Package examples for networkDynamic: Dynamic Extensions for Network Objects

Version 0.5

Carter T. Butts, Skye Bender-deMoll, Ayn Leslie-Cook,
Pavel N. Krivitsky, Zack Almquist, David R. Hunter,
Martina Morris

August 20, 2013

Contents

1 Introduction

The `networkDynamic` package provides support for a simple family of dynamic extensions to the `network` (Butts, 2008) class; these are currently accomplished via the standard `network` attribute functionality (and hence the resulting objects are still compatible with all conventional routines), but greatly facilitate the practical storage and utilization of dynamic network data. The dynamic extensions are motivated in part by the need to have a consistent data format for exchanging data, storing the inputs and outputs to relational event models, statistical estimation and simulation tools such as `ergm` (Hunter et al., 2008b) and `tergm` (Krivitsky P and Handcock M , 2013), and dynamic visualizations.

The key features of the package provide basic utilities for working with networks in which:

- Vertices have 'activity' or 'existence' status that changes over time (they enter or leave the network)
- Edges which appear and disappear over time
- Arbitrary attribute values attached to vertices and edges that change over time
- Meta-level attributes of the network which change over time
- Both continuous and discrete time models are supported, and it is possible to effectively blend multiple temporal representations in the same object

In addition, the package is primarily oriented towards handling the dynamic network data inputs and outputs to network statistical estimation and simulation tools like `statnet` and `tergm`. This document will provide a quick overview and use demonstrations for some of the key features. We assume that the reader is already familiar with the use and features of the `network` package.

Note: Although `networkDynamic` shares some of the goals (and authors) of the experimental and quite confusable `dynamicNetwork` package (Bender-deMoll et al., 2008), they are are incompatible.

2 How to start and end relationships easily

A very quick condensed example of starting and ending edges to show why it is useful and some of the alternate syntax options.

2.1 Activating edges

The standard assumption in the `network` package and most sociomatrix representations of networks is that an edge between two vertices is either present or absent. However, many of the phenomena that we wish to describe with networks are dynamic rather than static processes, having a set of edges which change over time. In some situations the edge connecting a dyad may break and reform multiple times as a relationship is ended and re-established. The `networkDynamic` package adds the concept of 'activation spells' for each element of a `network` object. Edges are considered to be present in a network when they are active, and treated as absent during periods of inactivity. After a relationship has been defined using the normal syntax or network conversion utilities, it can be explicitly activated for a specific time period using the `activate.edges` methods. Alternatively, edges can be added and activated simultaneously with the `add.edges.active` helper function.

```
> library(networkDynamic)          # load the dynamic extensions
> 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
```

```
> activate.edges(triangle,at=1) # turn on all edges at time 1 only
> activate.edges(triangle,onset=2, terminus=3,
+                e=get.edgeIDs(triangle,v=1,alter=2))
> add.edges.active(triangle,onset=4, length=2,tail=3,head=1)
```

Notice that the `activate.edges` method refers to the relationship using the `e` argument to specify the ids of the edges to activate. To be safe, we are looking up the ids using the `get.edgeIDs` method with the `v` and `alter` arguments indicating the ids of the vertices involved in the edge. The `onset` and `terminus` parameters give the starting and ending point for the activation period (more on this and the `at` syntax later). When a network object has dynamic elements added, it also gains the `networkDynamic` class, so it is both a `network` and `networkDynamic` object.

```
> class(triangle)

[1] "networkDynamic" "network"

> print(triangle)

NetworkDynamic properties:
  distinct change times: 5
  maximal time range: 1 to 6

 Network attributes:
  vertices = 3
  directed = TRUE
  hyper = FALSE
  loops = FALSE
  multiple = FALSE
  bipartite = FALSE
  total edges= 3
    missing edges= 0
    non-missing edges= 3

 Vertex attribute names:
    vertex.names

 Edge attribute names:
    active
```

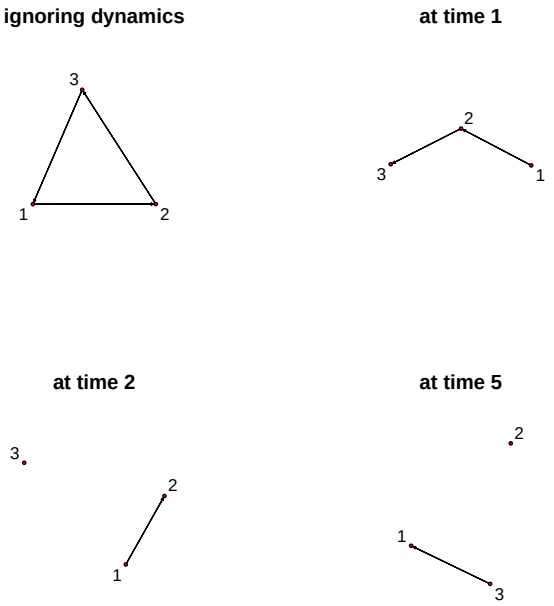
2.2 Peeking back in time

After the activity spells have been defined for a network, it is possible to extract views of the network at arbitrary points in time using the `network.extract` function in order to calculate traditional graph statistics.

```
> degree<-function(x){as.vector(rowSums(as.matrix(x))
+              +colSums(as.matrix(x)))} # handmade degree function
> degree(triangle)  # degree of each vertex, ignoring time

[1] 2 2 2

> degree(network.extract(triangle,at=0))

[1] 0 0 0

> degree(network.extract(triangle,at=1)) # just look at t=1

[1] 1 2 1

> degree(network.extract(triangle,at=2))

[1] 1 1 0

> degree(network.extract(triangle,at=5))

[1] 1 0 1

> degree(network.extract(triangle,at=10))

[1] 0 0 0
```

The vertex degrees at each extracted time point are different what would be expected for the "timeless" network. When the network was sampled outside of the defined time range (at 0 and 10) it returned degrees of 0, suggesting that no edges are present at all. It may be helpful to plot the networks to help understand what is going on. The plots below show the result of the standard plot command (`plot.network.default`) for the triangle, as well as plots of the network at specific time points.

```
> par(mfrow=c(2,2))   #show multiple plots
> plot(triangle,main='ignoring dynamics',displaylabels=T)
> plot(network.extract(
+   triangle,onset=1,terminus=2),main='at time 1',displaylabels=T)
> plot(network.extract(
+   triangle,onset=2,terminus=3),main='at time 2',displaylabels=T)
> plot(network.extract(
+   triangle,onset=5,terminus=6),main='at time 5',displaylabels=T)
```



3 Birth, Death, Reincarnation and other ways for vertices to enter and leave networks

3.1 Activating vertices

Many network models need the ability to specify activity spells for vertices in order to account for changes in the population due to 'vital dynamics' (births and deaths) or other types of entrances and exits from the sample population. In `networkDynamic` activity spells for a vertex can be specified using the `activate.vertices` methods. Like edges, vertices can have multiple spells of activity. If we build on the triangle example:

```
> 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)
> network.size.active(triangle,at=1) # how big is it?
```

```
[1] 2
```

```
> network.size.active(triangle,at=4)
```

```
[1] 3
```

```
> network.size.active(triangle,at=5)
```

```
[1] 2
```

Using the `network.size.active` function shows us that specifying the activity ranges has effectively changed the sizes (and corresponding vertex indices–more on that later) of the network. Notice also that we've created contradictions in the definition of this hand-made network, for example stating that vertex 3 isn't active until time 4 when earlier we said that there were ties between all nodes at time 1. The package does not prohibit these kinds of paradoxes, but it does provide a utility to check for them.

```
> network.dynamic.check(triangle)
```

```
Edges were found active where the endpoints where not in edge(s) 2 3.
$vertex.checks
[1] TRUE TRUE TRUE

$edge.checks
[1] TRUE TRUE TRUE

$dyad.checks
[1]  TRUE FALSE FALSE

$vertex.tea.checks
[1] TRUE TRUE TRUE

$edge.tea.checks
[1] TRUE TRUE TRUE

$network.tea.checks
[1] TRUE

$net.obs.period.check
NULL
```

3.2 Deactivating elements

In this case, we can resolve the contradictions by explicitly deactivating the edges involving vertex 3:

```
> deactivate.edges(triangle,onset=1,terminus=4,
+           e=get.edgeIDs(triangle,v=3,neighborhood="combined"))
> network.dynamic.check(triangle)
```

```
Edges were found active where the endpoints where not in edge(s) 3.
$vertex.checks
[1] TRUE TRUE TRUE

$edge.checks
[1] TRUE TRUE TRUE

$dyad.checks
[1]  TRUE  TRUE FALSE

$vertex.tea.checks
[1] TRUE TRUE TRUE

$edge.tea.checks
[1] TRUE TRUE TRUE

$network.tea.checks
[1] TRUE

$net.obs.period.check
NULL
```

The deactivation methods for vertices,`deactivate.vertices`, works the same way, but it accepts a `v=` parameter to indicate which vertices should be modified instead of the `e=` parameter.

4 "Spells": the magic under the hood

In which we provide a brief glimpse into the underlying data structures.

4.1 How we save time

There are many possible ways of representing change in an edge set over time. Several of the most commonly used are:

- A series of networks or network matrices representing the state of the network at sequential time points
- An initial network and a list of edge toggles representing changes to the network at specific time points
- A collection of 'spell' intervals giving the onset and termination times of each element in the network they are attached to
- A set of multiplex edges with time values attached.

