(net,onset=2,terminus=3,e=2)
get.edges.active(net,v=2,at=1)
get.edges.active(net,v=2,at=1, neighborhood="combined")
get.edges.active(net,v=2,at=2, neighborhood="combined")

## get the ids of the active edges instead (could be multiple incident edges)
get.edgeIDs.active(net,v=2,at=2)

## or get ids of the active nodes in a vertex neighborhood
get.neighborhood.active(net,v=2,at=2)

# returns both edges
get.edges.active(net,v=2,onset=1, terminus=3, neighborhood="combined", rule='any')
# returns only one edge (e=2)
get.edges.active(net,v=2,onset=1, terminus=3, neighborhood="combined", rule='all')

## network.edgecount.active
network.edgecount.active(net, at=1, active.default=FALSE)
network.edgecount.active(net, at=2, active.default=FALSE)
network.edgecount.active(net, at=5, active.default=FALSE)
```

---

`network.extract` *Temporal Extracts/Cross-sections of Dynamically Extended Network Objects*

---

## Description

Given a networkDynamic and a specified time point (or interval), return a reduced networkDynamic that only contains the vertices and edges active at the given point or over the given interval.

## Usage

```
network.extract(x, onset = NULL, terminus = NULL, length = NULL, at = NULL,
    rule = c("any", "all"), active.default = TRUE, retain.all.vertices = FALSE,
    trim.spells=FALSE)

x %t% at
```

## Arguments

| | |
|---|---|
| `x` | an object of class `network`. |
| `onset` | optionally, the start of the specified interval. This must be accompanied by one of `terminus` or `length`. |
| `terminus` | optionally, the end of the specified interval. This must be accompanied by one of `onset` or `length`. |

| | |
|---|---|
| length | optionally, the length of the specified interval. This must be accompanied by one of onset or terminus. |
| at | optionally, a single time point. |
| rule | a text string for defining "active" for this call: any if elements active at any time during the interval are to be used, or all if elements must be active over the entire interval. |
| active.default | logical; should elements without an activity attribute be regarded as active by default? |
| retain.all.vertices | logical; should the extracted network retain all vertices, ignoring the vertex activity spells of x in order to ensure that the network returned has the same size as x? |
| trim.spells | logical; should the spells of vertices, edges and their attributes in the extracted network be trimmed to match the query range? (Sensible thing to do, but could be expensive) |

## Details

For the purposes of extraction, edges are only considered active over some interval if: the edge itself is active over this time period, and both of the incident vertices are also active over the given time period.

When retain.all.vertices=FALSE (the default), the function has the following behavior:

- If at is used to specify the spell of interest, the returned network consists of all edges and vertices active at that time point.
- If an interval is supplied to specify the spell (via onset and one of terminus or length), edges and vertices active over the specified interval are used. If rule="any", then edges and vertices active at any time during the interval are returned; otherwise, only those active during the entire period are returned.

Vertices in the extracted network will have the same order as the original network, with inactive vertices removed.

When retain.all.vertices=TRUE

- All vertices are preserved in the output net to preserved network size and ids.
- if trim.spells=TRUE, 'retained' vertices will be marked as inactive (they will have 'null' spell (Inf,Inf))
- Edges that are active (by the specified rule) during the specified spell are included in the returned network (same as above.)
- Edges with one or more inactive vertices still will be removed (even if the vertex is being 'retained' to preserve network size)

If a net.obs.period network attribute is present, its observation spells will be truncated by the onset and terminus of extraction. If the onset and terminus do not intersect with any observation spells, the '$observations' component will be set to the 'null' spell c(Inf,Inf).

%t% (the temporal cross-section operator) is a simplified (and less flexible) wrapper for network.extract that returns the network of active vertices and edges at a given time point.

## Value

A networkDynamic object containing the active edges and vertices for the specified spell, when retain.all.vertices=FALSE, otherwise the network object containing all vertices, and only those edges active for the specified spell.

## Note

Note that only active vertices are included by default (retain.all.vertices=FALSE). As a result, the size of the extracted network may be smaller than the original. Vertex and edge ids will be translated, but may not correspond to their original values. If it is necessary to maintain the identities of vertices, see persistent.ids.

## Author(s)

Carter T. Butts <buttsc@uci.edu>, skyebend

## See Also

is.active, activity.attribute, network.extensions, and get.inducedSubgraph for a related non-temporal version, network.collapse to squish a networkDynamic object into a static network

## Examples

