Package examples for networkDynamic: Dynamic Extensions for Network Objects (Version 0.2-2)

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

March 7, 2012

THIS IS A DRAFT. Not all authors have approved, and in some situations the package gives incorrect results.

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 `stergm`, 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 `stergm`. 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.

```
> 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
> add.edge(triangle,3,1)
> 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))
> activate.edges(triangle,onset=4, length=2,
+                e=get.edgeIDs(triangle,v=2,alter=3))
```

The `onset` and `terminus` parameters giving the starting and ending point for the activation period (more on this and the `at` syntax later). Notice that the 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. 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.

2.2   Extracting a network

```
> 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] 2 2 2

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

[1] 1 1 0
```

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

[1] 0 1 1

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

[1] 0 0 0
```

At time 1, the vertex degrees match what would be expected for the 'timeless' network, but for the other time points the degrees are quite different. 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. Figure 1 (page 5) shows 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 exists from the sample population. In `networkDynamc` 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:
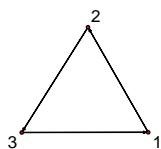
```
> activate.vertices(triangle,onset=1,terminus=5,v=1)
> activate.vertices(triangle,onset=1,terminus=10,v=2)
> activate.vertices(triangle,onset=4,terminus=10,v=3)
> network.size(network.extract(triangle,at=1)) # how big is it?

[1] 2

> network.size(network.extract(triangle,at=4))

[1] 3
```
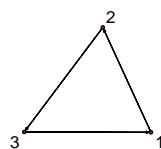
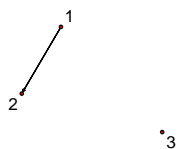Figure 1: Network plot of our trivial triangle network

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

```
[1] 2
```

Using the `network.size` function on extracted networks 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)
```

```
Problems detected with edge activity matrices.
$problem.vertexIDs
integer(0)

$problem.edgeIDs
[1] 2 3
```

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)
```

```
$problem.vertexIDs
integer(0)

$problem.edgeIDs
integer(0)
```

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.

There are many possible ways of representing change in an edgeset 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

This package uses the spell representation, and stores the spells as 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, `?activity.attribute`. As an example, to peek at the spells defined for the vertices:

```
> get.vertex.attribute(triangle,'active',unlist=F) # vertex spells

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

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

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

> get.edge.attribute(triangle$mel,'active',unlist=F) # edge spells

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

[[2]]
     [,1] [,2]
[1,]    4    6

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

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 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='until', 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 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.

5 Differences between Discrete and Continuous data

Its 2 am on Tuesday. Do you know what your temporal model is? Does 2am mean 2:00 am, or from 2:00 to 2:59:59? And other existential questions. The differences between at and onset, terminus syntax.

There are two different 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. Time is a 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. Egocentric survey data may aggregated into a set of weekly network 'panels', each of which is though 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 and an instantaneous point in time (as precisely as can be reasonably measured). Events can 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 7:45pm and ends on Friday at 10:01am. Continuous time models are useful when the the ordering of events is important. It still may be useful to represent observations in panels, 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.

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 which give different results:

```
> 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

> cont <- network.initialize(2)
> cont[1,2]<-1
> activate.edges(cont,onset=4,terminus=5)
> is.active(cont,at=4,e=1)

[1] TRUE

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

[1] FALSE

> cont <- network.initialize(2)
> cont[1,2]<-1
> activate.edges(cont,onset=3.0,terminus=5.0001)
> is.active(cont,at=4,e=1)

[1] TRUE

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

[1] TRUE
```

```
> 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=5,e=1)

[1] TRUE
```

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

Working with spells correctly can be complex, so 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.

```
> 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 by 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 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 Dynamic attributes: the next frontier

Clearly an essential feature of dynamic networks is the ability to express time-varying attributes for networks, vertices (changing properties) and edges (changing weights). The authors of this package have completed a specification and draft implementation of dynamic attributes, and will include dynamic attribute features and utility methods in an upcoming release of `networkDynamic`.

8   Other Coming Attractions

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. Provides the tools developed in (Bender-deMoll et al., 2008) but with R methods for building, controling, and rendering out animations.

References

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.

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

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/`.

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