This package uses the spell representation, and stores the spells as a perfectly normal but specially named `active` attributes on the network. These attributes are a 2-column spell matrix in which the first column gives the onset, the second the terminus, and each row defines an additional activity spell for the network element. For more information, see `?activity.attribute`. As an example, to peek at the spells defined for the vertices:

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

```
[[1]]
     [,1] [,2]
[1,]    1    5

[[2]]
     [,1] [,2]
[1,]    1   10

[[3]]
     [,1] [,2]
[1,]    4   10
```

```
> get.edge.activity(triangle) # edge spells
```

```
[[1]]
     [,1] [,2]
[1,]    1    1
[2,]    2    3

[[2]]
NULL

[[3]]
     [,1] [,2]
[1,]    4    6
```

Notice that the first edge has a 2-spell matrix where the first spell extends from time 1 to time 1 (a zero-duration or instantaneous spell), and the second from time 2 to time 3 (a "unit length" spell. More on this below). The third edge has the interesting special "null" spell `c(Inf,Inf)` defined to mean 'never active' which was produced when we deleted the activity associated with the 3rd edge.

Within this package, spells are assumed to be 'right-open' intervals, meaning that the spell includes its lower bound but not its upper bound. For example, the spell [2,3) covers the range between t>=2 and t<3. Another way of thinking

of it is that terminus means "until". So the spell ranges from 2 *until* 3, but does not include 3.

Although it would certainly be possible to directly modify the spells stored in the `active` attributes, it is much safer to use the various `activate` and `deactivate` methods to ensure that the spell matrix remains in a correctly defined state. The goal of this package is to make it so that it is rarely necessary to work with spells, or even worry very much about the underlying data structures. It should be possible to use the provided utilities to convert between the various representations of dynamic networks. However, even if the details of data structure can be ignored, it is still important to be very clear about the underlying temporal model of the network you are working with.

4.2 Multiple spells != multiplex

One of the features that makes the `network` package so flexible is that it allows *multiplex* edges. This means that a pair (or set ...) of vertices can be linked by multiple "parallel" edges. Often this is used as a way to store several different kinds of relations within the same network object. It is important to be clear that, as we have defined it, having multiplex edges between vertices is not the same thing as an edge with multiple activity spells. It is entirely possible to activate multiple edges between a vertex pair with different spell values in order to attach relationship-specific timing information for situations where this an appropriate and useful representation.

5 Differences between Discrete and Continuous data

Its 2 AM on Tuesday. Do you know what your temporal model is? Does 2 AM mean 2:00 AM, or from 2:00 to 2:59:59? We discuss this below, as well as other existential questions such as the differences between "at" and "onset, terminus" syntax.

There are two key approaches to representing time when measuring something.

5.1 You might be discrete if...

The *discrete* model thinks of time as equal chunks, ticks, discrete steps, or panels. To measure something we count up the value of interest for that chunk. Discrete time is expressed as series of integers. We can refer to the 1st step, the 365th step, but there is no concept of ordering of events within steps and we can't have fractional steps. A discrete time simulation can never move its clock forward by half-a-tick. As long as the steps can be assumed to be the same duration, there is no need to worry about what the duration actually is. This model is very common in the traditional social networks world. Sociometric survey data may aggregated into a set of weekly network "panels", each of which

is thought of as a discrete time step in the evolution of the network. We ignore the exact timing of what minute each survey was completed, so that we can compare the week-to-week dynamics.

5.2 You might be continuous if...

In a *continuous* model, measurements are thought of as taking place at an instantaneous point in time (as precisely as can be reasonably measured). Events may have specific durations, but they will almost never be integers. Instead of being present in week 1 and absent in week 2 a relationship starts on Tuesday at 7:45 PM and ends on Friday at 10:01 AM. Continuous time models are useful when the the ordering of events is important. It still may be useful to represent observations in panels or measure time in integer units, but we must assume that the state of the network could have changed between our observation at noon on Friday of week 1 and noon on Friday of week 2.

5.3 Comparing models

Although underlying data model for the `networkDynamic` package is continuous time, discrete time models can easily be represented. But it is important to be clear about what model you are using when interpreting measurements. For example, the `activate.vertex` methods can be called using an `onset=t` and `terminus=t+1` style, or an `at=t` style (which converts internally to `onset=t` , terminus=t). Here are several ways of representing the similar time information for an edge lasting two time steps:

```
> disc <- network.initialize(2)
> disc[1,2]<-1
> activate.edges(disc,onset=4,terminus=6) # terminus = t+1
> is.active(disc,at=4,e=1)
```

```
[1] TRUE
```

```
> is.active(disc,at=5,e=1)
```

```
[1] TRUE
```

```
> is.active(disc,at=6,e=1)
```

```
[1] FALSE
```

Remember that the edge is not active at time 6, because we specified that it is only active *until* time 6. And since we are thinking of this as a discrete network, we shouldn't ask if the edge is active at `t=5.5` (but it is).

```
> is.active(disc,at=5.5,e=1)

[1] TRUE
```

If we really wanted it to be active at time 6, we'd have to think of it as a continuous network and add on a tiny smidgen of time [1].

```
> cont <- network.initialize(2)
> cont[1,2]<-1
> activate.edges(cont,onset=3.0,terminus=6.0001)
> is.active(cont,at=4,e=1)

[1] TRUE

> is.active(cont,at=6,e=1)

[1] TRUE

> is.active(cont,at=6.5,e=1)

[1] FALSE
```

We could also chose to represent each measurement as the point in time at which the edge was observed.

```
> point <- network.initialize(2) # continuous waves
> point[1,2]<-1
> activate.edges(point,at=4)
> activate.edges(point,at=5)
> is.active(point,at=4,e=1)

[1] TRUE

> is.active(point,at=4.5,e=1) # this doesn't makes sense

[1] FALSE

> is.active(point,at=4,e=1)

[1] TRUE
```

In short, `networkDynamic` provides some great tools, but you need to think carefully about how time is measured in your data to get correct results.

[1] Sometimes a tiny bit of time can get added on due to floating point rounding errors. In rare cases this causes problems in spell comparisons where spells don't match even though it seems they should. This happens because many decimal numbers do not have exact binary equivalents. For example, 1.0-0.9-0.1 = -2.775558e-17 , not 0 as we might expect. So according to the rules of floating point math, 3.6125 != (289*0.0125).

6 Show me how it was: extracting static views of dynamic networks

Because working with spells correctly can be complicated, the package provides utility methods for dynamic versions of common network operations. View the help page at `?network.extensions` for full details and arguments.

6.1 Testing for activity

As is probably already apparent, the activity range of a vertex, set of vertices, edge, or set of edges can be tested using the `is.active` method by including a time range and list of vertexIDs or edgeIDs to check.

```
> is.active(triangle, onset=1, length=1,v=2:3)
```

```
[1]  TRUE FALSE
```

```
> is.active(triangle, onset=1, length=1,e=get.edgeIDs(triangle,v=1))
```

```
[1] TRUE
```

6.2 Listing active elements

Depending on the end use, a more convenient way to express these queries might be to use utility functions to retrieve the ids of the network elements of interest that are active for that time range.

```
> get.edgeIDs.active(triangle, onset=2, length=1,v=1)
```

```
[1] 1
```

```
> get.neighborhood.active(triangle, onset=2, length=1,v=1)
```

```
[1] 2
```

```
> is.adjacent.active(triangle,vi=1,vj=2,onset=2,length=1)
```

```
[1] TRUE
```

These methods of course accept the same additional arguments as their `network` counterparts.

6.3 Are regular network objects active?

What happens when we ask about the activity of a regular `network` object? Or what if only some vertices or edges in a `networkDynamic` object have activity attributes defined? Many functions include the `active.default` parameter for controlling how elements without spells should be treated. If the parameter is not explicitly given (`active.default=TRUE`), they will behave as if they are active from `-Inf` to `Inf`.

```
> static<-network.initialize(3)
> is.active(static,at=100,v=1:3)
```

```
[1] TRUE TRUE TRUE
```

```
> is.active(static,at=100,v=1:3,active.default=FALSE)
```

```
[1] FALSE FALSE FALSE
```

```
> dynamic<-activate.vertices(static,onset=0,terminus=200,v=2)
> is.active(dynamic,at=100,v=1:3)
```

```
[1] TRUE TRUE TRUE
```

```
> is.active(dynamic,at=100,v=1:3,active.default=FALSE)
```

```
[1] FALSE  TRUE FALSE
```

The `active.default` parameter doesn't alter the activity of elements that have been explicitly deactivated and are represented by the "null spell" (Inf,Inf).

```
> inactive<-network.initialize(2)
> deactivate.vertices(inactive,onset=-Inf,terminus=Inf,v=2)
> is.active(inactive,onset=Inf,terminus=Inf,v=1:2,active.default=TRUE)
```

```
[1]  TRUE FALSE
```

6.4 Basic descriptives

In some contexts, especially writing simulations on a network that can work in both discrete and continuous time, it may be important to know all the time points at which the structure of the network changes. The package includes a function `get.change.times` that can return a list of times for the entire network, or edges and vertices independently:

```
> get.change.times(triangle)

[1]  1  2  3  4  5  6 10

> get.change.times(triangle,vertex.activity=FALSE)

[1] 1 2 3 4 6

> get.change.times(triangle,edge.activity=FALSE)

[1]  1  4  5 10
```

We have also implemented dynamic versions of the basic network functions `network.size` and `network.edgecount` which accept the standard activity parameters:

```
> network.size.active(triangle,onset=2,terminus=3)

[1] 2

> network.edgecount.active(triangle,at=5)

[1] 1
```

6.5 Collapsing a network vs. extracting it

We've already introduced the `network.extract` function which can extract a sub-range of time from a `networkDynamic` and return it as a `networkDynamic`.

```
> get.change.times(triangle)

[1]  1  2  3  4  5  6 10
```

```
> network.edgecount(triangle)

[1] 3

> notflat <- network.extract(triangle,onset=1,terminus=3,trim.spells=TRUE)
> is.networkDynamic(notflat)

[1] TRUE

> network.edgecount(notflat) # did we lose edge2?

[1] 1

> get.change.times(notflat)

[1] 1 2 3
```

By default, the `network.extract` function returns a `networkDynamic` object with the subset of edges in the original network that are active during the query period. The `trim.spells` parameter tells it to take the more computationally expensive step of actually modifying the activity spells in all of the network elements to trim them to the specified range.

