Journal of Statistical Software

February 2008, Volume 24, Issue 7. *http://www.jstatsoft.org/*

Prototype Packages for Managing and Animating Longitudinal Network Data: dynamicnetwork and rSoNIA

Skye Bender-deMoll **Martina Morris** University of Washington **James Moody** Duke University

Abstract

Work with longitudinal network survey data and the dynamic network outputs of the **statnet** ERGMs has demonstrated the need for consistent frameworks and data structures for expressing, storing, and manipulating information about networks that change in time. Motivated by our requirements for exchanging data among researchers and various analysis and visualization processes, we have created an R package **dynamicnetwork** that builds upon previous work in the **network**, **statnet** and **sna** packages and provides a limited functional implementation. This paper discusses design issues and considerations, describes classes and forms of dynamic data, and works through several examples to demonstrate the utility of the package. The functionality of the **rSoNIA** package that uses **dynamicnetwork** to exchange data with the Social Network Image Animator (**SoNIA**) software to create animated movies of changing networks from within R is also demonstrated.

Keywords: dynamic networks, network animation, longitudinal networks, **SoNIA**, Java, R, data exchange, modeling, disease transmission.

1. Motivation

The relational data-sets available for network researchers are becoming more complex, due in part to the use of data available from the Web, increasing sophistication in methods for data collection, and the availability of powerful tools and computing resources. Advances in storage techniques and analysis software continue to emerge, yet the conversion and manipulation of data remains challenging. As the costs of data collection, management and specialized analytic tool development increase in step with the sophistication and complexity of the data, the need for clarity and interoperability is becoming critical. Because of its acceptance in multiple research areas, its large existing code base, and a plug-in design structure to

facilitate communication between specialized packages, R (R Development Core Team 2007) provides natural host environment for developing inter-domain tools, provided that suitable common data formats can be developed and agreed upon.

This paper describes R packages developed by the CSDE Network modeling group to facilitate internal data exchange and analysis of rich, longitudinal, relational data, such as that produced by simulations and studies of disease dynamics[1]. The exact implementations of the packages described here are preliminary (not yet robust enough for research use) and may be drastically restructured and improved. Yet the discussion and examples will be useful for users who wish to approach the forefront in visualization, as well as establishing working examples of common flexible data structures that may help in the design of this or future tools, reducing the proliferation of incompatible data formats. This paper assumes a familiarity with R, and general concepts of network analysis. We also recommend reading the paper describing the use and structures of the **network** package for representing rich relational data (Butts 2008; Butts *et al.* 2007).

2. Dynamic networks

Renewed interest in studying time-varying relations has been emerging, perhaps due to the recent widespread awareness of network perspectives and the increasing feasibility of collecting and analyzing rich dynamic data. There are many ways of measuring longitudinal relations and coding them for storage and analysis. Early efforts at capturing network change over time used panel data, in which data were collected in the form of discrete 'waves', and sets of matrices were used to describe the states of bundles of relations at multiple points in time. Another common way of documenting dynamic networks is to record relations between nodes as lists specifying pairs of node identifiers and a series of intervals giving the starting and ending times of relationship activity. An example of this would be egocentric sex history self reports, or logs of phone conversations. Many kinds of traditional network analysis discard timing information and focus on relational structures. Alternatively, research may focus on the longitudinal aspects of the data using time series techniques but ignore the relational component. The combined use of timing and relational information is usually named 'Dynamic Network Analysis.' Any set of relational data containing time information can be referred to as a 'dynamic network.' Confusingly, the term 'dynamic network' is often used in the literature to describe various specific subclasses:

- Networks in which the edge and node sets remain fixed, but values of attributes on nodes or edges may vary in time (transmission models)
- Networks in which edges are added or deleted over time (computer networks, friendship relations)
- Networks in which the weights of edges change over time (neural networks, exchange networks)
- Networks in which nodes are added or removed in time (ecological food webs, organizations)

[1]This work is a product of many interesting discussions in the CSDE network modeling team including Carter Butts, Steve Goodreau, Mark S. Handcock, Dave Hunter, James Moody, Martina Morris, Deven Hamilton, Pavel N. Krivitsky and Krista Gile. Thanks to Deven for edits and comments on drafts.

Clearly these categories are not exclusive as most real world networks have some elements of each. The distinctions are mostly about what features modelers believe are most crucial for their problems. However, the data structures necessary to represent these dynamics are radically different. In order to support general research, we must be able to express all of the above in a single data structure.

We have found it useful to draw some finer distinctions among the kinds of edge-changing dynamic networks:

renewal dynamics in which an edge between a particular node pair may be renewed, switched on and off, having multiple active and inactive spells/intervals. In the case of an attribute, a variable which can have an arbitrary number of states over time.

non-renewal dynamics in which the existence of an edge can be defined by a single spell or interval; once it is deleted it cannot come back. Or a variable with a single state change.

time-function dynamics where the existence (weight) of an edge or value of a variable is defined by a time-varying function specifying a continuous value range over a time period (not dealt with in **dynamicnetwork** at this time)

These distinctions are useful because non-renewal networks can be represented with much simpler data structures and input formats. Relations can be expressed as single 'row' or observation with a single start time and end time. When the renewal constraint is relaxed, data structures must allow for the fact that some relations may toggle more often than others, necessitating some compromises on memory efficiency or more advanced design.

In order to measure or model the properties of an entity (relation or attribute) changing in time, we need to be clear if we are using a discrete-time (in which values of measurements represent discrete blocks of time, usually at regularly-spaced intervals) or continuous-time (in which values represent instantaneous observations at arbitrary time points) methods.[2] Both discrete and continuous techniques have merit (and when the data have already been collected, the decision may be moot) but each have implications for the concepts that should be applied to the data and how it can be accurately represented.

Of course in addition to the complexities of representing the temporal aspects of relations, we need to be able to encode networks having a wide range of features and distinctions, such as those supported within the **network** package (Butts *et al.* 2007). Networks may need be directed, undirected, bipartite, permit multiple types of edges between nodes, have missing or unobserved data, permit hyper-edges, to name a few examples. We may need to attach arbitrarily complex objects to nodes or edges. We would like to have a consistent way of representing all of these options, expressing changes of the properties of nodes over time (such as changes in size, infection status, etc.) along with edge dynamics in an intuitive way.

[2]For more on this, and how it relates to networks, see the discussion in Bender-deMoll and McFarland (2006)

3. Previous work

3.1. Stacks of matrices

The **dynamicnetwork** package builds upon a great deal of previous work and package structures. Historically when sociologists have longitudinal network data to work with the number of individuals in the network is not large and the simplest mode to express the data is a set of socio-matrices corresponding to the waves of the survey (Newcomb 1961; Zachary 1977; Sampson 1968). The widely accepted **UCINET** (Borgatti *et al.* 1999) software for network analysis works in this way. The original versions of the **sna** package for R (Butts 2007), implemented a logical extension of this as the 'graph stack', a set of socio-matrices such that the value of the relation between i and j observed at time t is indexed by a triple $[t, i, j]$. If we were to print out an example network of this structure to the R terminal, we would see the following matrices in which the values indicate the existence of ties from the row-th node to the column-th node:[3]

The socio-matrix for a 5-node network at time 1:

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    1    0    0
[2,]    1    0    1    0    1
[3,]    1    0    0    1    0
[4,]    0    1    1    0    0
[5,]    1    0    1    0    0
```

The socio-matrix for the network at time 2:

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    1    0    1    0
[2,]    0    0    1    1    0
[3,]    0    1    0    1    0
[4,]    0    0    1    0    1
[5,]    1    0    0    1    0
```

This system is straight-forward and effective, but quickly becomes inefficient when the network has more than a moderate duration (> 10 time points) and/or number of individuals is moderately large (> 100). Some of the inefficiencies of the matrix-based technique can be reduced by taking advantage of the fact that most socio-matrices are sparse and can be expressed internally in the software as edge-lists rather than matrices containing mostly zeros. But using numerous graphs to represent the evolution of a network is still inefficient when few cells in the matrix change at each time point (the network is 'temporally-sparse'). The graph stack is also only appropriate for discrete time data, and does not help with representing changes in other attributes of the network.