```
triangle <- network.initialize(3)    # create a toy network

# add edges with activity
# first add an edge between vertices 1 and 2
add.edges.active(triangle,onset=0,terminus=10,tail=1,head=2)
# add a more edges
add.edges.active(triangle,onset=0,length=4,tail=2,head=3)
add.edges.active(triangle,at=4.5,tail=3,head=1)

# specify some vertex activity
activate.vertices(triangle,onset=0,terminus=10)
deactivate.vertices(triangle,onset=1,terminus=2,v=3)

degree<-function(x){as.vector(rowSums(as.matrix(x))
  + colSums(as.matrix(x)))} # handmade degree function

degree(triangle) # degree of each vertex, ignoring time

degree(network.extract(triangle,at=0))
degree(network.extract(triangle,at=1)) # just look at t=1
degree(network.extract(triangle,at=2))
degree(network.extract(triangle,at=5))

# watch out for empty networks! they are just an empty list
t10 <- network.extract(triangle,at=10)
t10
```

```
# notice difference between 'at' syntax and 'onset,terminus'
# when network is not in discrete time
degree(network.extract(triangle,at=4))
degree(network.extract(triangle,onset=4,terminus=5))

# the %t% (time) operator is like an alias for the 'at' extraction syntax
degree(triangle%t%4)

par(mfrow=c(2,2))
#show multiple plots
plot(triangle,main='ignoring dynamics',displaylabels=TRUE)
plot(network.extract(triangle,onset=1,terminus=2),main='at time 1',displaylabels=TRUE)
plot(network.extract(triangle,onset=2,terminus=3),main='at time 2',displaylabels=TRUE)
plot(network.extract(triangle,onset=5,terminus=6),main='at time 5',displaylabels=TRUE)
```

---

`networkDynamic` *Convert various forms of network timing information into networkDynamic objects.*

---

## Description

Converts various forms of temporal data (spell matrices, togles, lists of networks ) describing dynamics of vertices and edges into networkDynamic objects.

## Usage