There is also a `network.collapse` function which extracts the appropriate range and returns a static `network` object with the timing information removed.

```
> flat <-network.collapse(triangle,onset=1,terminus=3)
> is.networkDynamic(flat)

[1] FALSE

> get.change.times(flat)

numeric(0)

> network.edgecount(flat)

[1] 1

> list.edge.attributes(flat)
```

```
[1] "activity.count"    "activity.duration" "na"

> flat%e%'activity.count'

[1] 2

> flat%e%'activity.duration'

[1] 1
```

The `network.collapse` function also adds `.count` and `.duration` attributes to the vertices and edges to give a crude summary of the timing information that has been removed. However, the duration information does not take into account possible censoring of ties at the beginning and end of the network observation time period.

6.6 Wiping the slate: removing activity information

Most `network` methods will ignore the timing information on a `networkDynamic` object. However, there may be situations where it is desirable to remove all of the timing information attached to a `networkDynamic` object. (Note: this is not the same thing as deactivating elements of the network.) This can be done using the `delete.edge.activity` and `delete.vertex.activity` functions which accept arguments to specify which elements should have the timing information deleted.

```
> delete.edge.activity(triangle)
> delete.vertex.activity(triangle)
> get.change.times(triangle)

numeric(0)

> get.vertex.activity(triangle)

[[1]]
     [,1] [,2]
[1,] -Inf  Inf

[[2]]
     [,1] [,2]
[1,] -Inf  Inf

[[3]]
     [,1] [,2]
[1,] -Inf  Inf
```

Although 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`.

6.7 Differences between "any" and "all' aggregation rules

In addition to the point-based (`at` syntax) or unit interval (`length=1`) activity tests and extraction operations used in most examples so far, the methods also support the idea of a "query spell' specified using the same onset and terminus syntax. So it is also possible (assuming it makes sense for the network being studied) to use `length=27.52` or `onset=0, terminus=256`.

Querying with a time range does raise an issue: how should we handle situations where edges or vertices have spells that begin or end part way through the query spell? Although other potential rules have been proposed, the methods currently include a `rule` argument that can take the values of `any` (the default) or `all`. The former returns elements if they are active for any part of the query spell, and the later only returns elements if they are active for the entire range of the query spell.

```
> query <- network.initialize(2)
> query[1,2] <-1
> activate.edges(query, onset=1, terminus=2)
> is.active(query,onset=1,terminus=2,e=1)


[1] TRUE


> is.active(query,onset=1,terminus=3,rule='all',e=1)


[1] FALSE


> is.active(query,onset=1,terminus=3,rule='any',e=1)


[1] TRUE
```

7 Squooshing data into networkDynamic objects

Obviously for most non-trivial data-sets it doesn't make sense to write out long lists of each edge and vertex to be added and removed. The package includes some handy conversion tools for moving from some common representations of network dynamics to a `networkDynamic` object.

Currently, the `networkDynamic()` conversion function importing edge and vertex timing information from the following formats:

lists of networks 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.

spells A matrix or data.frame of spells specifying edge timing. Assumed to be `[onset,terminus,vertex.id]` for vertices and `[onset,terminus,tail vertex.id, head vertex.id]` for edges.

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,vertex.id]` for vertices and `[toggle time, tail vertex id of the edge, head vertex id of the edge]` for edges.

changes Like toggles, but with an additional `direction` column indicating `1` if the toggle should change the element state to active and `0` if it should be deactivated.

Please see `?networkDynamic` for full parameter explanations. The sections below give some more in-depth examples.

7.1 But my data are panels of network matrices...

Researchers frequently have network data in the form of network panels or "stacks" of network matrices. The `networkDynamic` package includes one such classic dynamic network data-set in this format: Newcomb's Fraternity Networks. The data are 14 panel observations of friendship preference rankings among fraternity members in a 1956 sociology study. (For more details, see `?newcomb`.) This network is a useful example because it has edge weights that change over time and the `newcomb.rank` version has asymmetric rank choice ties.[2]

```
> require(networkDynamic)
> data(newcomb)             # load the data
> length(newcomb)   # how many networks?

[1] 14

> is.network(newcomb[[1]]) # is it really a network?

[1] TRUE

> as.sociomatrix(newcomb[[1]]) # peek at sociomatrix
```

[2] Although this release of the networkDynamic package supports dynamic edge attributes, the import utilities yet don't support edge wights.

```
   1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
1  0 1 0 0 0 1 0 0 0  0  1  0  1  1  1  1  1
2  1 0 0 1 0 0 1 0 0  0  0  1  1  0  1  1  1
3  0 0 0 1 1 0 0 0 1  1  1  1  0  0  1  0  1
4  0 1 0 0 0 1 1 0 0  1  1  0  1  0  0  1  1
5  0 0 0 1 0 0 0 1 1  1  1  1  0  0  1  0  1
6  1 0 0 1 0 0 0 1 1  0  0  1  1  0  0  1  1
7  0 1 0 1 0 1 0 1 0  0  1  1  0  0  0  1  1
8  0 1 0 1 0 1 0 0 0  1  1  1  1  0  0  0  1
9  1 0 1 0 0 0 1 0 0  1  1  1  0  1  0  0  1
10 1 0 0 0 0 1 1 0 1  0  0  1  0  0  1  1  1
11 0 1 1 1 1 0 0 0 1  0  0  1  0  0  0  1  1
12 0 0 1 1 1 0 1 0 0  1  1  0  0  1  0  0  1
13 1 0 0 1 1 1 0 0 0  1  1  0  0  0  1  0  1
14 0 1 1 1 0 0 1 0 1  1  0  1  0  0  1  0  0
15 0 0 1 1 1 0 0 0 1  1  1  1  0  0  0  0  1
16 1 0 0 1 0 0 0 0 1  0  1  1  0  1  1  0  1
17 0 0 0 1 1 0 1 0 1  1  1  1  1  0  0  0  0
```

```
> newcombDyn <- networkDynamic(network.list=newcomb)  # make dynamic
```

```
Neither start or onsets specified, assuming start=0
Onsets and termini not specified, assuming each network in network.list should have a discre
Argument base.net not specified, using first element of network.list instead
Created net.obs.period to describe network
 Network observation period info:
  Number of observation spells: 1
  Maximal range of observations: 0 to 14
  Temporal mode: discrete
  Time unit: step
  Suggested time increment: 1
```

```
> get.change.times(newcombDyn)
```

```
 [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
```

When converting panel data in this form, `as.networkDynamic` assumes that the panels should be assigned times of unit intervals starting at t=0, so the first panel is given the spell [0,1), the second [1,2), etc. This is important because if you use "at" query syntax the time does not correspond to the panel index.

```
> all(as.sociomatrix(newcomb[[5]]) ==
+       as.sociomatrix(network.extract(newcombDyn,at=5)))
```

```
[1] FALSE
```

```
> all(as.sociomatrix(newcomb[[5]]) ==
+       as.sociomatrix(network.extract(newcombDyn,at=4)))
```

```
[1] TRUE
```

```
>
```

If this isn't consistent with how you would like to model your data, you can use the `onsets` and `termini` parameters to provide timings for each of the panels. This is also useful if we want to be explicit about the gap in observations due to the missing week 9.

```
> newcombGaps <- networkDynamic(network.list=newcomb,
+                       onsets=c(1:8,10:15),termini=c(2:9,11:16))
```

```
Argument base.net not specified, using first element of network.list instead
Created net.obs.period to describe network
 Network observation period info:
  Number of observation spells: 14
  Maximal range of observations: 1 to 16
  Temporal mode: continuous
  Time unit: unknown
  Suggested time increment: NA
```

```
> get.vertex.activity(newcombGaps)[[1]] # peek at spells for v1
```

```
     [,1] [,2]
[1,]    1    9
[2,]   10   16
```

We can also store some descriptive meta-data for the network:

```
> nobs <-get.network.attribute(newcombGaps,'net.obs.period')
> names(nobs)
```

```
[1] "observations"   "mode"           "time.increment" "time.unit"
```

```
> nobs$'time.unit'<-'week'
> nobs$'mode'<-'discrete'
> set.network.attribute(newcombGaps,'net.obs.period',nobs)
```

7.2 Converting from toggles.

Sometimes dynamic network data from a simulation process arrives in an efficient "toggle" format. The edge dynamics of the network can be expressed as a three-column matrix giving simply the time at which an edge changes, and the vertices at either end of the edge. Because it doesn't say if the edge is turned on or off (see the "changes" format for that) we also need an initial network to give the starting state for each of the edges.