3.2. Lists of networks

The **statnet** and **network** packages (Handcock *et al.* 2003) for R used the object-oriented

[3] Confusingly, the indices in the example have been changed from $[t, i, j]$ to $[i, j, t]$ so that the output from R will appear more legibly as matrices.

capabilities of R to further extend the functionality of **sna** by adding an edgelist-based data structure back-ended in C, including support for multiplex networks, arbitrary graph, node, and edge attributes, and a number of other helpful capabilities documented in Butts *et al.* (2007). This created a very flexible structure for representing cross-sectional network data as rich-graphs, but did not aid in dealing with longitudinal data, which needed to be stored as a series of network objects in a list. A potential problem with this method (and also with the graph-stack) is that there is some ambiguity about the duration of a relation because it not possible to distinguish edges uninterrupted edges from edges that break and reform in the subsequent time step. This approach can also be very wasteful when there are a large number of time-steps with only minor changes to the graph.

3.3. Change lists

An alternate and more memory-efficient method is to store only the network at the first time point, and then store a list of modifications as 'toggles' or deltas to the initial graph. This change-list can be 'played back' to recover the state of the graph at any chosen time. This format is quite convenient for some kinds of simulations (especially Markov) and it is how data are output from the **statnet** `simulate.ergm()` commands. The data format is a thee column matrix with the columns indicating the time step, the sender and the receiver respectively[4]. Each row is a single relation change in the network time series, so values listed under time step t are the changes made to the network at time step t to make it the network at time $t + 1$. They must be listed in time step order. An example of a very trivial network in the format would appear as:

```
     tstep sender receiver
[1,]   1   1   2
[2,]   5   1   3
[3,]   6   1   2
[4,]   6   4   1
```

This structure would be read as: starting with an empty graph, at time 1, turn on the edge from 1 to 2, turn on the edge from 1 to 3 at time 5, turn off the edge from 1 to 2 at time 6, and then turn on the edge from 4 to 1 at time 6.

Writing code to use data in this format can be a bit more complicated, and the general approach can by costly in terms of processing time because obtaining the graph at any point along the time series requires the reconstruction of the entire history of the network up to the time point of interest. For short durations and temporally sparse networks these costs are negligible, but as the length and temporal density increase processing times can become prohibitive. However, one desirable feature of the technique is that the times themselves need not be integers, so the format can be used for either discrete or continuous data.[5]

3.4. Time intervals

Perhaps the most parsimonious technique for storing dynamic network data is to assign a

[4]The documentation for the **network** package refers to the sender of the relation as the 'tail' and the receiver as the 'head'

[5]A variation on this system is described in Ferreira and Viennot (2002) and a similar grammar for expressing network change is include in one of the variations in Pajek's `.net` format (Batagelj and Mrvar 2007).

'valid time interval' or 'spell' to each edge in the graph. If the graph is renewal, it may be necessary for an edge to have multiple valid intervals assigned to it. These intervals can be stored in a 'spell matrix' in which each row contains information on the spells of time for which the edge is present. This information can be stored in the form of a two-column matrix with one row per spell, such that the first and second columns contain the onset and termination times of the spell (respectively). Spells must be disjoint, and should be listed in a standard order to facilitate searching. When using a simple data structure, the spell matrix can be combined with the edge list to form a four column matrix giving the source, target, start time, and end time of each edge.

```
      source target start end
[1,]   1 2 1 6
[2,]   1 3 5 9
[3,]   4 1 6 9
```

This structure indicates that the edge from 1 to 2 is active from time 1 to 6, the edge from 1 to 3 is active from time 5 to 9, and edge from 4 to 1 is active from time 6 to 9.

Alternatively, the spell matrices can be used as attributes of edges in a more complex data structure. An example is the draft **diffusion** package (Butts 2006), that builds on the **network** data structures by attaching spell matrices to edges as attributes. If we print the properties of the attribute list (`mel`) for a single edge into text form we see:

```
R> myNet$mel

[[1]]
  [[1]]$inl
    [1] 2
  [[1]]$outl
    [1] 1
  [[1]]$atl
    [[1]]$atl$na
      [1] FALSE

    [[2]]$atl$edge.timing  #  a variable named edge.timing
      [1]  #contains spell matrix
             start end
       [1,]   0    3
       [2,]   5    10

[[2]]
(output truncated)
```

This structure is displayed as a set of nested lists. The first line indicates that this is the list of all the properties belonging to edge id 1. The sub-list `$inl` states that the edge links from node 2 to node 1 (indicated by the `$outl` element). The attributes of the edge are stored on the `$atl` list. The first element, `$na`, has the value `FALSE`. The second element,

`$edge.timing`, contains the spell matrix with the timing information. The matrix indicates that the edge is active from time 0 to 4, and also from time 5 to time 10. If the output was not truncated, it would next begin listing the attributes for edge id 2.

An advantage of the interval based system of data storage is that many standard calculations (durations and interval comparisons for example) become trivial because the information is already in interval form. Intervals can be used to represent either discrete or continuous time data. However, the raw syntax for accessing the time data in such systems is very complex and modifications to the graph can be expensive unless advanced indexing structures are used.

4. The dynamicnetwork package

Clearly there are trade-offs and advantages to each of the many possible representation styles. The approach taken in this version of **dynamicnetwork** was to design a set of extensions to the data structures of the **network** package and encode dynamics using a hybrid of several approaches. The `network` object of the **network** package appears within R as a list of named lists[6]:

`mel` the Master Edge List containing a sub list for each edge, in turn containing edge head- and tail-sets and attributes

`gal` the Graph Attribute List containing various named graph meta-properties or user-assigned attributes

`val` the Vertex Attribute list, with sublists for each vertex (node) containing named attributes

`iel` the In Edge List, an indexing structure listing the incoming edges for each vertex

`oel` the Out Edge List, an indexing structure giving the outgoing edges for each vertex

Users rarely interact with this structure directly, instead using a number of helpful constructor and accessor methods to build and manipulate the network.

The functionality of the **network** package is not changed by **dynamicnetwork** but extended with new temporal capabilities where appropriate. An advantage of R's multiple class inheritance system is that objects representing rich-temporal-graphs can have both '`dynamic`' and '`network`' class names.[7] Network objects having the class '`dynamic`' are required to have three additional graph-level attributes: `isRenewal`, `dynam.attr.names`, `dyn.edge.attr.name` explained below. Many of the functions work similarly to their **network** counterparts, with the addition of a two-element vector specifying the beginning and ending of a time range to which the function will be applied on the network. As with the **network** package, it is assumed that normal users should not need to manipulate data structures directly but can instead use the provided accessor functions, moving the messy details into the background. A version of the **dynamicnetwork** package is located with the supplementary files for this paper,[8]

[6]The data are actually stored in a special C back-end that can be accessed via an API to perform low level tasks very quickly (see Butts *et al.* 2007).

[7]This means that when using overloaded functions such as `plot()` R will search first for the appropriate **dynamicnetwork** function and call the **network** counterpart if it is not located.