```
networkDynamic(base.net = NULL, edge.toggles = NULL, vertex.toggles =NULL,
                edge.spells = NULL, vertex.spells = NULL,
                edge.changes = NULL, vertex.changes = NULL,
                network.list = NULL, onsets = NULL, termini = NULL,
                vertex.pid = NULL, start = NULL, end = NULL,
                net.obs.period=NULL,verbose=TRUE,create.TEAs = FALSE,
                edge.TEA.names=NULL,vertex.TEA.names=NULL,...)
```

## Arguments

| | |
|---|---|
| `base.net` | A network (or network-coearceable) object which will be used to define network-level properties (directedness,etc) of the output network. When constructing from toggles, the edges in base.net give the initially active set of edges and vertices that the activity states will be toggled from. Network and vertex attributes of `base.net` will be copied to output (unless they are overwritten by dynamic attributes with the same names) |
| `edge.spells` | A matrix or data.frame of spells specifying edge timing. Assumed to be [onset,terminus,tail vertex.id, head vertex.id]. Any additional columns can loaded as dynamic attributes (see `edge.TEA.names`) |

| | |
|---|---|
| vertex.spells | A matrix or data.frame of spells specifying vertex timing. Assumed to be [onset,terminus,vertex.id] |
| edge.toggles | A matrix or data.frame of toggles giving a sequence of activation and deactivation times for toggles. Columns are assumed to be [toggle time, tail vertex id of the edge, head vertex id of the edge]. |
| vertex.toggles | A matrix or data.frame of toggles giving a sequence of activation and deactivation times for vertices. Column order assumed to be [time,vertex.id] |
| edge.changes | A matrix or data.frame of edge changes with at least 4 columns, assumed to be [time, tail, head, direction] |
| vertex.changes | A matrix or data.frame of vertex changes with at least 3 columns, assumed to be [time, vertex.id,direction] |
| network.list | a list of network objects assumed to describe sequential panels of network observations. Network sizes may vary if some vertices are only active in certain panels. See onsets, termini, vertex.pid. If base.net not specified, first element of list will be used as base.net. Network, vertex, and edge attributes will be converted to TEAs if create.TEAs=TRUE. |
| onsets | an optional array of onset times to be assigned to the network panels of network.list. defaults to seq(from=0,length=length(network.list)-1) |
| termini | an optional array of terminus times to be assigned to the network panels of network.list defaults to seq(from=1,length=length(network.list) |
| vertex.pid | an optional name of a vertex attribute to be used as a unique vertex identifier when constructing from a network list with different sized networks. |
| start | Optional argument to indicate the earliest time at which any changes in the network could have been observed or simulated; any spells before this time point are considered onset-censored. |
| end | Optional argument to indicate the latest time at which any changes in the network could have been observed or simulated; any spells after this time point are considered terminus-censored. |
| net.obs.period | Optional argument. A structured list for providing additional information about when and how the network was observed. |
| verbose | Logical, If TRUE (default), status message will be printed about the assumptions made in the conversion process. |
| create.TEAs | If TRUE, Dynamic TEA attributes will be created corresponding to the static attributes appear on the network elements of network.list |
| edge.TEA.names | an optional vector of names for the dynamic (TEA) edge attributes to be imported from the extra columns of edge.spells (if create.TEAs=TRUE) |
| vertex.TEA.names | an optional vector of names for the dynamic (TEA) vertex attributes to be imported from the extra columns of vertex.spells (if create.TEAs=TRUE) |
| ... | Additional arguments controlling the creating of the network or processing of attached data objects. |

## Details

This function provides ways to convert multiple forms of timing information for vertices and edges into network dynamic objects.

**Converting from lists of networks:** If the timing information is provided in the form of a lists of networks (specified by the `network.list` argument ) the assumption is that each network is a discrete 'panel' observation indicating the active set vertices and edges. By default, each observation is assumed to span a unit interval, (so the 1st goes from 0 to 1, 2nd from 1-2, etc). However, the onset and termini of each observation panel can be explicitly specified via the `onsets` and `termini` arguments. If the networks in `network.list` vary in size, the name of a vertex attribute to be used as a persistent id must be specified using the `vertex.pid` attribute so that the vertices in each network can be matched up correctly. If `create.TEAs=TRUE`, dynamic attributes will be created for any (non-default) attributes appearing in the list of networks. Otherwise, network and vertex attributes will be copied from the network specified via `base.net`. If `base.net` is not specified, the first element of `network.list` will be used. A `net.obs.period` will be constructed to describe the range of observations used to construct the network, or one can be specified directly using the `net.obs.period` argument.

**Converting from matrices of spells, changes or toggles:** Alternatively, timing information for edges and vertices can be provided separately as matrices or data.frames in the forms of tables of spells, changes, or toggles. For vertices, the arguments are `vertex.spells`, `vertex.changes` or `vertex.toggles`. The columns for each argument have a specific order and function, indicated in the argument definitions above. Edge arguments are named similarly `edge.spells`, `edge.changes` or `edge.toggles`. The vertex ids and 'head' and 'tail' ids must be integer index ids. Network properties (number of vertices, bipartite, directedness, etc) can currently be specified by including a `base.net` argument to be used as a template, network and vertex attributes will be copied from `base.net` as well. If only edge information is provided, the network size will be imputed from the set of edge endpoints.

In the `edge.toggles` case, the edges present in `base.net` provide the initial state to be toggled from. If a `net.obs.period` is not specified directly using the `net.obs.period` argument one will be constructed. Since the correct observation information is not known it may not describe the data accurately. Spells data will be assumed to be 'continuous' and where toggles and changes will be assumed to be 'discrete' and the `net.obs.period$observations` may include infinate values indicating unknown bounds.

When constructing a network with separate sources of data for vertex and edge timing, edge and vertex activity will not be made consistent if input data is inconsistent – edges may be active when their incident vertices are inactive, etc. These situations can be detected using the `network.dynamic.check` function and possible resolved using one of the `reconcile.activity` functions.

Curently, dynamic attributes (TEAs, see `attribute.activity.functions`) for edges or vertices (with onset and termination times matching those of the edge or vertex spell) can be loaded in by by setting `create.TEAs=TRUE` and including additional columns in the `edge.spells` or `vertex.spells` argument and specifying a vector of names for the attributes with `edge.TEA.names` or `vertex.TEA.names`. Note that when using a `data.frame` to pass in non-numeric attributes, the default behavior of converting strings to factors should be avoided.

## Value

A networkDynamic object with vertex and edge timing specified by the input data. It will also print out information about the assumptions it makes while constructing the network.

## Note

This function provides the features that were formerly (in versions < 0.4) called using as.networkDynamic.* syntax.

## Author(s)

Li Wang lxwang@uw.edu, skyebend, statnet team

## See Also

See Also as get.edge.activity,get.vertex.activity, network.dynamic.check

## Examples

```
# construct network out of a list of panels of varying sizes
# omiting missing day 25
data(windsurferPanels)
dynBeach<-networkDynamic(network.list=beach[-25], vertex.pid="vertex.names",
                         onsets=c(1:24,26:31),termini=c(2:25,27:32))

# read in tsv files for vertex and edge spells and
# construct network of McFarland classroom interaction data
# see ?cls33_10_16_96 for more info about this data set