Usually this kind of input would come from the low-level output of a simulation, but we can create a crude synthetic data-set to demonstrate the conversion. Lets say we have a network of size 10, and at each time step we want a single randomly chosen edge to turn on or off, and we will do this 1000 times.

```
> toggles <-cbind(time=1:1000,
+                     tail=sample(1:10,1000,replace=TRUE),
+                     head=sample(1:10,1000,replace=TRUE))
> head(toggles) # peek at begining
```

```
     time tail head
[1,]    1    7   10
[2,]    2    6    3
[3,]    3    8    2
[4,]    4    5    3
[5,]    5    1    2
[6,]    6    9    2
```

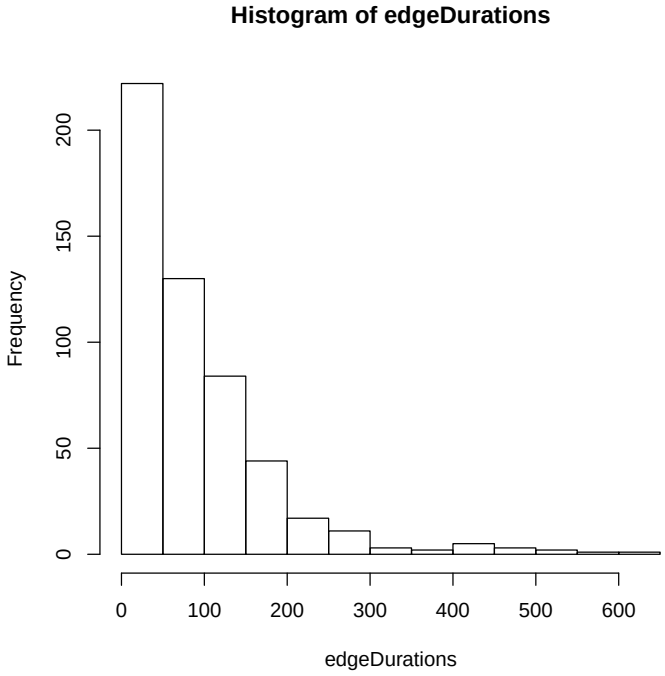
```
> empty<-network.initialize(10,loops=TRUE) # to define initial states
> randomNet <-networkDynamic(base.net=empty,edge.toggles=toggles)
```

```
Created net.obs.period to describe network
 Network observation period info:
  Number of observation spells: 1
  Maximal range of observations: 1 to 1000
  Temporal mode: discrete
  Time unit: step
  Suggested time increment: 1
```

We converted the toggles using the `edge.toggles`argument to `networkDynamic()`. If we wanted the vertices to flip on and off as well, the function also accepts a `vertex.toggles` argument. Once the toggles have been translated into a network format, we can do things like look at the distribution of edge durations created by our crude model.

```
> edgeDurations<-get.edge.activity(randomNet,as.spellList=TRUE)$duration
> hist(edgeDurations)
> summary(edgeDurations)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.0    25.0    63.0    88.1   123.0   613.0
```



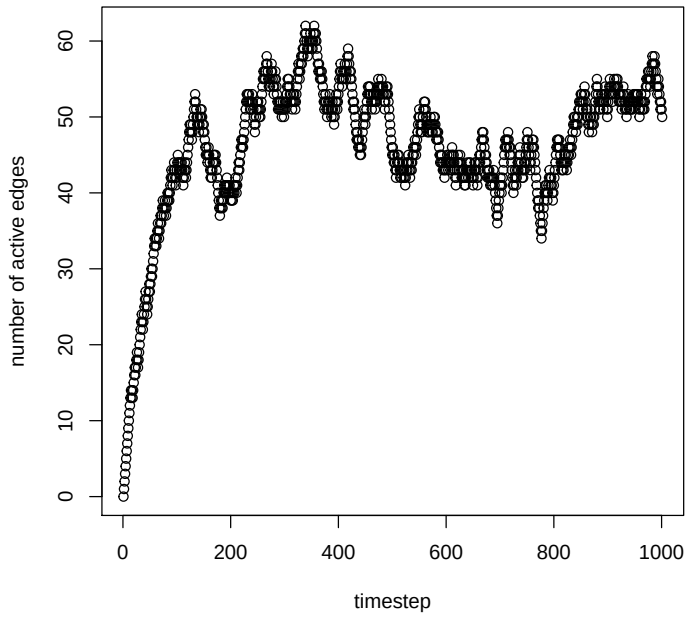
Our "simulation" was long enough that we should see a nice long-tailed distribution of edge activity durations. However, when we check the terminus censoring,

```
> sum(get.edge.activity(randomNet,as.spellList=TRUE)$terminus.censored)
```

```
[1] 50
```

we can see that there are around 50 edges that had not ended at the termination of our simulation period, so we should interpret the mean durations with caution. It is also interesting to consider how we might have skewed the edge durations because we started with an empty network. To examine this we can construct a time-series of the number of active edges in the network by repeatedly applying the dynamic edge-counting function at each time point.

```
> nEdgesActive<-sapply(0:1000,
+                function(t){network.edgecount.active(randomNet,at=t)})
> plot(nEdgesActive,xlab='timestep',ylab='number of active edges')
```



This shows us that it took a few hundred time steps of "burn in" for the network to move from its initial extreme (the zero-edges condition) to a sort of equilibrium state. After which there were enough active edges that some of them started getting toggled off again, and it continues to wobble around a value of about 50 edges until the end.

7.3 Batteries and tergm example not included

Unfortunately we can't include a real live `tergm` model example here, because the `tergm` package depends on `networkDynamic`, and we don't want to create a circular package dependency. But are nice examples located in the `term` and `ndtv` package vignettes.

7.4 Converting a stream of spells: McFarland's classroom interactions

Not surprisingly, the `networkDynamic()` function can create `networkDynamic` objects from a matrix of activity spells stored in a `data.frames`. It assumes that the first two columns give the onset and terminus of a spell, and the third and forth columns correspond to the network indices of the ego and alter vertices for that dyad. Multiple spells per dyad are expressed by multiple rows. In the following example, we read some tabular data describing arc relationships out of

example text files. For more information about the data-set (which also exists as a `networkDynamic` object) see `?cls33_10_16_96`.

```
> vertexData <-read.table(system.file('extdata/cls33_10_16_96_vertices.tsv',
+             package='networkDynamic'),,header=T)
> vertexData[1:5,] # peek
```

```
  vertex_id data_id start_minute end_minute sex     role
1         1  122658            0         49   F grade_11
2         2  129047            0         49   M grade_11
3         3  129340            0         49   M grade_11
4         4  119263            0         49   M grade_12
5         5  122631            0         49   F grade_12
```

```
> edgeData <-read.table(system.file('extdata/cls33_10_16_96_edges.tsv',
+             package='networkDynamic'),header=T)
> edgeData[1:5,] # peek
```

```
  from_vertex_id to_vertex_id start_minute end_minute weight interaction_type
1             14           12        0.125      0.125      1           social
2             12           14        0.250      0.250      1           social
3             18           12        0.375      0.375      1         sanction
4             12           18        0.500      0.500      1         sanction
5              1           12        0.625      0.625      1         sanction
```

Now that the spell data is loaded in, we need to form it into a network. We want to use the `vertex_id`, `start_minute` and `end_minute` from the vertex data, and the `from_vertex_id`, `to_vertex_id`, `start_minute` and `end_minute` from the edge data. Since the columns are not in the order that we want, we re-order the column indices when passing to the `edge.spells` and the `vertex.spells` arguments of `networkDynamic`.

```
> classDyn <- networkDynamic(vertex.spells=vertexData[,c(3,4,1)],
+                            edge.spells=edgeData[,c(3,4,1,2)])
```

```
Initializing base.net of size 20 imputed from maximum vertex id in edge records
Created net.obs.period to describe network
 Network observation period info:
  Number of observation spells: 1
  Maximal range of observations: 0 to 49
  Temporal mode: continuous
  Time unit: unknown
  Suggested time increment: NA
```

The conversion printed out summary of the (optional) network observation period attribute (`net.obs.period`) which tells us that it made a guess that this was a continuous time network. And if we peek at the change times of the network, it appears that it this is probably accurate.

```
> get.change.times(classDyn)[1:10]
```

```
 [1] 0.000 0.125 0.250 0.375 0.500 0.625 0.750 0.875 1.000 1.167
```

We can also store the time units for the network, just in case we (or someone else) needs to know them later

```
> nobs <-get.network.attribute(classDyn,'net.obs.period')
> names(nobs)
```

```
[1] "observations"   "mode"           "time.increment" "time.unit"
```

```
> nobs$'time.unit'<-'minutes'
> set.network.attribute(classDyn,'net.obs.period',nobs)
```

The original data include some attribute information for the vertices which we'd like to add, but first we need to check if they are dynamic or not. We will assume that if each `vertex_id` has only one row, the attributes must have only one spell associated with them and can be treated as static. We also must make sure the `vertex_ids` are in order. Since `read.table` creates a `data.frame` object, we explicitly convert factors to character values.

```
> nrow(vertexData)==length(unique(vertexData$vertex_id))
```

```
[1] TRUE
```

```
>
```

Looks good! Lets load 'em up...

