Package examples for networkDynamic: Dynamic Extensions for Network Objects (Version 0.3.2-1695-1697.1-2012.11.26-22.50.33)

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

January 11, 2013

THIS IS A DRAFT. The package is not complete.

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. 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)
> class(triangle)
```

```
[1] "networkDynamic" "network"
```

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. 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] 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
```

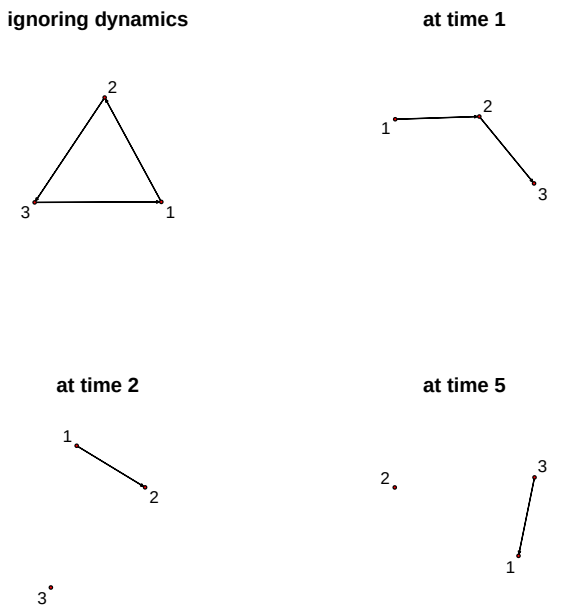


Figure 1: Network plot of our trivial triangle network

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
```

```
> 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)
```

```
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
```

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
```

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 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.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]]
     [,1] [,2]
[1,]  Inf  Inf

[[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 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.

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 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. Infact, it is entirely possible to activate the multiplex edges to attach (possibly conflicting) spell information. There are even a few 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 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 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
```

6.4 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 desireable 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 do accept arguments to specifiy which elements should have the timing information deleted.

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

numeric(0)
```

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`.

6.5 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
```

6.6 Stergm Example

COULD copy stergm example like in the ndtv package vignette, but since it uses `tergm`, which depends on `networkDynamic`, wouldn't adding it here create a circular package dependency?

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

How to get there from here. Some handy conversion tools for common representations of dynamics.

7.1 Converting from a list of networks

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 dataset 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, run `?newcomb`. This network is a useful example because it has edge weights that change over time and the `newcomb.rank` version has asymetric rank choice ties. Although this release of the `networkDynamic` package support dynamic edge attributes, the import utilities yet don't support edge wights.

```
> require(networkDynamic)
> data(newcomb)             # load the data
> names(newcomb)            # its some kind of named list object
```

```
NULL
```

```
> 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
```

```
   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
> 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, 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
```

```
>
```

7.2 Converting from initial network and array of toggle times

7.3 Converting a dynamic network to timed edge spell list

```
> newcombEdgeSpells<-as.data.frame(newcombDyn)
> dim(newcombEdgeSpells) # how big is it?
```

```
[1] 368   8
```

```
> newcombEdgeSpells[1:5,] # peek at the beginning
```

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

The first two columns of the spell matrix gives 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 the entire duration of the edge's activity is included or if it was partially censored by the observation. The `duration` column gives the total duration for the specific spell of the edge, not the entire edge duration.

8 A streaming data example: McFarland's classroom interactions

The `networkDynamic` function can create network dynamic objects from 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 dataset (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
1             14           12        0.125      0.125
2             12           14        0.250      0.250
3             18           12        0.375      0.375
4             12           18        0.500      0.500
5              1           12        0.625      0.625
  weight interaction_type
1      1           social
2      1           social
3      1         sanction
4      1         sanction
5      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 reorder 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)])
```

The data include some attribute information for the vertices which we'd like to add, but first we need to check if it is dynamic or not. We will assume that if each `vertex\_id` has only one row, the attributes 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 explicity convert factors to character values.

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

```
[1] TRUE
```

```
> 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))
```

If we peek at the change times of the network, it appears that it is certainly not a discrete time network.