# read vertex data
rawVerts<-read.table(paste(path.package('networkDynamic'),
   "/extdata/cls33_10_16_96_vertices.tsv",sep=''),header=TRUE)

# peek at column headings to find ids and times
names(rawVerts)

# read in interation (edge) data
rawEdges<-read.table(paste(path.package('networkDynamic'),
  "/extdata/cls33_10_16_96_edges.tsv",sep=''),header=TRUE)

# peek at column headings to find ids and times
names(rawEdges)


# construct network using vertex and edge timing information
cls33 <-networkDynamic(vertex.spells=rawVerts[,c(3,4,1)],
                       edge.spells=rawEdges[,c(3,4,1,2)])

# add in the unchanging vertex attribute data
set.vertex.attribute(cls33,"sex",as.vector(rawVerts$sex))
set.vertex.attribute(cls33,"role",as.vector(rawVerts$role))
```

```
 ## Not run:   # takes 5 seconds, too slow for CRAN checks
# loop over edge data to add the dynamic attributes on the edge
for(r in 1:nrow(rawEdges)){
  # get the id of the edge from its tail and head
  eid <- get.edgeIDs(cls33,v=rawEdges$from_vertex_id[r],
                     alter=rawEdges$to_vertex_id[r])
  activate.edge.attribute(cls33,'interaction_type',rawEdges$interaction_type[r],
          onset=rawEdges$start_minute[r],terminus=rawEdges$end_minute[r],e=eid)
  activate.edge.attribute(cls33,'weight',rawEdges$weight[r],
          onset=rawEdges$start_minute[r],terminus=rawEdges$end_minute[r],e=eid)
}

# convert the set of newcomb panels with rank weights
data(newcomb)
newRankDyn <-networkDynamic(network.list=newcomb.rank)
get.network.attribute.active(newRankDyn,'title',at=3)
as.matrix(network.collapse(newRankDyn,at=5),attrname='rank')


## End(Not run)
```

---

Newcomb *Newcomb's Fraternity Networks*

---

## Description

These 14 networks record weekly sociometric preference rankings from 17 men attending the University of Michigan in the fall of 1956; Data were collected longitudinally over 15 weeks, although data from week 9 are missing.

The men were recruited to live in off-campus (fraternity) housing, rented for them as part of the Michigan Group Study Project supervised by Theodore Newcomb from 1953 to 1956. All were incoming transfer students with no prior acquaintance of one another.

The data set contains two longitudinal networks as `network.list`.

`newcomb.rank` is a `network.list` object with 14 networks. Each network is complete and the edge value `rank` is the preference of the $i$th men for the $j$th man from 1 through 16. A 1 indicates first preference, and no ties were allowed.

`newcomb` is a `network.list` object that has binary edge values but is similar in structure to `newcomb.rank` and derived from it. Each network has a tie from the $i$th men to the $j$th man if $i$ had a preference for $j$ of 8 or less. Otherwise there is not tie from $i$ to $j$. Note that since these are ranks, the degree of each vertex (and the total number of edges) does not vary over time

## Usage

```
data(newcomb)
```

## Details

Use `data(package="netdata")` to get a full list of networks.

## Licenses and Citation

If the source of the data set does not specified otherwise, this data set is protected by the Creative Commons License https://creativecommons.org/licenses/by-nc-nd/2.5/.

When publishing results obtained using this data set the original authors should be cited. In addition this package should be cited as:

Mark S. Handcock, David Hunter, Carter T. Butts, Steven M. Goodreau, and Martina Morris. 2003 *statnet: An R package for the Statistical Modeling of Social Networks* https://statnet.org/ and the source should be cited as:

Vladimir Batagelj and Andrej Mrvar (2006): *Pajek datasets* http://vlado.fmf.uni-lj.si/pub/networks/data/

## Source

http://vlado.fmf.uni-lj.si/pub/networks/data/ucinet/ucidata.htm#newfrat

## References

See the link above. Newcomb T. (1961). The acquaintance process. New York: Holt, Reinhard and Winston.\ Nordlie P. (1958). A longitudinal study of interpersonal attraction in a natural group setting. Unpublished doctoral dissertation, University of Michigan.\ White H., Boorman S. and Breiger R. (1977). Social structure from multiple networks, I. Blockmodels of roles and positions. American Journal of Sociology, 81, 730-780.

## See Also

network, sna

## Examples

```
data(newcomb)
```

---

# persistent.ids *Use and manipulate persistent ids of network elements*

---

## Description

Persistent ids (pids) are unique values attached to vertices or edges which remain unchanged throughout network manipulation and extraction processes. The "vertex.pid" of a vertex is an overall dataset id, unlike the "vertex.id" which is an index dependent on network size. These functions provide ways to look up vertex.pids from vertex.ids (and the other way around) and also provide necessary modifications to some standard network functions to safely work with pids.

## Usage