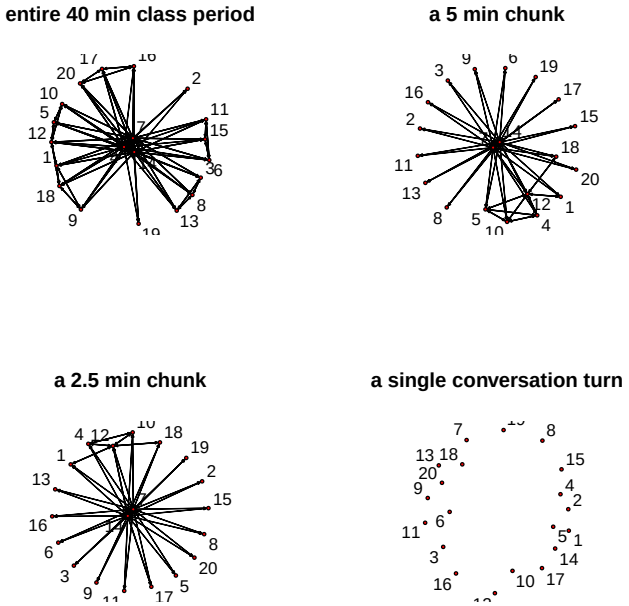
```
> set.vertex.attribute(classDyn,"data_id",vertexData$data_id)
> set.vertex.attribute(classDyn,"sex",as.character(vertexData$sex))
> set.vertex.attribute(classDyn,"role",as.character(vertexData$role))
```

To run standard network measures we will need to first "bin" or "slice" the network up into static networks. Although there is not yet a utility function for this operation, we will convert the classroom data into series of networks, each of which aggregates 5 minutes of streaming interactions. We do this by creating a list of times, and then using the `lapply` function to apply `network.extract` for each of those times.

```
> startTimes <-seq(from=0,to=40,by=5) # make a list of times
> classNets <- lapply(startTimes, function(t){
+   network.extract(classDyn,onset=t,terminus=t+5)})
> classMats <- lapply(classNets,as.sociomatrix) # make into matrices
> classDensity <- sapply(classNets, network.density)
> plot(classDensity,type='l',xlab='network slice #',ylab='density')
```

Since this data-set consists of continuous time streams of relational information, the choice of 5 minutes is fairly arbitrary. Other durations will reveal dynamics at various timescales.

```
> par(mfrow=c(2,2))   # show multiple plots
> plot(network.extract(
+   classDyn,onset=0,length=40,rule="any"),
+   main='entire 40 min class period',displaylabels=T)
> plot(network.extract(
+   classDyn,onset=0,length=5,rule="any"),
+   main='a 5 min chunk',displaylabels=T)
> plot(network.extract(
+   classDyn,onset=0,length=2.5,rule="any"),
+   main='a 2.5 min chunk',displaylabels=T)
> plot(network.extract(
+   classDyn,onset=0,length=.1,rule="any"),
+   main='a single conversation turn',displaylabels=T)
```



8 Persistent IDs

As has already been mentioned, the standard vertex and edge ids used in the network package are indices, so they must change when the network size changes during an extraction operation [3]. So how can we follow a specific network element through a series of slicing and dicing operations? Since v0.5, the networkDynamic package supports defining an (optional) "persistent id" (pid) for edges and vertices.

Once a persistent id has been defined, the functions `get.vertex.id()` and `get.vertex.pid` can be used to translate between the normal ids and the pids. For edges, the functions are named `get.edge.id` and `get.edge.pid`. Lets look at an example where we find the original vertices corresponding to vertices in smaller extracted net.

```
> haystack<-network.initialize(30)
> activate.vertices(haystack,v=10:20)
```

Now hide some needles in the haystack...

```
> set.vertex.attribute(haystack,'needle',TRUE,v=sample(10:20,2))
```

[3] In the case of vertex ids, they may also change during vertex deletions, or additions to the first mode of a bipartite network

... make up an id for the vertices, and define that it will be our persistent id.

```
> set.vertex.attribute(haystack,'hayId',paste('straw',1:30,sep=''))
> set.network.attribute(haystack,'vertex.pid','hayId')
```

Lets find the needles in the new stack after some hay has been removed over time

```
> newstack<-network.extract(haystack,at=100,active.default=FALSE)
> network.size(newstack)
```

```
[1] 11
```

```
> needleIds <-which(get.vertex.attribute(newstack,'needle'))
> needleIds
```

```
[1] 3 5
```

What are the pids of vertices with needles? Which vertices are the corresponding ones in the original haystack?

```
> get.vertex.pid(newstack,needleIds)
```

```
[1] "straw12" "straw14"
```

```
> get.vertex.id(haystack,get.vertex.pid(newstack,needleIds))
```

```
[1] 12 14
```

In the example above, we made up a new id, but if the data set already had some type of unique identifier for vertices, we could have used it instead. In some cases it might be tempting to use the `vertex.names` attributes of networks as a persistent id without checking that it is unique. This can cause problems if vertices are added or deleted.

```
> net<-network.initialize(3)
> add.vertices(net,1)
> delete.vertices(net,2)
> # notice the NA value
> as.matrix(net)
```

```
     1 3 <NA>
1    0 0    0
3    0 0    0
<NA> 0 0    0
```

To make life easier, we can just indicate that a unique set of `vertex.names` can safely be used as a vertex.pid by setting `vertex.pid` to `'vertex.names'`. This has the advantage of not adding an extra attribute that needs to be carried around, and the pids will appear as the labels.

```
> net<-network.initialize(3)
> set.network.attribute(net,'vertex.pid','vertex.names')
> add.vertices(net,1,vertex.pid='4')
> add.vertices(net,1)
> delete.vertices(net,2)
> as.matrix(net)
```

```
             1 3 4 53ce56c09f9d
1            0 0 0            0
3            0 0 0            0
4            0 0 0            0
53ce56c09f9d 0 0 0            0
```

Notice that when we added vertices in the first case we explicitly included in a vertex pid for the new vertex. In the second case, we didn't specify a pid, so it made up a messy one to make sure they stayed unique.

The function initialize.pids can also 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 meaningless but unique pseudo-random hex strings using the `tempfile` function (something like `'4ad912252bc2'`) These are also the types of new pids that will be created if add.vertices is called in a network with a vertex.pid defined, as in the example above. It is a good idea to define pids after a network object as been constructed and before any extractions are performed.

```
> net<-network.initialize(3)
> add.edges(net,tail=1:2,head=2:3)
> initialize.pids(net)
> net%v%'vertex.pid'
```

```
[1] "53ce427c8786" "53ce1a5660af" "53ce6629d3c4"
```

```
> net%e%'edge.pid'
```

```
[1] "53ce50e0766" "53ce54fe49c"
```

The edge pids can be useful in looking up edges if vertex deletions cause the ids of the edge's vertices to be permuted.

9 Transforming networkDynamic objects to other representations

Great, I got all my data into your magic format, now how do I get it out again?

9.1 Converting to lists of spells

As we've already demonstrated, for number of types of analysis it is useful to be able to dump the edge timing information into a "flat" tabular representation.

```
> newcombEdgeSpells<-get.edge.activity(newcombDyn,as.spellList=TRUE)
> newcombEdgeSpells[1:5,] # peek at the beginning
```

```
  onset terminus tail head onset.censored terminus.censored duration edge.id
1     0        1    2    1          FALSE             FALSE        1       1
2     2        3    2    1          FALSE             FALSE        1       1
3     6        9    2    1          FALSE             FALSE        3       1
4    11       12    2    1          FALSE             FALSE        1       1
5     0       14    6    1          FALSE             FALSE       14       2
```

The first two columns of the spell matrix give the network indices of the vertices involved in the edge, and the next two give the onset and terminus for the spell. The `right.censored` column indicates if a statistical estimation process using this spell list should assume that the entire duration of the edge's activity is included or that it was partially censored by the observation window. Note that the `duration` column gives the total duration for the *specific spell* of the edge,(not the entire edge duration) and an edge may appear in multiple rows. Because this may be the most common type of conversion people need to do, we also created an `as.data.frame` alias `get.edge.activity` function.

```
> newcombEdgeSpells<-as.data.frame(newcombDyn)
> newcombEdgeSpells[1:5,] # peek at the beginning
```

```
  onset terminus tail head onset.censored terminus.censored duration edge.id
1     0        1    2    1          FALSE             FALSE        1       1
2     2        3    2    1          FALSE             FALSE        1       1
3     6        9    2    1          FALSE             FALSE        3       1
4    11       12    2    1          FALSE             FALSE        1       1
5     0       14    6    1          FALSE             FALSE       14       2
```

Of course, these methods only return information about the edge dynamics so there is a corresponding `get.vertex.activity` function.

```
> vertSpells <- get.vertex.activity(newcombDyn,as.spellList=TRUE)
> vertSpells[1:5,]
```

```
  onset terminus vertex.id onset.censored terminus.censored duration
1     0       14         1          FALSE             FALSE       14
2     0       14         2          FALSE             FALSE       14
3     0       14         3          FALSE             FALSE       14
4     0       14         4          FALSE             FALSE       14
5     0       14         5          FALSE             FALSE       14
```

Which is not so exciting in this example, since we don't have any vertex dynamics.