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

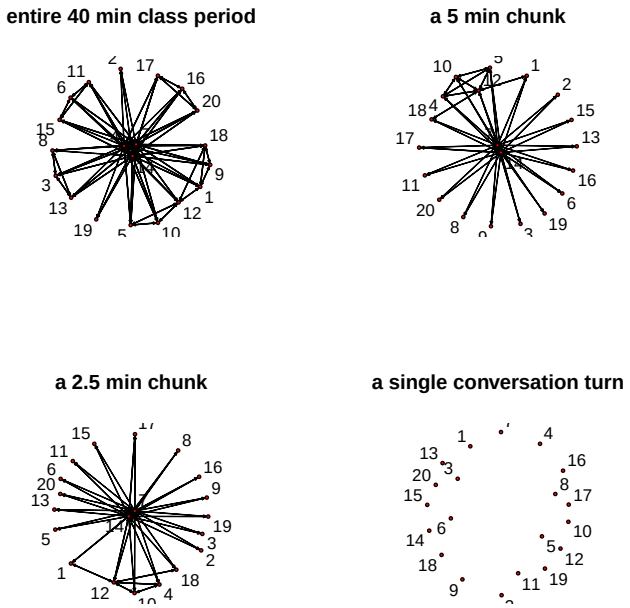
```
 [1] 0.000 0.125 0.250 0.375 0.500 0.625 0.750 0.875 1.000
[10] 1.167
```

This mean that 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 dataset consits 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)
```



9 Dynamic attributes

Clearly an essential feature of dynamic networks is the ability to express time-varying attributes for 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 giving the values respective time periods of activity. 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.

9.1 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 arey 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',list("red","green","blue","gray"),
+                            onset=c(0,1,2,3),terminus=c(1,2,3,4))
> 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.

9.2 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 happens 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 make it possible have a query that will intersect with multiple values. This returns a value, but 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
```

In many cases the user might want do 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 chosen at random? In order to handle these cases correctly, code must be designed to explicty 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 trimed) 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 objects is similar to the vertex case `get.network.attribute.active`. However, in keeping with the pattern established by `network`, the query function for edge TEAs is named `get.edge.value.active`.

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


[1]  5 12  7


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


[1] 1 2 1
```

There are are also functions for checking if TEA attributes are present at any point in time.

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


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

9.3 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
```

10 Making Lin Freeman's windsurfers gossip

In 1988, Lin Freeman collected a month-long dataset of daily social interactions between windsurfers on California beaches (Almquist, et al , 2011), (Freeman et al , 1988). The dataset 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 obeservation. (Run `?windsurfers` for more details).

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

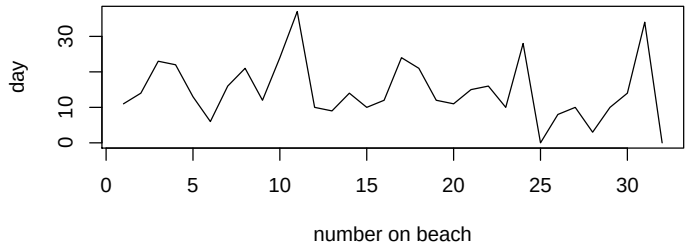
```
[1]  0 31
```

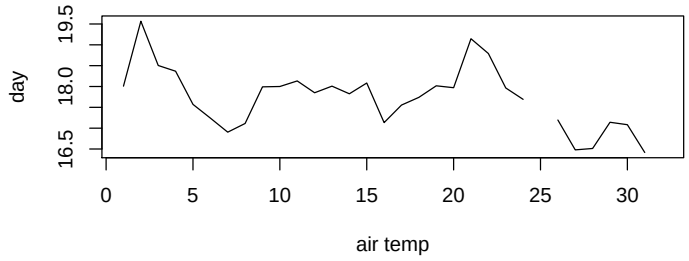
```
> sapply(0:31,function(t){ # how many people in net each day?
+   network.size.active(windsurfers,at=t)})
```

```
 [1] 11 14 23 22 13  6 16 21 12 24 37 10  9 14 10 12 24 21
[19] 12 11 15 16 10 28  0  8 10  3 10 14 34  0
```

Although not directly relevent of for the trivial sim we are about to build, the `windsurfers` network object also as some network-level dynamic attributes that give information about the weather, etc.

```
> 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))
```





We will create a very crude model of rumor transmission as an example of a process simulation using 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 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)
+ }
```

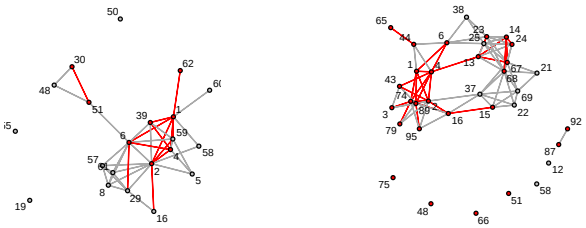
Then we set the paramters and the initial state of the network and run the simulation.