```
get.vertex.id(nd, pid)

get.vertex.pid(nd, id)

get.edge.id(nd, pid)

get.edge.pid(nd, id)

initialize.pids(nd)

vertex.pid.check(nd)

edge.pid.check(nd)
```

## Arguments

| | |
|---|---|
| nd | a network or networkDynamic object to be modified |
| pid | persistent id(s) of the network element(s) for which the network-specific id(s) should be returned |
| id | network-specific (index) id(s) of the network element for which the persistent id(s) should be returned |

## Details

The persistent id functionality is an optional feature of networkDynamic objects. If a network-level attribute named vertex.pid exists, its value is required to be the name of a vertex attribute that can safely be used as a vertex.pid. If it is set to NULL, pids will be ignored. A vertex.pid must have a unique value for all vertices. Persistent ids for edges function in the same way, except the attribute is named edge.pid.

Some existing network code uses the vertex.names attributes of networks as a persistent id without checking that it is unique. It is possible to indicate that vertex.names can safely be used as a vertex.pid by setting vertex.pid to 'vertex.names'

The function initialize.pids can be used to create a set of pids on all existing vertices (named vertex.pid and edges named edge.pid). The pids are currently initialized with pseudo-random hex strings using the tempfile function (something like '4ad912252bc2'). It is especially useful if called after a network object as been constructed and before any extractions are performed.

The element addition functions (add.vertices) override their network-package counterparts in order to permit assigning pids to newly-added elements if the pid is defined. They can be defined by the user with the vertex.pids argument, or, if not specified, a unique random id will be generated. (Note that any new values added by default to a vertex.names pid will not be numeric.)

Some of the import/conversion methods may set pids. See network.

User-specified pids are checked for uniqueness. The the current auto-generated pid implementation produces ids that are unique within the current network. They are also *almost* certain to be unique within an R session (so that vertices will have a unique id if added and removed) and quite likely across sessions, but we need more details on the tempfile's implementation.

With the exception of the "check" utilities, all of these functions modify their network argument in place.

## Value

Get methods:

- `get.vertex.id` returns the vertex.id(s) of vertices corresponding to the vertex.pid(s) provided.
- `get.vertex.pid` returns the vertex.pid(s) of vertices corresponding to the vertex.id(s) provided.
- `get.edge.id` returns the edge.id(s) of edge corresponding to the edge.pid(s) provided.
- `get.edge.pid` returns the edge.pid(s) of edges corresponding to the edge.id(s) provided.

Each of the above return `NA` values where no match is found.

- `vertex.pid.check` throws an error if the `vertex.pid` is found to not meet specifications. Otherwise returns `TRUE` or `FALSE` and gives a warning if vertex.pid does not exist
- `edge.pid.check` throws an error if the `edge.pid` is found to not meet specifications. Otherwise returns `TRUE` or `FALSE` and gives a warning if edge.pid does not exist

## Note

Adding edges via the extraction/replacement operators `[,]` bypasses the pid code and will break the edge pids defined for the network. Similarly, `add.vertices.active` and `add.edges.active` do not yet support including pids.

## Author(s)

lxwang, skyebend, the statnet team

## See Also

See also `add.vertices` in network.

## Examples

```
# use vertex.names as a persistent id
net<-network.initialize(5)
set.network.attribute(net, 'vertex.pid','vertex.names')

# find original vertex corresponding to vertex in smaller extracted net
haystack<-network.initialize(30)
activate.vertices(haystack,v=10:20)

# hide a needle somewhere in the haystack
set.vertex.attribute(haystack,'needle',TRUE,v=sample(1:30,1))

# set up the persistand ids with defaults
```

```
initialize.pids(haystack)

# some hay is removed over time ...
newstack<-network.extract(haystack,at=100,active.default=FALSE)
network.size(newstack)
# we find the needle!
needleId <-which(get.vertex.attribute(newstack,'needle'))
needleId

# which vertex is the corresponding one in original stack?
oldId<-get.vertex.id(haystack,get.vertex.pid(newstack,needleId))
oldId

# check if we got it right..
get.vertex.attribute(haystack,'needle')[oldId]


# one reason you wouldn't want to use ordinary vertex.names
net<-network.initialize(3)
add.vertices(net,3)
network.vertex.names(net)