[8]The source for **dynamicnetwork** is in supplementary file `dynamicnetwork_0.0-4.tar.gz`, a Windows binary is available in `dynamicnetwork_0.0-4.zip`.

the most recent version can be installed by answering 'yes' to the appropriate prompt in the `update.statnet` command in in the **statnet** package.

4.1. Edge changes

The name of the structure in **dynamicnetwork** implemented to represent the addition and deletion of edges in the network is the 'Edge Time List' (`etl`). It is a simple list that parallels the 'Master Edge List' (`mel`) structure and uses the 'time interval' strategy, storing a single (start, end) valid interval for each edge with the same list position as on the `mel`list. In a sense the `etl` list serves as a 'time index' for looking up the edges the way the `iel` and `oel` lists are 'neighbor indexes' for the nodes. Although the current design does not permit directly representing renewal relations in the `etl` list, we can take advantage of the network package's ability to represent multiplex edges in order to represent a network in which a relation needs to toggle multiple times. Additional 'clone' edges with the same starting and ending vertices and different `etl` values can be added to encode later appearances of the relation. If we print the `etl` list for a network having only two edges into text form we might see:

```
R> myGraph$etl

[[1]]
  [1]  1 10
[[2]]
  [1]  2 12
```

This indicates that the first edge starts at time 1 and ends at 10, while the second edge starts at time 2 and ends at time 12.

This stratagem has some other desirable properties. When a `dynamicnetwork` object is passed to a **network** algorithm that does not or cannot use the timing information, it will appear as multiplex network. Also, when working with network data in a simulation context it is possible to update the edge timing quickly by either incrementing the end time for the edge, or by using `NA` to represent an open interval and inserting the appropriate end time when the edge is toggled off.

Unlike the network package, the functions and structures of **dynamicnetwork** have not been back-ended in C and as a consequence they are less efficient and their implementation is relatively fragile[9].

4.2. Attribute changes (using spell matrix)

Attribute changes are implemented with a system similar to the change-list format. Dynamic attributes are stored as a two-column matrix in which the first column contains the attribute value, and the second the time at which the attribute should take on that value. Values must be in ascending order by time, but it is up to the user to insert them in order. The attributes are stored on the vertex attribute list (`val`) in the same manner as static attributes but are distinguished by being listed in `dynam.attr.names`.

[9] It is possible to create weird masking errors or for the timing data to not be copied correctly if the structures are accessed in non standard ways

```
R> myGraph$val[[10]]

$color
     [,1]      [,2]
[1,] "blue"      "8"
[2,] "green"    "9"
[3,] "red"      "10"
```

This example shows the vertex attribute list for node 10 of codemyGraph. The value of '`color`' for node 10 is undefined for all times before 8, 'blue' from 8 to 9, 'green' from 9 to 10, and is 'red' for all values after time 10.

4.3. Vertex changes

There is not yet a mechanism for expressing node entrances and exits (births and deaths) in **dynamicnetwork**. We would propose implementing this in a fashion similar to the edges (adding a 'Vertex Time List' with valid intervals) but this would mean that the effective size of the network could fluctuate in time, having serious implications for many of the functions in **network**. A good implementation would require more substantial modification of the network package and thus conflicting with our goal of extending by overriding. However, in some modeling and visualization situations it may be appropriate to emulate the addition and deletion of nodes by adding a dynamic attribute to express that they should not be drawn (or should not form ties) but this is not implemented in the package.

4.4. Simple example

What follows is a toy example of how to create a network programmatically, one command at a time, and perform a few simple visualization tasks.

We want to create a dynamic network with 10 nodes and no edges.

```
R> dyn <- as.dynamic(network.initialize(10))
```

Now we add an edge to the empty network dyn, with a time interval 1 to 10, linking nodes 1 and 2.

```
R> dyn <- add.edge.dynamic(dyn, c(1, 10), 1, 2)
```

Now we add a bunch more edges with various times.

```
R> dyn <- add.edge.dynamic(dyn, c(2, 10), 2, 3)
R> dyn <- add.edge.dynamic(dyn, c(3, 10), 3, 4)
R> dyn <- add.edge.dynamic(dyn, c(4, 10), 4, 5)
R> dyn <- add.edge.dynamic(dyn, c(5, 10), 5, 6)
R> dyn <- add.edge.dynamic(dyn, c(6, 10), 6, 7)
R> dyn <- add.edge.dynamic(dyn, c(6, 10), 7, 8)
R> dyn <- add.edge.dynamic(dyn, c(7, 10), 8, 9)
R> dyn <- add.edge.dynamic(dyn, c(8, 10), 9, 1)
R> dyn <- add.edge.dynamic(dyn, c(9, 10), 10, 1)
```

```
R> dyn <- add.edge.dynamic(dyn, c(9, 20), 10, 1)
R> dyn <- add.edge.dynamic(dyn, c(9, 20), 10, 5)
R> dyn <- add.edge.dynamic(dyn, c(9, 20), 10, 3)
R> dyn <- add.edge.dynamic(dyn, c(9, 20), 10, 8)
```

Now add some attributes. First give vertex ten the color blue at time 8.

```
R> dyn <- set.dynamic.vertex.attribute(dyn, "color", "blue",
+    valid.time = 8, v = 10)
R> dyn <- set.dynamic.vertex.attribute(dyn,"color", "green",
+    valid.time = 9, v = 10)
R> dyn <- set.dynamic.vertex.attribute(dyn,"color", "blue",
+    valid.time = 10, v = 10)
```

A quick way to examine the dynamics of a network would be to plot a series of 'snapshots' of the network, showing the edges and attributes present at each time.

Now plot a series of slices through our example network to show it at various times.

```
R> par(mfrow = c(3, 3))
R> for(p in 2:10) {
+    slice <- get.slice.network(dyn, p)
+    plot(slice, vertex.col = get.vertex.attribute(slice, "color"),
+    vertex.cex = 5, main = p)
+    }
```

The graphic output resulting from the command is shown in Figure 1.

Another simple and effective technique for getting an overview perspective on the dynamics of a network is to plot the spells/intervals associated with the edge and node events on a timeline. This shows none of the connectivity information, but makes it possible to visually asses rates and durations of events. We can combine the two approaches, allowing the user to use the timeline to select what snapshot to view. When called with `pickSlice = TRUE`, the plot will wait until the user clicks on a point on the timeline before drawing the graph for the corresponding time slice.

```
R> plot.intervals(dyn)
```

The graphic output resulting from the command is shown in Figure 2.

If we call `plot()` without specifying a time interval, we implicitly call the `plot.network()` function and draw a network that is the union of all the edges (all the edges ever present in the graph's history). Because all of the edges are present at time 10 in our example, this looks structurally like the network at time 10. As we are looking at the entire time range there multiple values of the attributes for each node, and we lack a rule to determine the coloring.