```
> 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!
```

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

```
> par(mfcol=c(1,2)) # show two plots side by site
> 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))
```



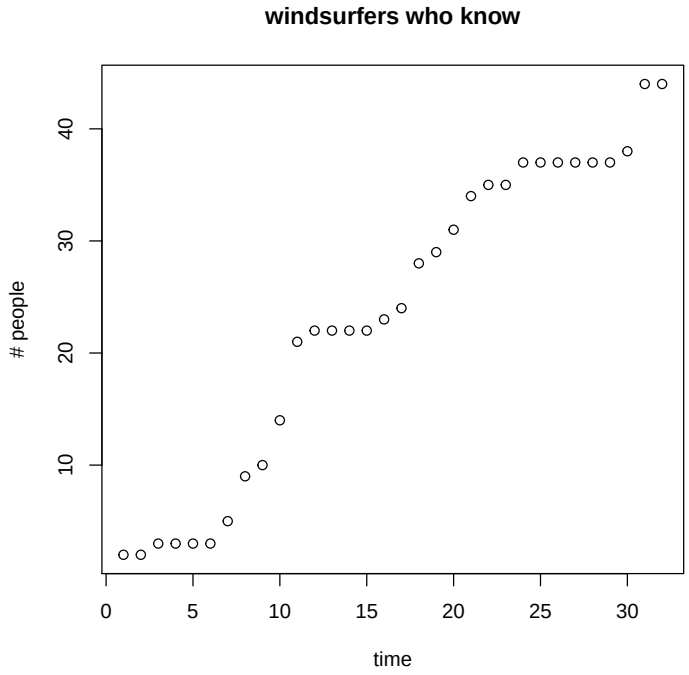
Which people heard the rumor halfway through the month? How many heard each day?

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

```
 [1] TRUE TRUE   NA TRUE   NA TRUE   NA   NA   NA TRUE   NA
[12]   NA TRUE   NA   NA   NA   NA   NA   NA   NA   NA   NA
[23]   NA   NA   NA   NA   NA   NA   NA TRUE   NA   NA   NA
[34]   NA TRUE   NA   NA   NA TRUE TRUE TRUE   NA   NA   NA
[45]   NA   NA   NA   NA   NA   NA TRUE   NA TRUE TRUE   NA
```

```
[56]   NA   NA   NA TRUE   NA   NA TRUE TRUE   NA TRUE   NA
[67]   NA   NA   NA   NA   NA   NA   NA TRUE TRUE TRUE   NA
[78]   NA TRUE   NA   NA   NA TRUE   NA   NA   NA   NA   NA
[89]   NA   NA   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 additiona to extracting values, we can do operations using the TEA attributes directly. We wrote the sim function so that it 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]])
```

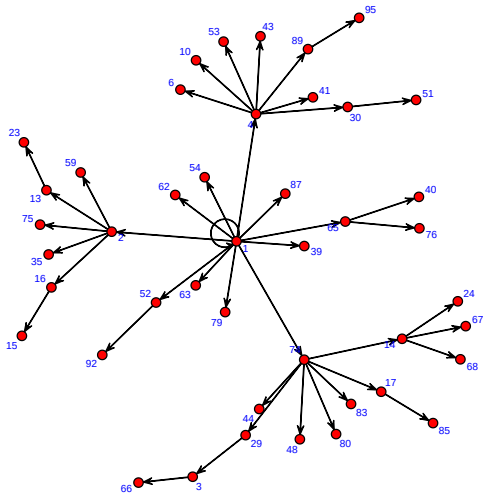
```
[1] 29 74 41  1 74
```

```
> # 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
```

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)
```



11 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.

12 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 begining (least time) of observation period (or series of spells)

end end (greatest time) of obeservation 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-censord 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 storeing dynamic attribute data on vertices, edges, and networks.

13 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.edges.active
add.vertices.active
as.data.frame.networkDynamic
as.networkDynamic
deactivate.edge.attribute
deactivate.edges
deactivate.network.attribute
deactivate.vertex.attribute
deactivate.vertices
delete.edge.activity
delete.vertex.activity
get.change.times
```

```
get.edge.activity
get.edgeIDs.active
get.edges.active
get.edge.value.active
get.neighborhood.active
get.network.attribute.active
get.vertex.activity
get.vertex.attribute.active
is.active
is.adjacent.active
is.networkDynamic
list.edge.values.active
list.network.attributes.active
list.vertex.attributes.active
network.dyadcount.active
networkDynamic
network.dynamic.check
network.edgecount.active
network.extract
network.naedgecount.active
network.size.active
print.networkDynamic
search.spell
spells.hit
spells.overlap
%t%
```

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/`.

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.