# but if you make it a persistant id, new names will be created
net<-network.initialize(3)
set.network.attribute(net,'vertex.pid','vertex.names')
add.vertices(net,3)
network.vertex.names(net)

# try with edges and add/remove vertices
net <-network.initialize(10)
add.edges(net,1:9,2:10)
set.edge.attribute(net,'test',"you found me!",e=7)
initialize.pids(net)
changed<-net
add.vertices(changed,5)
delete.vertices(changed,c(1,3,5,15))
delete.edges(changed,eid=1:3)

# which edge in changed corresponds to edge 7 in net?
network.edgecount(changed)
get.edge.id(changed,get.edge.pid(net,7))
# actually, they are the same because of NULL edges in edgelist
get.edge.attribute(changed,'test',unlist=FALSE)[[7]]
# however, the ids of the vertices have changed
changed$mel[[7]]$inl
net$mel[[7]]$inl

# do they still match up?
get.vertex.pid(changed,changed$mel[[7]]$inl)==get.vertex.pid(net,net$mel[[7]]$inl)
```

`print.networkDynamic` *Provide a view of a networkDynamic object including timing information*

## Description

Prints out some very basic descriptive stats about the network's dynamics, and the the normal information printed for network objects.

## Usage

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

## Arguments

| | |
|---|---|
| `x` | an object of class networkDynamic including dynamic relational information |
| `...` | |

## Details

Prints some info about the number of time events in a networkDynamic object, then calls print.network

## Value

Returns the networkDynamic object passed to the function.

## Note

Only prints out the network information without any dynamic data.

## Author(s)

Pavel

## See Also

`print.network`

## Examples

```
library(networkDynamic)
data(flo)
net1 <- network(flo)
activate.edges(net1, onset=1:20, terminus=101:120)
activate.vertices(net1, at=seq(2,32,2))
print(net1)
```

read.son *Read .son-formatted (SoNIA) input files containing longitudinal network data and create a corresponding networkDynamic object*

## Description

A .son file is a tab-separated text file with two sections, the first describing node attribute events and the second arc attribute events. Each section has a set of defined column names as a header. See the SoNIA file specification for more information: https://sourceforge.net/p/sonia/wiki/Son_format/.

## Usage

```
read.son(file, guess.TEA=TRUE)
```

## Arguments

| | |
|---|---|
| file | path to the file to be read |
| guess.TEA | logical: should the parser guess that non-changing attribute values are intended to be static instead of TEAs? |

## Details

Uses networkDynamic internally to build a dynamic network out of an array of vertex spells and an array of edge spells. Vertex and edge attributes will be attached, usually as TEA. However, If the values of an attribute never change, they will be attached as a static attribute. This is convenient but will be incorrect if a value was intended to be defined for only a single spell matching with the vertex activation. Setting guess.TEA=FALSE will force all attributes to be loaded as TEAs (slower).

## Value

A network dynamic object with vertex and edge durations and attributes corresponding the the spells defined in the input file.

## Note

The current version of this function does not support reading clusters.

## Author(s)

Skye Bender-deMoll

## References

https://sourceforge.net/p/sonia/wiki/Son_format/.

## See Also

See Also networkDynamic

## Examples

```
classroom<-read.son(system.file('extdata/cls33_10_16_96.son',package='networkDynamic'))
```

---

`reconcile.activity` *Modify the activity spells of vertices to match incident edges or the other way around*

---

## Description

When networkDynamic objects are created from real-world data it is often the case that activity information for vertices and edges may not come from the same source and may not match up exactly. Vertices may be inactive when incident edges are active, etc. The `reconcile.vertex.activity` function modifies the activity of a network's vertices acording to the `mode` specified, while `reconcile.edge.activity` performs similar operations for edges.

## Usage

```
reconcile.vertex.activity(net,
               mode = c ("expand.to.edges", "match.to.edges", "encompass.edges"),
                      edge.active.default = TRUE)

reconcile.edge.activity(net,
                     mode = c("match.to.vertices", "reduce.to.vertices"),
                     active.default = TRUE)
```

## Arguments

| | |
|---|---|
| `net` | a networkDynamic object |
| `mode` | string indicating the method for reconciling |
| `edge.active.default` | boolean, should edges with no definied activity be considered active? |
| `active.default` | boolean, should vertices with no definied activity be considered active? |

## Details

The `mode` argument takes the following values:

`match.to.edges` vertices will be modified so as to be only active when incident edges are active. Existing vertex spells are deleted.

- `expand.to.edges` vertices activity will be expanded to include the activity periods of any incident edges (still permits isolated vertices). Existing vertex spells are preserved.

- encompass.edges vertices activity will be modified so that it has a single spell beginning with the earliest incident edge activity, and encompasses the last edge activity. (Note that this implies that isolated vertices will become inactive).
- match.to.vertices edges will be modified so as to be active whenever all incident vertices are active. Existing inactive edges will be activated if necessary.
- reduce.to.vertices edges will be modified so as to have no active spells when incident vertices are inactive. Inactive edges will not be activated.

## Value

The input networkDynamic object is modified in place and returned invisibly.

## Author(s)

skyebend, lxwang

## Examples