9.2 Converting to a list of networks or matrices

We are still missing the `get.slice.networks` function. It should have an example here. Instead we can use `lapply` to extract a list of several non-overlapping unit slices from the random network we created a while back, and them print them out as matrices.

```
> onsets<-0:2
> slices<-lapply(onsets,function(t){
+           network.extract(randomNet,onset=t,terminus=t+1)
+         })
> is.network(slices[[1]])

[1] TRUE

> lapply(slices,as.matrix) # print 'em out as matrices

[[1]]
   1 2 3 4 5 6 7 8 9 10
1  0 0 0 0 0 0 0 0 0  0
2  0 0 0 0 0 0 0 0 0  0
3  0 0 0 0 0 0 0 0 0  0
4  0 0 0 0 0 0 0 0 0  0
5  0 0 0 0 0 0 0 0 0  0
6  0 0 0 0 0 0 0 0 0  0
7  0 0 0 0 0 0 0 0 0  0
8  0 0 0 0 0 0 0 0 0  0
9  0 0 0 0 0 0 0 0 0  0
10 0 0 0 0 0 0 0 0 0  0

[[2]]
   1 2 3 4 5 6 7 8 9 10
1  0 0 0 0 0 0 0 0 0  0
2  0 0 0 0 0 0 0 0 0  0
3  0 0 0 0 0 0 0 0 0  0
```

```
4  0 0 0 0 0 0 0 0 0  0
5  0 0 0 0 0 0 0 0 0  0
6  0 0 0 0 0 0 0 0 0  0
7  0 0 0 0 0 0 0 0 0  1
8  0 0 0 0 0 0 0 0 0  0
9  0 0 0 0 0 0 0 0 0  0
10 0 0 0 0 0 0 0 0 0  0

[[3]]
   1 2 3 4 5 6 7 8 9 10
1  0 0 0 0 0 0 0 0 0  0
2  0 0 0 0 0 0 0 0 0  0
3  0 0 0 0 0 0 0 0 0  0
4  0 0 0 0 0 0 0 0 0  0
5  0 0 0 0 0 0 0 0 0  0
6  0 0 1 0 0 0 0 0 0  0
7  0 0 0 0 0 0 0 0 0  1
8  0 0 0 0 0 0 0 0 0  0
9  0 0 0 0 0 0 0 0 0  0
10 0 0 0 0 0 0 0 0 0  0
```

10 Dynamic attributes

An important tool for working with dynamic networks is the ability to represent time-varying attributes of networks, vertices (changing properties) and edges (changing weights). In the `networkDynamic` package we refer to these as dynamic attributes or "TEAs" (Temporally Extended Attributes). A TEA is a standard edge, vertex, or network attribute that has a name ending in `.active` and carries meta-data regarding its state over time. We store the TEAs as a two-part list, where the first part is a list of values, and the second is a spell matrix where each row gives the onset and terminus of activity for the corresponding value. See `?activate.vertex.attribute` for the full specification of Temporally Extended Attributes. Of course we try to hide most of this as much as possible using a set of accessor functions.

10.1 Activating TEA attributes

The functions for creating TEA attributes are named similarly to the regular functions for manipulating network, vertex, and edge attributes but they also accept the spell-related arguments (onset, terminus, at, length).

```
> net <-network.initialize(5)
> activate.vertex.attribute(net,"happiness", -1, onset=0,terminus=1)
> activate.vertex.attribute(net,"happiness", 5, onset=1,terminus=3)
```

```
> activate.vertex.attribute(net,"happiness", 2, onset=4,terminus=7)
> list.vertex.attributes(net)    # what are they actually named?

[1] "happiness.active" "na"               "vertex.names"

> get.vertex.attribute.active(net,"happiness",at=2)

[1] 5 5 5 5 5

> get.vertex.attribute(net,"happiness.active",unlist=FALSE)[[1]]

[[1]]
[[1]][[1]]
[1] -1

[[1]][[2]]
[1] 5

[[1]][[3]]
[1] 2


[[2]]
     [,1] [,2]
[1,]    0    1
[2,]    1    3
[3,]    4    7
```

Notice that when using the `activate.vertex.attribute` and `get.vertex.attribute.active` functions we don't have to include the ".active" part of the attribute name, it handles that on its own. When we used the regular `get.vertex.attribute` function to peek at the attribute of the first vertex we can see the list of values (-1,5,2) and the spell matrix. We also had to include the `unlist=FALSE` argument so that it didn't mangle the list object by smooshing into a vector when it was returned.

There are similar activation functions for edge and network-level attributes.

```
> activate.network.attribute(net,'colors',"red",
+                            onset=0,terminus=1)
> activate.network.attribute(net,'colors',"green",
+                            onset=1,terminus=5)
> add.edges(net,tail=c(1,2,3),head=c(2,3,4)) # need edges to activate-
> activate.edge.attribute(net,'weight',c(5,12,7),onset=1,terminus=3)
> activate.edge.attribute(net,'weight',c(1,2,1),onset=3,terminus=7)
```

Since we didn't give the edges themselves timing info, they will be assumed to be always active. But we've specified that the 'weight' of the edges should vary over time.

10.2 Querying TEA attributes

What happens when there are no values defined? When we activate the vertex attributes, we left a gap in the spell coverage. What if we ask for values in the time period?

```
> get.vertex.attribute.active(net,"happiness",at=3.5)
```

```
[1] NA NA NA NA NA
```

```
> get.vertex.attribute.active(net,"happiness",
+                             onset=2.5,terminus=3.5)
```

```
[1] 5 5 5 5 5
```

```
> get.vertex.attribute.active(net,"happiness",
+                             onset=2.5,terminus=3.5,rule="all")
```

```
[1] NA NA NA NA NA
```

In the first case, no values are defined so `NA` is returned. In the second case, the query spell included part of a defined value since inclusion rule defaults to '`rule='any'` and the query intersected with part of the spell associated with the value 5. We can ask it to only return values if they match the entire query spell by setting `rule='all'`, which is what happened in the third case.

The functions also permit queries that will intersect with multiple attribute values. In this case the earliest value is returned, but it also gives a warning that the value returned may not be the appropriate value for the time range.

```
> get.vertex.attribute.active(net,"happiness",onset=2.5,terminus=4.5)
```

```
[1] 5 5 5 5 5
```

```
Warning message:
In get.vertex.attribute.active(net, "happiness", onset = 2.5,
    terminus = 4.5) : Multiple attribute values matched query
    spell  for some vertices, only earliest value used
```

If we know that this behavior (returning the earliest attribute value that intersects with the query spell) is what is desired, we can suppress the warnings by specifying `rule='earliest'`.

```
> get.vertex.attribute.active(net,"happiness",onset=2.5,terminus=4.5,rule='earliest')
```

```
[1] 5 5 5 5 5
```

As might be expected, `rule='latest'` also works, but it returns the latest (most recent, largest time value) attribute intersecting with the query spell.

```
> get.vertex.attribute.active(net,"happiness",onset=2.5,terminus=4.5,rule='latest')
```

```
[1] 2 2 2 2 2
```

In many cases the user might want to aggregate the values together in some way, but that there is no way for the query function know what the correct aggregation method would be–especially if the attributes have categorical rather than numeric values. Should the results be a sum? An average? A time-weighted average? A value sampled at random? In order to handle these cases correctly, code must be designed to explicitly handle the multiple values. To facilitate this the query functions have an argument `return.tea=TRUE` which can be set so that they will return the (appropriately trimmed) TEA structure to be evaluated.

```
> get.vertex.attribute.active(net,"happiness",onset=2.5,terminus=4.5,
+                          return.tea=TRUE)[[1]]
```

```
[[1]]
[[1]][[1]]
[1] 5

[[1]][[2]]
[1] 2


[[2]]
     [,1] [,2]
[1,]    1    3
[2,]    4    7
```

If we wanted to calculate the sum value for an attribute over a particular time range

```
> sapply(get.vertex.attribute.active(net,"happiness",onset=0,terminus=7,
+                    return.tea=TRUE),function(splist){
+                       sum(unlist(splist[[1]]))
+                    })
```

```
[1] 6 6 6 6 6
```

The query syntax for network- and edge-level TEAs is similar to the vertex case `get.network.attribute.active` and `get.edge.attribute.active`. However, in keeping with the pattern established by the `network` package, `get.edge.value.active` works as an alternate.

```
> get.edge.attribute.active(net,'weight',at=2)

[1]  5 12  7

> get.edge.attribute.active(net,'weight',at=5)

[1] 1 2 1
```

There are are also functions for checking which attributes are present at any point in time (optionally excluding non-TEA attributes).

```
> list.vertex.attributes.active(net,at=2)

[1] "na"               "vertex.names"     "happiness.active"

> list.edge.attributes.active(net,at=2)

[1] "na"            "weight.active"

> list.network.attributes.active(net,at=2,dynamic.only=TRUE)

[1] "colors.active"
```

10.3 Modifying TEAs

The TEA functions are designed to maintain the appropriate sorted representation of attributes and spells even if attributes are not added in temporal order. So its possible to overwrite the attribute values.

```
> activate.vertex.attribute(net, "happiness",100, onset=0,terminus=10,v=1)
> get.vertex.attribute.active(net,"happiness",at=2)

[1] 100   5   5   5   5
```

Or set attributes to be inactive for specific time ranges and vertices.

```
> deactivate.vertex.attribute(net, "happiness",onset=1,terminus=10,v=2)
> get.vertex.attribute.active(net,"happiness",at=2)

[1] 100  NA   5   5   5
```

11 Making Lin Freeman's windsurfers gossip

For a more advanced and realistic demonstration of TEAs and, we will construct a toy rumor diffusion model. Our intention is to release packages in the near future which provide built-in functions for much of the simulation code below.