Make a new plot window and plot the all-time network.

```
R> plot.new()
R> plot(dyn)
```

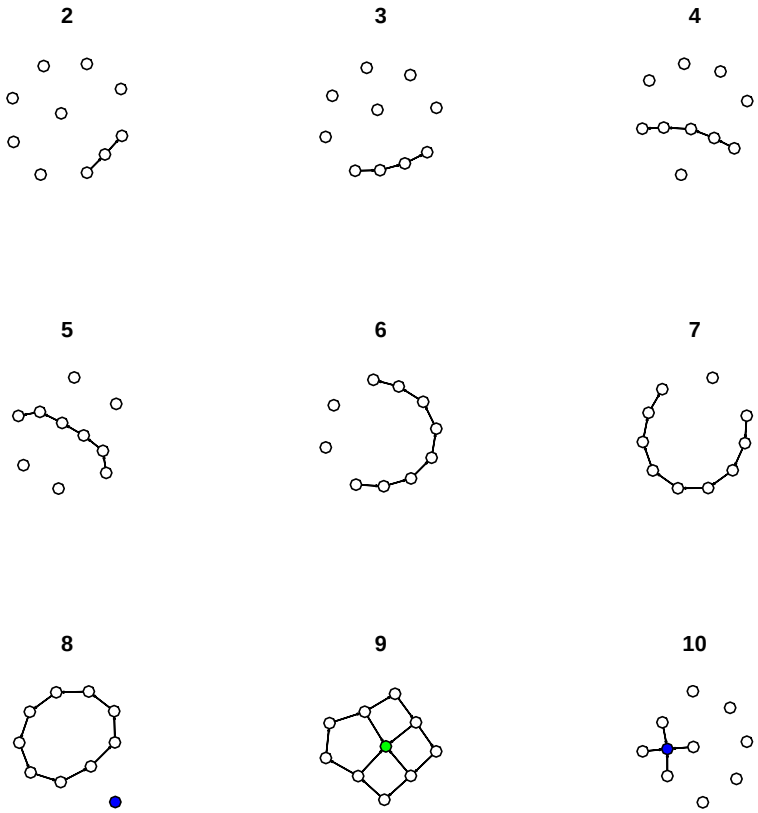

Figure 1: Graphical output of `plot()` command showing nine time steps of the dynamicnetwork example.

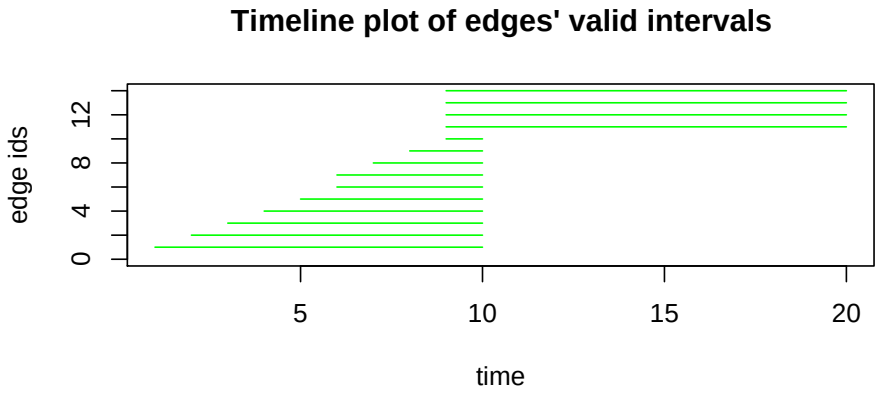

Figure 2: Graphical output of the `plot.intervals()` command for the example network.

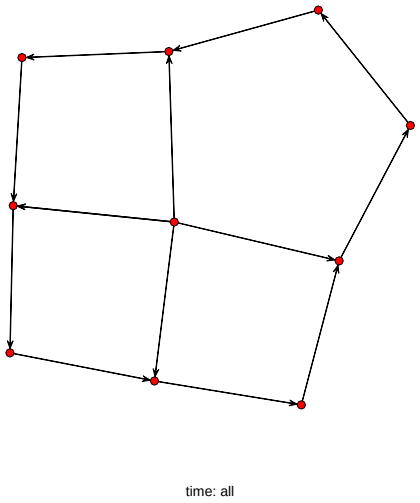

Figure 3: Graphic output of the `plot.network` command.

The graphic output resulting from the command is shown in Figure 3. Edge timing information is not used.

Because dynamic network data arrive in any of the forms we discussed, **dynamicnetwork** includes a number of methods for converting other structures into the network objects, including lists of networks, change-lists, and intervals formats. These constructors are explained in more detail in the Appendix, along with a listing of some other basic methods for working with time data.

5. Visualizing dynamics as network animations

5.1. Why visualize?

Summary statistics that describe the properties of networks have been improving over the years, but still leave much to be desired as a means for succinctly communicating the structure of a network data-set to a reader or research team member. Graphical depictions of networks are often used although, as with any representation of multi-dimensional structures in lower dimensions, they usually must introduce some distortion of the distances or selective compression of less interesting features in order to create a 2D display. The particular details of the distortions in an image are a function of the drawing algorithm used to create it and the structure and subject of the input network. Visualizations often serve as a shared point of reference for discussion, making it possible to effectively demonstrate complicated models without hand-waving. There are myriad algorithms available for creating these types of visualizations, each with their own advantages and disadvantages, and several of them are included in the `plot.network()` function of the **network** package.

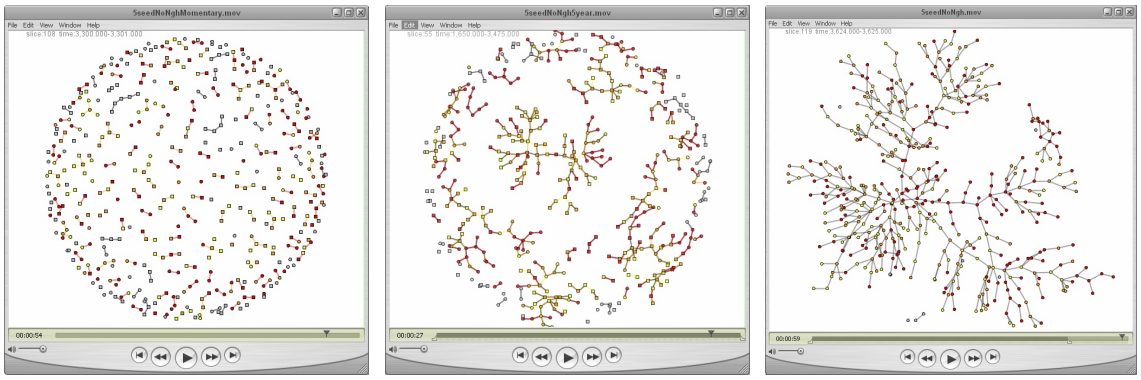

Figure 4: Example of three levels of time aggregation for the same data, movies of a simulation of disease spread, with 1 day, 5 year, and 10 year windows of aggregation. Complete movies in `.mov` format are located with the supplementary files for this paper as `5seedNoNghMomentary.mov`, `5seedNoNgh5year.mov`, and `5seedNoNgh.mov`.

A natural way of showing the time component of relational structures is to use animations—at least when communication with a viewer is not restricted to the printed page. Bender-deMoll and McFarland (2006) argue that viewing animated data may make it more accessible to researcher's intuitions, allowing us to use the advanced motion and pattern detection capabilities evolved into our perceptual systems. Network animations are also very helpful as debugging tools. Errors or miss-specification of a simulation model often became glaringly obvious once its output is rendered in a visual form. Visualization should not replace rigorous numeric analysis and hypothesis testing. Overly 'flashy' animations can be distracting, and should be used judiciously. Animations seem to increase anthropomorphizing and projection of causality on to co-occurring events, partially because our ability to critically examine network diagrams is as yet undeveloped. Despite these drawbacks visual representations and animations are evolving into increasingly valuable research tools coincident with our increasing ability to generate them in an efficient and principled way.

5.2. Network time-scales

In order to create an animation of a network we need to know what that graph looks like at each point in time. But to show a single graph for a dynamic network, we need to specify what time range we are going to look at. In other words, although network data may be given a form such that they describe the state of the relations at each consecutive day, it may be more useful from an analytic perspective to consider instead the network formed by the set of all relations that take place in a month's time. These networks may show very different kinds of structures (trivially, we expect the month-long network to be denser) and the appropriate time-scale for a specific data-set will depend on the methods of data collection, the pace of change in the network, and the researcher's questions (see Figure 4).