```
nd<-network.initialize(6)
add.edges.active(nd,tail=1:3,head=2:4,onset=1,terminus=3)
add.edges.active(nd,tail=4,head=1,onset=5,terminus=7)
add.edge(nd,tail=1,head=6)
# before
get.vertex.activity(nd,as.spellList=TRUE)
reconcile.vertex.activity(nd)
# after
get.vertex.activity(nd,as.spellList=TRUE)

# induce edge activity for known vertex timing
nd<-network.initialize(4,directed=FALSE)
activate.vertices(nd,onset=1:4,terminus=3:6)
nd[,]<-1
get.edge.activity(nd,as.spellList=TRUE)
reconcile.edge.activity(nd, mode="reduce.to.vertices")
get.edge.activity(nd,as.spellList=TRUE)
```

---

# spells.overlap *Functions to compare and search spell intervals*

---

## Description

spells.overlap checks if two spells overlap at all, spells.hit searches a spell matrix for an overlapping spell

## Usage

```
spells.overlap(s1, s2)

spells.hit(needle, haystack)

search.spell(needle, haystack)
```

## Arguments

| | |
|---|---|
| s1 | First spell for in the comparison, must be a two-element numeric vector in the form [onset,terminus] |
| s2 | Second spell for in the comparison, must be a two-element numeric vector in the form [onset,terminus] |
| needle | The query spell used in the search, must be a two-element numeric vector in the form [onset,terminus] |
| haystack | The spell matrix to be searched by needle. Must be two column numeric matrix meeting the order specifications of a spell matrix. See activity.attribute |

## Details

Spell overlap is defined as true if one of the following conditions sets is met, otherwise false:

onset1 >= onset2 AND onset1 < terminus2

terminus1 > onset2 AND terminus1 <= terminus2

onset1 <= onset2 AND terminus1 >= terminus2

## Value

spells.overlap returns a boolean indicating if the two spells overlap or not.

spells.hit returns the integer row index of the first (earliest) spell in haystack that overlaps with needle, or -1 if no overlaps are found

search.spell returns a vector containing the row indices of spells in haystack that overlap with needle, or numeric(0) if none found

## Author(s)

skyebend@uw.edu

## See Also

See Also activity.attribute

## Examples

```
a <- c(1,3)
b <- c(2,5.5)
c <- c(-1,10)
d <- c(4,4)
```

```
spells.overlap(a,b)
spells.overlap(b,c)
spells.overlap(a,d)

spellmat <- rbind(c(0,1), c(1,2), c(2,3))
spells.hit(c(1,2),spellmat)
```

---

`when.vertex.attrs.match` *return the times at which the TEA attributes of elements of a network match a value*

---

## Description

The functions query the TEA attributes of network elements (vertices or edges) and return the time that each element matches the specified value. The match operator defaults to '==', but other binary comparison operators that return logical, such as '>' or even '%in%' can be used as well.

## Usage

```
when.vertex.attrs.match(nd, attrname, value, match.op = "==", rule = "earliest",
                        no.match = Inf, v = seq_len(network.size(nd)))

when.edge.attrs.match(nd, attrname, value, match.op = "==", rule = "earliest",
                        no.match = Inf, e = seq_along(nd$mel))
```

## Arguments

| | |
|---|---|
| `nd` | a `networkDynamic` object |
| `attrname` | character name of the TEA attribute to be queried |
| `value` | the value to be passed to the comparison operator to be matched against the TEA values |
| `match.op` | the binary operator to be used in determining a match (defaults to '==') |
| `rule` | character giving the name of the rule to determine what value should be returned from the spell of matching attributes. Options are `'earliest'` (the default) and `'latest'` which will return the onset and terminus of the spell respectively. |
| `no.match` | The value to be returned when the TEA attribute never matches `value`. Default is `Inf`, `NA` may be useful as well. |
| `v` | possible numeric vector of vertex ids indicating a subset of vertices to be queried |
| `e` | possible numeric vector of edge ids indicating a subset of edges to be queried |

## Details

The `no.match` argument is included to make it possible for user to distinguish cases where the attribute is missing from the network element (which will return `NA`) from cases where the value of the attribute never matches `value` according to the operation `match.op`

## Value

A numeric vector of attribute onset times (if `rule='earliest'`) or termination times (if `rule='latest'`)

## Note

this is a draft implementation, suggestions on function and argument names welcome.

## Author(s)

skyebend

## See Also

See also attribute.activity.functions

## Examples