In 1988, Lin Freeman collected a month-long data-set of daily social interactions between windsurfers on California beaches (Almquist, et al , 2011), (Freeman et al , 1988). The data-set is included in `networkDynamic` and has some challenging features, including vertex dynamics (different people are present on the beach on different days) and a missing day of observation. (Run `?windsurfers` for more details).

```
> data(windsurfers)    # let's go to the beach!
> range(get.change.times(windsurfers))
```

```
[1]  0 31
```

```
> sapply(0:31,function(t){ # how many people in net each day?
+   network.size.active(windsurfers,at=t)})
```

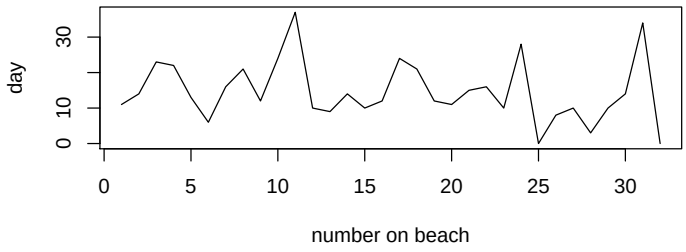
```
 [1] 11 14 23 22 13  6 16 21 12 24 37 10  9 14 10 12 24 21 12 11 15 16 10 28  0
[26]  8 10  3 10 14 34  0
```

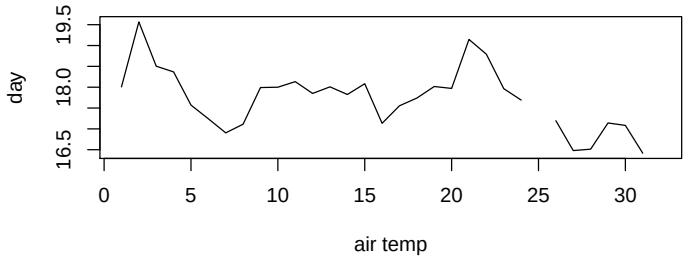
Although not directly relevant for the trivial simulation we are about to build, the `windsurfers` network object also includes some network-level dynamic attributes that give information about the weather, etc. We can extract the information as a time-series for plotting.

```
> list.network.attributes.active(windsurfers,-Inf,Inf,dynamic.only=TRUE)
```

```
[1] "atmp.active" "cord.active" "day.active"  "gst.active"  "week.active"
[6] "wspd.active" "wvht.active"
```

```
> par(mfcol=c(2,1)) # show multiple plots
> plot(sapply(0:31,function(t){ # how many people in net each day?
+   network.size.active(windsurfers,at=t)}),
+      type='l',xlab="number on beach",ylab="day"
+ )
> plot(sapply(0:31,function(t){ # how many people in net each day?
+   get.network.attribute.active(windsurfers,'atmp',at=t)}),
+      type='l',xlab="air temp",ylab="day"
+ )
> par(mfcol=c(1,1))
```





But the appropriate values will also appear in the network returned when we collapse to a specific day.

```
> day3 <-network.collapse(windsurfers,at=2)
> day3%n%'day' # what day of the week is day 3?

[1] "Saturday"

> day3%n%'atmp' # air temp?

[1] 18.50417
```

11.1 A toy diffusion model

We will create a very crude model of information transmission as an example of simulation employing dynamic attributes on a network with changing edges and vertices. We will assume that there is a "rumor" spreading among the windsurfers. At each time step, they have some probability of passing the rumor to the people they are interacting with on the beach that day. First we define a function to run the simulation:

```
> runSim<-function(net,timeStep,transProb){
+   # loop through time, updating states
+   times<-seq(from=0,to=max(get.change.times(net)),by=timeStep)
+   for(t in times){
+     # find all the people who know and are active
+     knowers <- which(!is.na(get.vertex.attribute.active(
+       net,'knowsRumor',at=t,require.active=TRUE)))
+     # get the edge ids of active friendships of people who knew
+     for (knower in knowers){
+       conversations<-get.edgeIDs.active(net,v=knower,at=t)
+       for (conversation in conversations){
+         # select conversation for transmission with appropriate prob
+         if (runif(1)<=transProb){
+           # update state of people at other end of conversations
+           # but we don't know which way the edge points so..
+           v<-c(net$mel[[conversation]]$inl,
+                  net$mel[[conversation]]$outl)
+           # ignore the v we already know
+           v<-v[v!=knower]
+           activate.vertex.attribute(net,"knowsRumor",TRUE,
+                                     v=v,onset=t,terminus=Inf)
+           # record who spread the rumor
+           activate.vertex.attribute(net,"heardRumorFrom",knower,
+                                     v=v,onset=t,length=timeStep)
+           # record which friendships the rumor spread across
+           activate.edge.attribute(net,'passedRumor',
+                     value=TRUE,e=conversation,onset=t,terminus=Inf)
+         }
+       }
+     }
+   }
+   return(net)
+ }
```

11.2 Go!

Then we set the parameters and the initial state of the network and run the simulation.

```
> timeStep <- 1  # units are in days
> transProb <- 0.2 # how likely to tell in each conversation/day
> # start the rumor out on vertex 1
> activate.vertex.attribute(windsurfers,"knowsRumor",TRUE,v=1,
+                           onset=0-timeStep,terminus=Inf)
> activate.vertex.attribute(windsurfers,"heardRumorFrom",1,v=1,
```

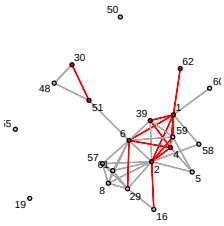
```
+                            onset=0-timeStep,length=timeStep)
> windsurfers<-runSim(windsurfers,timeStep,transProb) # run it!
```

11.3 OK, what happened?

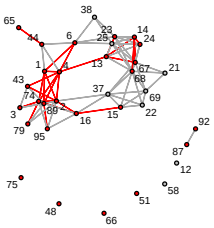
We'll make some network plots so we can get an idea of what happened.

```
> par(mfcol=c(1,2)) # show two plots side by side
> wind7<-network.extract(windsurfers,at=7)
> plot(wind7,
+      edge.col=sapply(!is.na(get.edge.value.active(wind7,
+       "passedRumor",at=7)), function(e){ switch(e+1,"darkgray","red")}),
+      vertex.col=sapply(!is.na(get.vertex.attribute.active(wind7,
+       "knowsRumor",at=7)), function(v){switch(v+1,"gray","red")}),
+      label.cex=0.5,displaylabels=TRUE,main="gossip at time 7")
> wind30<-network.extract(windsurfers,at=30)
> plot(wind30,
+      edge.col=sapply(!is.na(get.edge.value.active(wind30,
+       "passedRumor",at=30)),function(e){switch(e+1,"darkgray","red")}),
+      vertex.col=sapply(!is.na(get.vertex.attribute.active(wind30,
+       "knowsRumor",at=30)),function(v){switch(v+1,"gray","red")}),
+      label.cex=0.5,displaylabels=TRUE,main="gossip at time 30")
> par(mfcol=c(1,1))
```

gossip at time 7



gossip at time 30

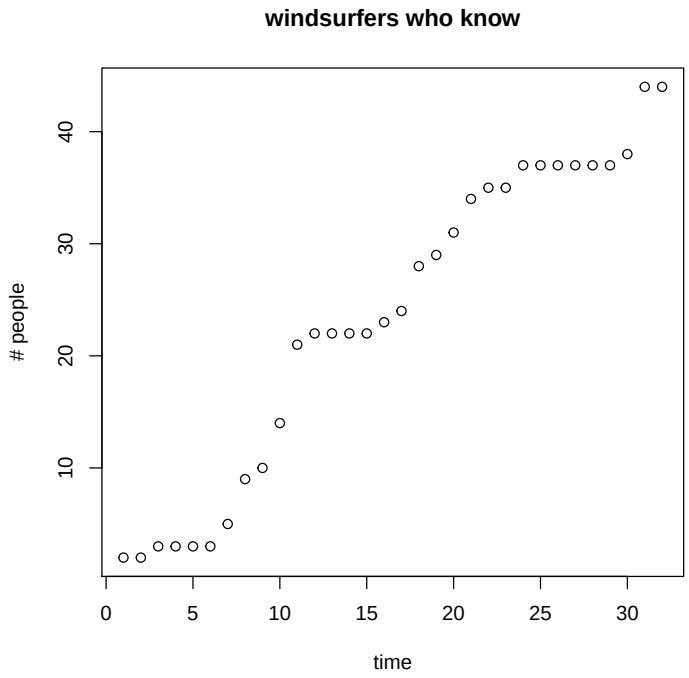


Which people heard the rumor halfway through the month? How many heard each day?

```
> get.vertex.attribute.active(windsurfers,'knowsRumor',at=15)
```

```
 [1] TRUE TRUE   NA TRUE   NA TRUE   NA   NA   NA TRUE   NA   NA TRUE   NA   NA
[16]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA TRUE
[31]   NA   NA   NA   NA TRUE   NA   NA   NA TRUE TRUE TRUE   NA   NA   NA   NA
[46]   NA   NA   NA   NA   NA TRUE   NA TRUE TRUE   NA   NA   NA   NA TRUE   NA
[61]   NA TRUE TRUE   NA TRUE   NA   NA   NA   NA   NA   NA   NA   NA TRUE TRUE
[76] TRUE   NA   NA TRUE   NA   NA   NA TRUE   NA   NA   NA   NA   NA   NA   NA
[91]   NA   NA   NA   NA   NA
```

```
> plot(sapply(0:31,function(t){
+   sum(get.vertex.attribute.active(windsurfers,'knowsRumor',at=t),
+       na.rm=TRUE)}),
+   main='windsurfers who know',ylab="# people",xlab='time'
+ )
```