5.3. Applying layouts

The basic model employed for visualization is that the relations and nodes of a dynamic network can be treated as events that specify various properties and have time information

attached. These events can be grouped into a series of bins according to their valid intervals, with all the edge events in a bin defining the relations and attributes of a cross-sectional network for the corresponding time period. Once the data have been binned up according to the user's specifications, layout algorithms can be applied to the cross-sectional networks in succession in order to determine appropriate placement for each node based on the weights of its connections to other nodes. Exactly how this is done depends on the layout algorithm, but the general idea is to optimize positions until the distances between nodes as measured on the paths of the network correspond as closely as possible to the distances between nodes on the layout.

The layouts of each cross-sectional 'slice' of the dynamic network serve as 'key frames' in a movie, snapshots of the network structure regularly spaced along a timeline. Once a key frame has been created for each bin, we can render a sequence of frames between the keyframes such that the nodes' coordinates are smoothly interpolated to give the visual impression of continuous motion. In addition, the bins can be allowed to overlap, somewhat like a rolling average. In a number of visualizations we found it desirable to see the moment-to-moment changes in the network against a slower-moving structure of node positions. We were able to achieve this using renders that are on a different time-scale than the layout slices, and adjusting the 'offset' (the relative position of the leading edge of the render). In other words, in many cases we could construct the layouts and node positions using networks built from an entire year's data, but animate through the networks showing only one day's worth of ties at a time. There are of course a large number of tricks and techniques involved in making this work well.[10]

The algorithm we have found most useful in constructing network animations is a variant of the method of Kamada and Kawai (1989), an energy optimization procedure that treats the network as if it was a collection of springs with resting length proportional to the graph-theoretic path distance between the nodes. The algorithm uses an energy minimization technique to squash the tangle of springs flat onto the page while introducing as little distortion as possible.[11] The calculations needed to compute the shortest paths for a single network and to perform the optimization can be very expensive and they scale quite badly with network size. In order to create an animation, we must repeat the calculation for each slice of the network ('tho we get some efficiency by using the output of the previous calculation as a starting point.) The size of the networks that can be effectively visualized with these techniques depends a great deal on the properties of the networks. We find it fairly feasible for networks up to 1k nodes and 2k edges on a modern desktop computer.[12]

6. The rSoNIA package

The process of creating network animations outlined in the previous section has been coded in a Java program named **SoNIA** (Social Network Image Animator, see Bender-deMoll and McFarland 2003) that permits interactive browsing of dynamic network data and exporting

[10] For more information on how this is done, see Bender-deMoll and McFarland (2006); Moody *et al.* (2005), and **SoNIA**'s online documentation (Bender-deMoll and McFarland 2003)

[11] The dimensionality reduction problem is essentially the same as Modern Multidimensional Scaling.

[12] This has been made possible by improvements in efficiency funded by a NIH grants R01 HD41877 and R01 DA12831 made to the Network Modeling Project of the Center for Studies in Demography and Ecology at the University of Washington.

animations as a **QuickTime** movies. Java (1.5+) must be installed on the system in order to run **SoNIA**. The **rSoNIA** package provides a set of methods to facilitate exporting data and parameter settings and launching the **SoNIA** software in interactive or batch mode from within R. **rSoNIA** consists of two major functions, `launchSonia()` and `updateSoNIA()`, as well as a number of parameter settings objects that store the key value pairs used to control **SoNIA**'s various settings. **rSoNIA** requires that **dynamicnetwork** be installed previously in order to provide a consistent data structure within R to store and export the data. The most recent version[13] of **rSoNIA** can be downloaded by running the `update.statnet` function in **statnet** and answering 'yes' when prompted about installing **rSoNIA**.

SoNIA requires a number of additional libraries such as **COLT** (CERN 1999) which provides advanced numerics, and **FreeHEP** (FreeHEP 2000) which gives the ability to export various image formats. R's package installer does not appear to be well suited to dealing with Java dependencies, so most of the details of installation are hidden from the user inside the `updateSonia()` function. This command checks that an appropriate version of Java is installed, downloads the Java-based installer from the sourceforge.net repository, determines an appropriate location to install (within the **rSoNIA** package) and launches it. This installer in turn retrieves the various Java libraries, unpacks them and then asks the users to download and install the **QuickTime** software and libraries (Apple Inc. 1999) from Apple's Web site.

The `launchSonia()` command de-parses the passed `dynamicnetwork` object into a cache file, modifies the settings objects as specified in its arguments, creates an appropriate command string, and launches **SoNIA** as an external Java application from the command line using the R `system()` function.[14] **SoNIA** in turn reads the settings and the network data and creates the appropriate layout. Although launching from within R is very convenient, a major disadvantage is that R continues to consume a large amount of CPU resources while waiting for **SoNIA** to return, making the layout process take extra time. When in interactive mode, the **SoNIA** user interface will be visible, allowing the user to further configure the layout parameters and export the resulting network. When launched with `interactive = FALSE`, the UI never appears, and the entire processes returns when the movie has been generated and written to a file. The most commonly modified parameters can be set with arguments to `launchSonia()`. The remaining parameters can be changed by modifying the values of the properties objects in the environment before the launch.

`soniaApplySettings` Settings object to specify the parameters for the algorithm (maximum iterations, etc.) and additional transformations (re-scaling, etc.)

`soniaBrowseSettings` Settings that control how **SoNIA** will step through the network when generating an animation (render duration, offset, number of interpolated frames).

`soniaGraphicsSettings` Parameter settings object that controls graphic details of how networks will be rendered, including scaling, transparency, labeling options

`soniaLayoutSettings` Parameter settings object that specifies how the dynamic network data should be binned up to create animations, how edges should be aggregated, and

[13] The source for the version of **rSoNIA** used in this paper is available with the supplementary file `rSoNIA_0.0-4.tar.gz` and a Windows binary as `rSoNIA_0.0-4.zip`.

[14] In the process of constructing **rSoNIA** we initially investigated using the **rJava** package which permits the construction and manipulation of Java objects from within R. Although the package is quite powerful, we found it unsuitable because it is not possible to share memory space between R (based in C) and Java directly.

what layout algorithm should be used.

`soniaMovieSettings` Settings controlling movie export options, file type and name

`soniaParseSettings` Settings that configure the parser, giving the names of the network variables that code for size, color of nodes and edges, etc.

Details of the specific parameters in the settings objects are available through the R documentation system.

SoNIA currently exports animations in two formats:

`.mov` **QuickTime** (Apple Inc. 1999) raster-based animation

`.swf` **Flash** vector animation (Adobe Systems Incorporated 1999).

Unfortunately Apple does not produce the necessary libraries for the linux platform, so `.mov` is only available on Mac and Windows machines. The **Flash** export works on all platforms, but is currently limited in the duration/number of objects it can include in a movie; this will hopefully improve in the future. **SoNIA** also includes the ability to store some dynamic data in the DynetML XML format (Tsvetovat *et al.* 2004).

7. More complete example (`faux.mesa.high`)

7.1. Define an ERGM based on the `faux.mesa.high` network data

Faux.mesa.high is a simulated friendship network created using an ERGM (Hunter *et al.* 2008) fit to a friendship network that was included in the national Longitudinal Study of Adolescent Health Wave I (Resnick *et al.* 1997). In addition to friendship ties the data set includes static vertex attributes of race, grade, and sex for each of the individuals, and the network shows some patterns of preference or segregation based on these attributes. See Hunter (2007) or **statnet** documentation for further details of the model construction.