```
# create a network with TEA attribute
net<-network.initialize(5)
net<-activate.vertex.attribute(net,'test','A',onset=0,terminus=2)
net<-activate.vertex.attribute(net,'test','B',onset=2,terminus=3)
net<-activate.vertex.attribute(net,'test','C',onset=4,terminus=5)

net<-activate.vertex.attribute(net,'weight',0,onset=-Inf,terminus=Inf)
net<-activate.vertex.attribute(net,'weight',-1,at=6,v=2)

# when does 'test' take the value 'B' for each vertex?
when.vertex.attrs.match(net,'test',value='B')

# when is the terminus of when 'test' takes a value 'A' or 'C'?
when.vertex.attrs.match(net,'test',value=c('A','C'),match.op='%in%',rule='latest')

# when does 'weight' first take a value less than 0?
when.vertex.attrs.match(net,'weight',value=0,match.op='<')
```

---

# windsurfers *Lin Freeman's Dynamic Network of Windsurfer Social Interactions*

---

## Description

A dynamic network object describing daily social interactions between windsurfers on California beaches over a months time. Collected by Lin Freeman in 1988.

## Usage

```
data(windsurfers)
```

## Format

The format is is a networkDynamic object with node and edge activity.attributes defined by spell matricies.

## Details

The data in this network was originally collected and analyzed in aggregate by Freeman et al. (1988) and has since been used in a number of influential articles (see Cornwell, 2009; Hummon and Doreian, 2003; Zeggelink et al., 1996, etc.). While this network is typically analyzed in aggregate, it was originally collected as a dynamically evolving network (where the vertex set is composed of windsurfers and the edge set is composed of interpersonal communication).

The network was collected daily (sampled at two time points each day) for 31 days (August 28, 1986 to September 27,1986). From Almquist and Butts, 201:

"Individuals were tracked with a unique ID, and were divided by Freeman et al. into those we will here call, 'regulars' (N = 54) – frequent attendees who were well-integrated into the social life of the beach community – and 'irregulars' (N = 41) on ethnographic grounds. The former category was further broken down by the researchers into two groups, Group 1 (N = 22) and Group 2 (N = 21), with 11 individuals not classified as belonging to either Group 1 or Group 2. Altogether, the union of vertex sets (Vmax ) consists of 95 individuals. On any given day during the observation period, the number of windsurfers appearing on the beach ranged from 3 to 37, with the number of communication ties per day ranging from 0 to 96."

The dynamicNetwork object was created from a list of network objects (length 31, but there is one missing entry index 25 marked with an NA). Each list entry is labeled 828, 829 ... 927 – this corresponds to the date the network was collected. Vertex attributes include: group1, group2, regular, vertex.names – group1, group2, and regular are ethnographically defined (and are simply dummies in this case, i.e., 0/1), and vertex.names is the original code number used by Lin. This are static vertex attributes.

The original set of static networks have been merged into a single dynamic network object with vertex activity coded in a spell matrix following the conventions of networkDynamic objects. There is very high daily turnover of who is present on the beach on each day.

Several dynamic network level attributes (e.g., list.network.attributes) have been added by Zack Almquist. These include:

- `atmp` atmospheric temperature.
- `cord` Grid location on the beach.
- `day` simply the Monday/Tuesday/etc labeling.
- `gst` ground surface temperature (celsius).
- `week` week is the position within the month (e.g., first week in august)
- `wspd` Wind speed from noaa.gov.
- `wvht` Wave height noaa.gov.

atm, gst, wspd, wvht are from the national atmospheric data (noaa.gov) and come from the closest beach that had accurate weather data. Day of week is relevant because weekly periodicity of the nodeset is quite high.

## Source

Dr. Lin Freeman (Research Professor UCI) and Zack W Almquist (Graduate Student, UCI)

## References

Almquist, Zack W. and Butts, Carter T. (2011). "Logistic Network Regression for Scalable Analysis of Networks with Joint Edge/Vertex Dynamics." IMBS Technical Report MBS 11-03, University of California, Irvine.

Freeman, L. C., Freeman, S. C., Michaelson, A. G., 1988. "On human social intelligence." Journal of Social Biological Structure 11, 415–425.

## Examples

```
data(windsurfers)

data(windsurferPanels)
```

# Index