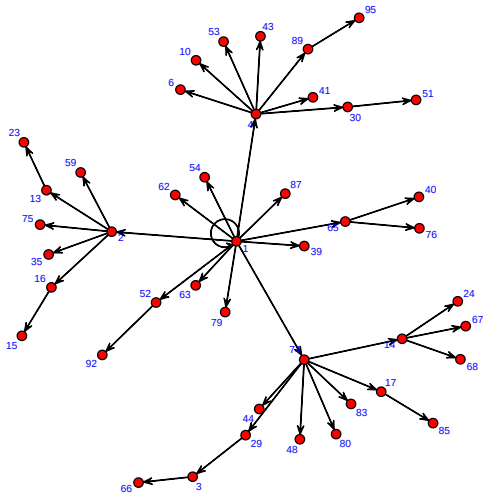
In additional to extracting values, we can do operations using the TEA attributes directly. Our simulation function recorded each time a person was told the rumor. What are the ids of the people who told person 3? On which days did person 3 hear the rumor?

```
> # pull TEA from v3, extract values from 1st part and unlist
> unlist(get.vertex.attribute.active(windsurfers,'heardRumorFrom',
+                 onset=0,terminus=31,return.tea=TRUE)[[3]][[1]])

[1] 29 74 41  1 74

> # pull TEA from v3, extract times from 2nd part and pull col 1
> get.vertex.attribute.active(windsurfers,'heardRumorFrom',
+               onset=0,terminus=31,return.tea=TRUE)[[3]][[2]][,1]

[1] 20 23 25 26 30
```

11.4 Picturing the rumor tree

We can also write a function to create a rumor transmission tree using the `heardRumorFrom` attribute in order to plot out the sequence of conversation steps that spread the gossip.

```
> transTree<-function(net){
+   # for each vertex in net who knows
+   knowers <- which(!is.na(get.vertex.attribute.active(net,
+                                          'knowsRumor',at=Inf)))
+   # find out who the first transmission was from
+   transTimes<-get.vertex.attribute.active(net,"heardRumorFrom",
+                       onset=-Inf,terminus=Inf,return.tea=TRUE)
+   # subset to only ones that know
+   transTimes<-transTimes[knowers]
+   # get the first value of the TEA for each knower
+   tellers<-sapply(transTimes,function(tea){tea[[1]][[1]]})
+   # create a new net of appropriate size
+   treeIds <-union(knowers,tellers)
+   tree<-network.initialize(length(treeIds),loops=TRUE)
+   # copy labels from original net
+   set.vertex.attribute(tree,'vertex.names',treeIds)
+   # translate the knower and teller ids to new network ids
+   # and add edges for each transmission
+   add.edges(tree,tail=match(tellers,treeIds),
+             head=match(knowers,treeIds) )
+   return(tree)
+ }
> plot(transTree(windsurfers),displaylabels=TRUE,
+      label.cex=0.5,label.col='blue',loop.cex=3)
```



We can see that the rumor started at v1, our seed vertex, which has a little loop because it infected itself.

12 Related packages and other coming attractions

The statnet team is releasing several packages that work closely with the `networkDynamic` package to provide additional features.

- `ndtv` : Network Dynamic Temporal Visualization package – like TV for your networks. The `ndtv` package creates network animations of dynamic networks stored in the `networkDynamic` format.
- `tsna` : Temporal SNA tools for measuring and doing descriptive statistics on dynamic networks stored in the `networkDynamic` format.

13 Citing networkDynamic

You can use R's built in citation function to give the citation for the package.

```
> citation(package='networkDynamic')
```

```
To cite package âĂŸnetworkDynamicâĂŹ in publications use:

  Carter T. Butts, Ayn Leslie-Cook, Pavel N. Krivitsky and Skye
  Bender-deMoll (2013). networkDynamic: Dynamic Extensions for Network
  Objects. R package version 0.5. http://statnet.org

A BibTeX entry for LaTeX users is

  @Manual{,
    title = {networkDynamic: Dynamic Extensions for Network Objects},
    author = {Carter T. Butts and Ayn Leslie-Cook and Pavel N. Krivitsky and Skye Bender-deM
    year = {2013},
    note = {R package version 0.5},
    url = {http://statnet.org},
  }
```

14 Vocabulary definitions

This is a list of terms and common function arguments giving their special meanings within the context of the `networkDynamic` package.

spell bounded interval of time describing activity period of a network element

onset beginning of spell

terminus end of a spell

length the duration of a spell

at a single time point, a spell with zero length where onset=terminus

start beginning (least time) of observation period (or series of spells)

end end (greatest time) of observation period (or series of spells)

spell list or spell matrix a means of describing the activity of a network or network element using a matrix in which one column contains the onsets and another the termini of each spell

toggle list a means of describing the activity of a network or network element using a list of times at which an element changed state ('toggled')

onset-censored when elements of a dynamic network are known to be active before start of the defined observation period, even if the onset of the spell is not known.

terminus-censored when elements of a dynamic network are known to be active after the end of the defined observation period, even if the terminus of the spell is not known.

TEA Temporally Extended Attribute: structure for storing dynamic attribute data on vertices, edges, and networks.

pid A "Persistent ID" for a vertex or edge that will remain the same despite extraction and deletion operations.

15 Complete package function listing

Below is a reference list of all the public functions included in the package `networkDynamic`

```
> cat(ls("package:networkDynamic"),sep="\n")
```

```
activate.edge.attribute
activate.edges
activate.edge.value
activate.network.attribute
activate.vertex.attribute
activate.vertices
add.edge
add.edges
add.edges.active
add.vertices
add.vertices.active
as.data.frame.networkDynamic
as.networkDynamic
as.networkDynamic.network
as.networkDynamic.networkDynamic
as.network.networkDynamic
deactivate.edge.attribute
deactivate.edges
deactivate.network.attribute
deactivate.vertex.attribute
deactivate.vertices
delete.edge.activity
delete.vertex.activity
edge.pid.check
get.change.times
get.edge.activity
get.edge.attribute.active
get.edge.id
get.edgeIDs.active
get.edge.pid
get.edges.active
get.edge.value.active
get.neighborhood.active
```

```
get.network.attribute.active
get.slices.networkDynamic
get.vertex.activity
get.vertex.attribute.active
get.vertex.id
get.vertex.pid
initialize.pids
is.active
is.adjacent.active
is.networkDynamic
%k%
list.edge.attributes.active
list.network.attributes.active
list.vertex.attributes.active
network.collapse
network.dyadcount.active
networkDynamic
network.dynamic.check
network.edgecount.active
network.extract
network.naedgecount.active
network.size.active
print.networkDynamic
reconcile.vertex.activity
search.spell
spells.hit
spells.overlap
%t%
vertex.pid.check
```

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.

Bender-deMoll, S., Morris, M. and Moody, J. (2008) Prototype Packages for Managing and Animating Longitudinal Network Data: dynamicnetwork and rSoNIA *Journal of Statistical Software* 24:7.

Butts CT (2008). network: A Package for Managing Relational Data in R. *Journal of Statistical Software*, 24(2). `http://www.jstatsoft.org/v24/i02/`.

. Fit, Simulate and Diagnose Models for Network Evolution based on Exponential-Family Random Graph Models. The Statnet Project

(<URL: http://www.statnet.org>). Version 3.2-11879-11880.1-2013.02.22-17.20.10, <URL: CRAN.R-project.org/package=tergm>.

Hunter DR, Handcock MS, Butts CT, Goodreau SM, Morris M (2008b). ergm: A Package to Fit, Simulate and Diagnose Exponential-Family Models for Networks. *Journal of Statistical Software*, 24(3). `http://www.jstatsoft.org/v24/i03/`.

Newcomb T. (1961) *The acquaintance process* New York: Holt, Reinhard and Winston.

Freeman, L. C., Freeman, S. C., Michaelson, A. G., (1988) On human social intelligence. *Journal of Social Biological Structure* 11, 415-425.