Load the faux.mesa.high data and check which attributes are present.

```
R> data("faux.mesa.high")
R> list.vertex.attributes(faux.mesa.high)

[1] "Grade" "na"    "Race"  "Sex"
```

Define an exponential random graph model includes various attribute effects

```
R> fauxmodel.01 <- ergm(faux.mesa.high ~ edges +
+    nodefactor("Race") +
+    nodematch("Race", diff = TRUE) +
+    nodefactor("Grade") +
+    nodematch("Grade", diff = TRUE) +
+    nodefactor("Sex") +
+    nodematch("Sex"))
```

```
Warning: The count of nodematch.Race6 is extreme.
To avoid degeneracy the terms nodematch.Race6 have been dropped.
```

Use the `simulate()` command to generate a sequence of graphs from that model. This means we are actually going to create an animation of the modeling process rather than a substantive dynamic model of the faux.mesa.high network. We will generate 20 networks using one of every 100 draws so there is enough change between networks for us to see something.

```
R> fauxsim <- simulate(fauxmodel.01, nsim = 20, interval = 50)
```

Load the **rSoNIA** and **dynamicnetwork** packages

```
R> library("rSoNIA")
```

Convert the simulation output to a `dynamicnetwork` object

```
R> fauxdyn <- as.dynamic(fauxsim$networks, check.renewal = FALSE)
```

Launch **SoNIA** with the basic network just to test, then quit the application.

```
R> launchSonia(fauxdyn)
```

7.2. Add attributes

The previous example could be used to create an animation of changing edges. But our input model included effects of node attributes, how can we show them in the animation? To use shapes of the nodes to show the sex of the students, we need to convert the 'sex' attribute into a vertex attribute of shape names and save them back into the network. Because the nodal attributes of faux.mesa.high are fixed the features for animating dynamic attributes will not be demonstrated.

Get the sex attribute, convert it to shapes, and save it back.

```
R> shapes <- get.vertex.attribute(fauxdyn, "Sex")
R> shapes <- replace(shapes, shapes == "M", "square")
R> shapes <- replace(shapes, shapes == "F", "circle")
```

Now we set the shape property of all the vertices

```
R> fauxdyn <- set.vertex.attribute(fauxdyn, "shapes", shapes)
```

Now we convert race into a vertex attribute of arbitrary colors using the `colorize()` function to assign colors to each category. First set vertex colors using results of a function.

```
R> set.vertex.attribute(fauxdyn, "color",
+    colorize(get.vertex.attribute(fauxdyn, "Race")))
```

Launch **SoNIA** in interactive GUI mode, specifying that it should read the color and shapes variables. As 'Grade' is already a numeric attribute, we can map it directly to node size using the `vertex.cex` parameter.

```
R> launchSonia(fauxdyn,
+    vertex.col = "color",
+    usearrows = FALSE,
+    displaylabels = FALSE,
+    vertex.shape = "shape",
+    vertex.cex = "Grade" )
```

7.3. Using the SoNIA GUI

After the **SoNIA** application has appeared, there are a number of steps necessary to actually create the animation to user specifications. Afterwards, when the appropriate settings for a network have been discovered, the layout can be generated by defining the same settings programmatically.

- Click the `Apply Layout` button in the layout window (Figure 5).
- When the layout parameter dialog appears (Figure 6), set `max passes` to `5000`. This specifies the maximum number of iterations the optimization will be allowed. Set `optimum distance` to `20`, this specifies the desired length (in pixels) of unit-length edges. Set `comp connect value` to `10`, this specifies a desired length of 'phantom' edges used to connected disconnected components.
- Click `Apply` and wait for the optimization process to finish laying out the first slice. The result should appear somewhat like Figure 7.
- If the layout looks good (it fits within the window, has nice component spacing) the same parameters should be applied to all the remaining slices in the sequence, starting each from the coordinates of the previous. Advance to the next slice layout with the `>|` button. Click `Apply Layout` again. Change `Starting Coordinates` from `use random positions` to `from previous slice` (Figure 8)
- Click `Apply to Remaining` to run through the rest of the slices. Once the layouts have completed the movie can be previewed or exported to **QuickTime** or **Flash** format.
- Rewind the movie with the `|<` button, or by entering `0` in the `Layout slice #` field and hitting return. Press play `>` to preview the movie. Choose `Export QuickTime Movie` or `Export Flash Movie` from the `Export` menu to save the movie to disk.

7.4. Using SoNIA in batch mode

In order to specify this sequence of operations in batch mode it is necessary to configure some of the parameter objects in **rSoNIA**. The settings are stored as key value pairs, so it is important to modify the value and not the key, or **SoNIA** will not be able to recognize the parameter.

Need to specify some of the settings so it will not use defaults, set desired spacing between connected nodes

```
R> soniaApplySettings$kk.opt.dist[2] <- 20
```

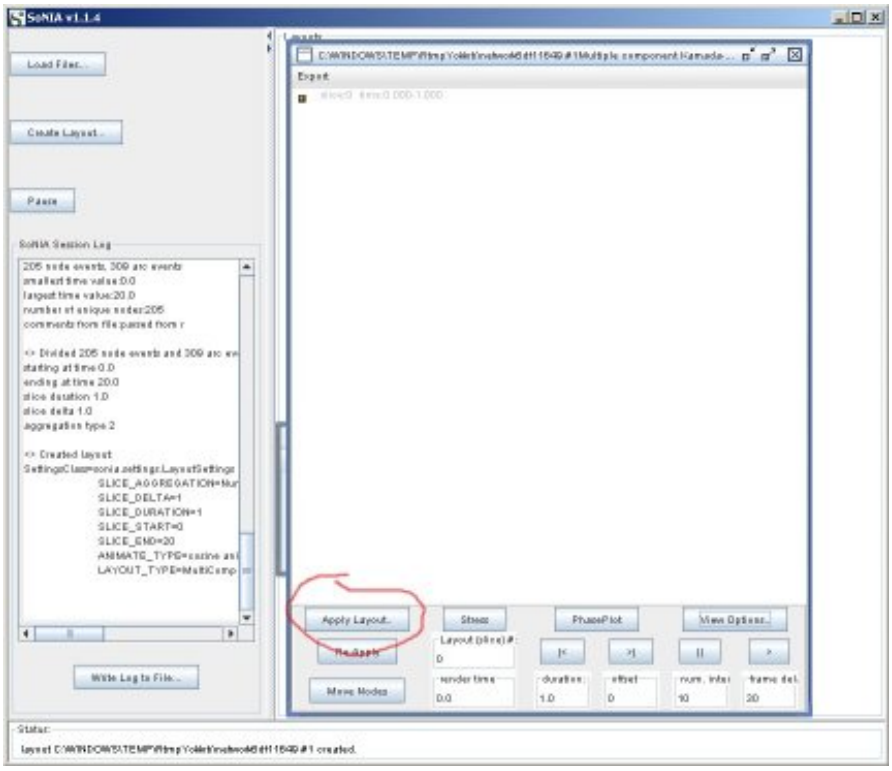

Figure 5: Screenshot of the initial **SoNIA** layout window.

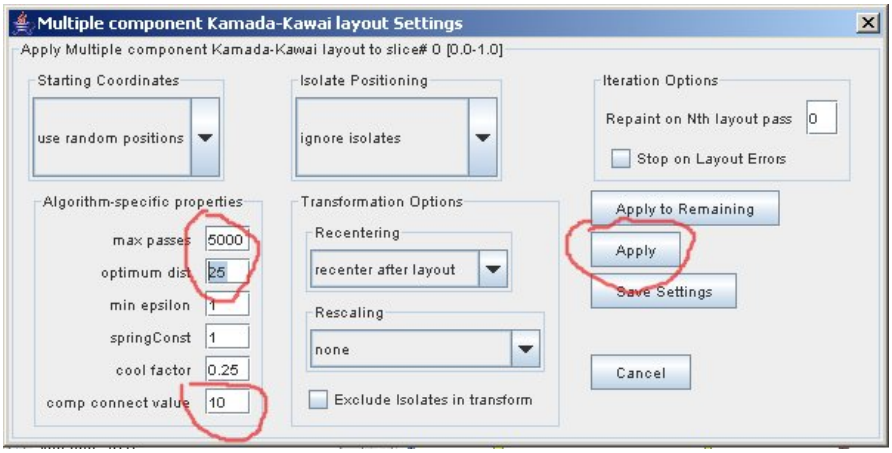

Figure 6: Screenshot of the layout parameter dialog in **SoNIA**

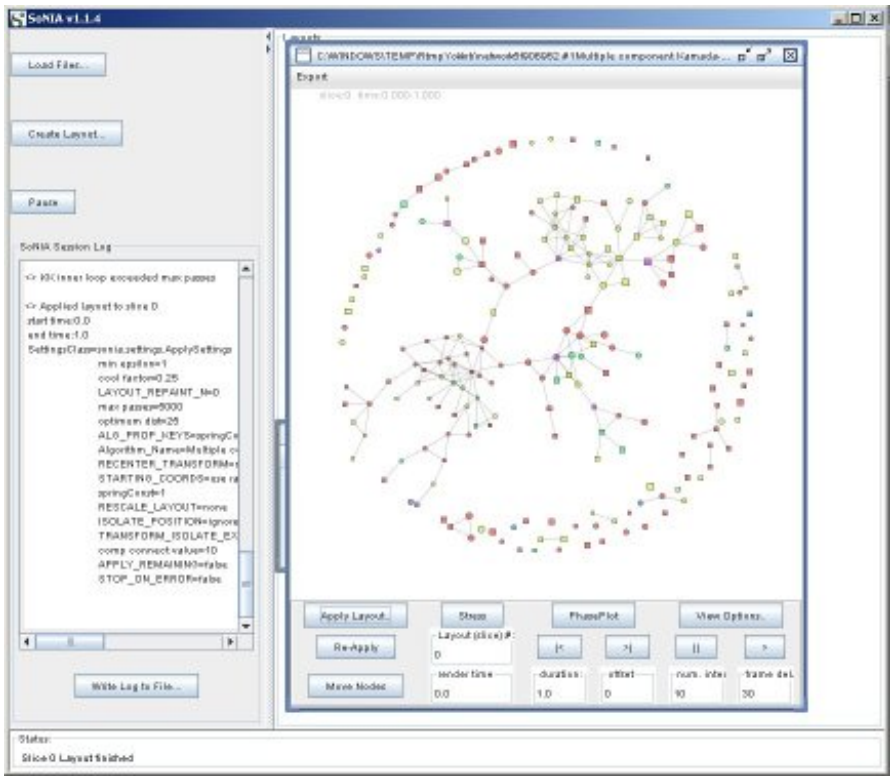

Figure 7: Screenshot of the first example layout in **SoNIA**.

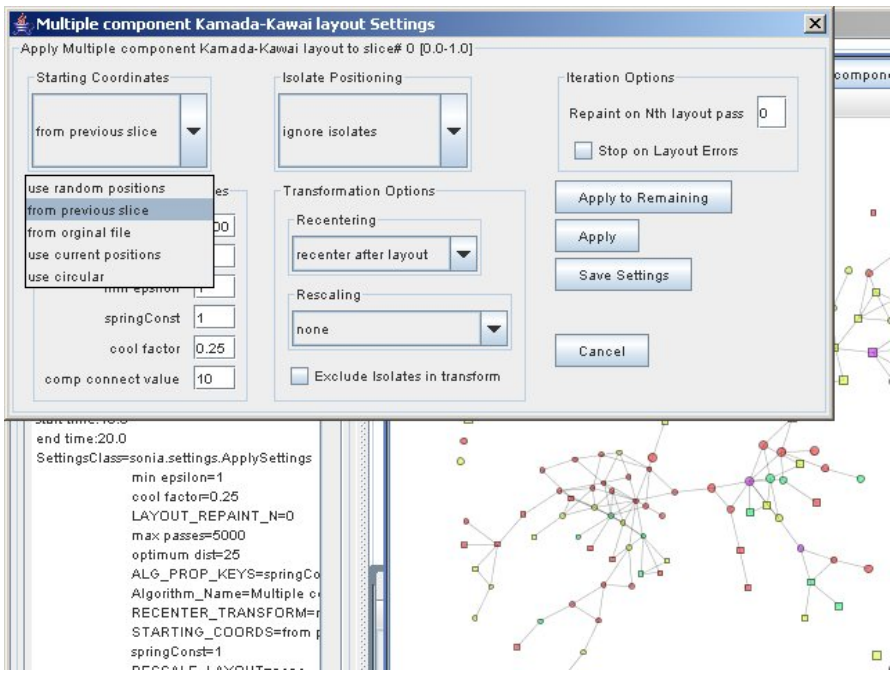

Figure 8: Screenshot of example in **SoNIA** showing how to select the `from previous slice` option.

Specify distance between un-connected nodes and components

```
R> soniaApplySettings$kk.comp.connect[2] <- 10
```

Increase height of the movie to 500 pixels

```
R> soniaGraphicsSettings$layout.height[2] <- 500
```

Decrease number of interpolated frames to reduce file size

```
R> soniaBrowseSettings$num.frames[2] <- 10
```

Generate a **Flash** movie in batch mode. save "`myMovie`" in the R working directory

```
R> launchSonia(fauxdyn,
+    max.iter = 5000,
+    interactive = FALSE,
+    vertex.col = "color",
+    usearrows = FALSE,
+    displaylabels = FALSE,
+    vertex.shape = "shape",
+    vertex.cex = "Grade",
+    movie.file = file.path(getwd(), "myMovie"),
+    movie.type = "swf")
```

This will result in a **Flash** file `myMovie.swf` than can be viewed in a browser or embedded on a Web page[15]. For an example, please see supplementary file `faux20.swf`.

On Mac and Windows machines[16] it is also possible export the movie as a **QuickTime** file.

Same command as before, but change `movie.type` to `"mov"` instead of `"swf"`

```
R> launchSonia(fauxdyn,
+    max.iter = 5000,
+    interactive = FALSE,
+    vertex.col = "color",
+    usearrows = FALSE,
+    displaylabels = FALSE,
+    vertex.shape = "shape",
+    vertex.cex = "Grade",
+    movie.file = file.path(getwd(), "myMovie"),
+    movie.type = "mov")
```

An example of the **QuickTime** movie created is `faux20.mov`.

[15] In some installations the default directory `getwd()` may not be writable from Java. If the movie does not appear, try specifying an alternate location to save the movie file

[16] Unfortunately, on Windows Server 2003 this does not work due to security settings

8. Extended example

8.1. Disease transmission and dynamic attributes

The key problem domain motivating the development of these packages is how to represent the rich temporal networks output by a combined simulation of network dynamics and disease transmission. These networks include changing edges (sex partners) and disease status (infection) and various static attributes (race, gender). In addition it is necessary to simulate very large networks ($n = 10,000$) in order to get results that are potentially generalizable. Previous work suggests than the amount of concurrency (overlap of simultaneous partners) has implications for the epidemic thresholds and prevalence of disease (Morris and Kretzschmar 2000, 1997). We constructed a number of visualizations from output of the in-house version of **statnet** in order to explore the properties of these networks. Although the size of these networks greatly exceeds the size of the networks **SoNIA** can effectively visualize, the problem can be reduced by extracting only the nodes and edges in the immediate (temporal) neighborhood of the simulated infection.

This example requires functions in '`buildInfectNet.R`' that are not properly part of the dynamic network package as they include code that is quite specific to CSDE research projects. We have included excerpts here to demonstrate the visualization process. These simulations were generated at a stage in the research processes in which the dynamics of the network and infection are assumed to be independent (we make the somewhat unrealistic assumption that individuals do not take infection status into account in their behavior). This makes it so that the simulation of the contact network evolution (based on data from Udry (2003)) and the simulation of the disease spread over the contacts can be separated into two stages. The first stage uses a detailed model of network dynamics including mixing and durations estimated from real data (coded by Mark S. Handcock, Martina Morris) and produces a file specifying the evolution of the network (in a change-list format). The stage second reads in the network dynamics, runs a disease transmission simulation (authored by Steve Goodreau and Susan Cassels), and produces a file giving the infection status of the nodes. For this exercise, the probability of infection was set to 1.0, so infected nodes will always infect their partners.

In addition, we were interested in tracking the sources of the simulated 'epidemics' and various details about the chains of transmission events in order to estimate the effects of concurrency. The simulation 'seeded' the population at the first time step with 10 independent infections. At each time step the infections were transmitted along the network of active ties. When a node become infected the time step, the id of the infection strain, and the concurrency status of the node is recorded. Concurrency is classified into multiple types depending on the onset and termination times of an individual's relationships and the infection status of his/her partners. The source and data files are available with the supplementary data for this paper (`buildInfectNet.R`, `tel.conrsmons.txt`, `infect.conrsmons.b1.10comseed.05.txt`) and can be loaded in as matrices using the standard R utilities.

8.2. Loading the data

Load in the `buildInfectNet` code

```
R> source("buildInfectNet.R")
```

Load edge dynamics file output by ergm simulation

```
R> edgeTiming <- read.table("tel.conrsmons.txt", header = TRUE)
```

Check that the file is a list of relations with the ids of partners start and end date for each.

```
R> edgeTiming[1:3,]
```

```
  partner1 partner2 startdate enddate
1     6932        2         0     868
2     8204        3         0     827
3     6075        4         0     225
```

Load infection dynamics file with node attributes and infection times

```
R> infectStatus <- read.table("infect.conrsmons.b1.10comseed.05.txt",
+    header = TRUE, fill = TRUE)
```

Peek at the variable names in the file

```
R> names(infectStatus)
```

```
 [1] "id"                  "race"                "sex"
 [4] "serostatus"          "time.infect"         "infector.ID"
 [7] "infector.recency"    "infector.race"       "ego.numpartners"
[10] "partner.numpartners" "seed"                "concurrencytype"
[13] "concurrencychain"
```

Some of the relevant columns are: `time.infect` which gives the time when node was infected, `seed` gives the id number of the infection seed, the 'strain' of infection, `concurrencytype`[17] is the type for the infecting event, `concurrencychain` is is the history of concurrency in transmission events leading to this infection.

The '`buildInfectNet.R`' code processes the edgeTiming and nodeStatus files to create a dynamic network with the appropriate properties for visualization. One if its main functions is to subsample the network to include only interesting nodes: those near to the infected nodes. The code includes options that control the coloring of edges and nodes based on a combination of input and derived properties. For example we can color the nodes by 'race', the 'strain' of the infection the received, the 'age' (number of steps) in the transmission chain, or the frequency of different concurrency relationships in the chain of infections.

Build a network including the neighbors of the infected nodes

```
R> infectNgh <- buildInfectNet(edgeTiming, infectStatus, neighbors = TRUE)
```

Use network plot command, coloring the infected nodes. This is a very large net, will take several minutes.

[17]The types of concurrency are: D one of the seeds, U unknown, S strict concurrency, A concurrent-accelerated/assisted, R residual concurrency, M monogamy.

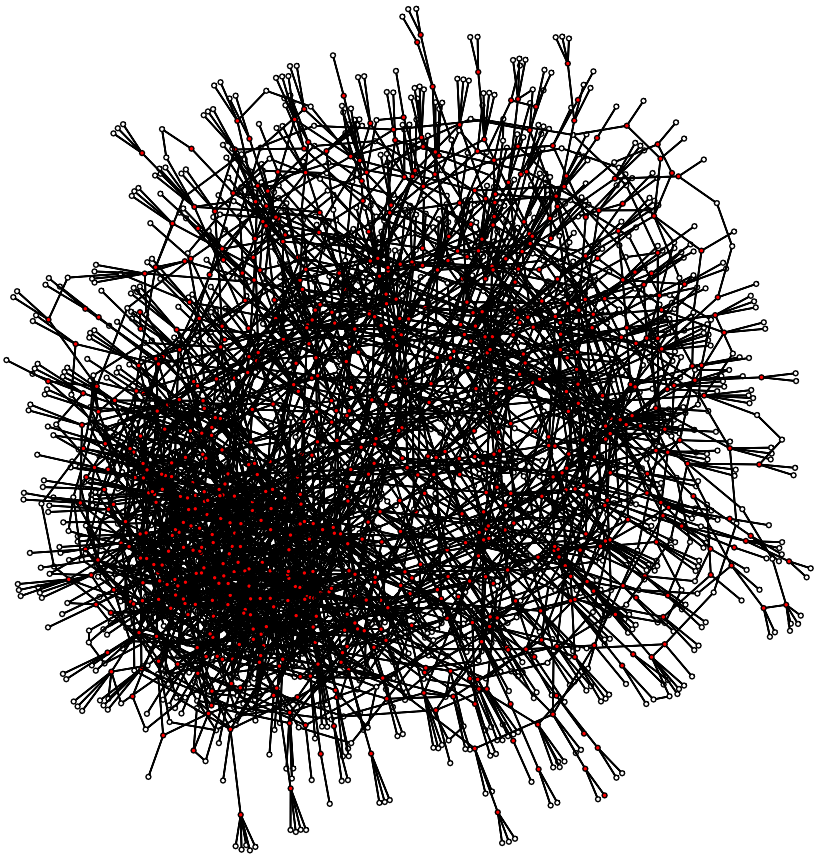

Figure 9: Plot of the 'infected neighborhood' (`infectNgh`) network—a 1449 node sub-sample of the 10000 node simulation network containing all the infected nodes and their immediate neighbors—with timing information discarded.

```
R> plot(infectNgh, vertex.cex = 0.3, edge.lwd = 0.1,
+     vertex.col = hiliteColor(
+     infectStatus$serostatus[infectNgh %v% "vertex.names"]))
```

The graphical output of the plot command is shown in Figure 9.

We can see from the image that there is one very large dense core, and a less dense group or perhaps several smaller groups. We also see many uninfected nodes (colored white) tied to the infected nodes. How can this be if the infection probability was 1.0? Presumably the answer lies in the timing and ordering of the relationships, which are very difficult to discern in a static image (see Figure 10).

Unfortunately, a network of $\sim$ 2500 nodes is more than we can currently effectively deal with in an animation, so we will have to work with the network of just the infected core. In a sense this gives us a biased sample of the overall network, so it is important not to forget the larger context when viewing the following examples.

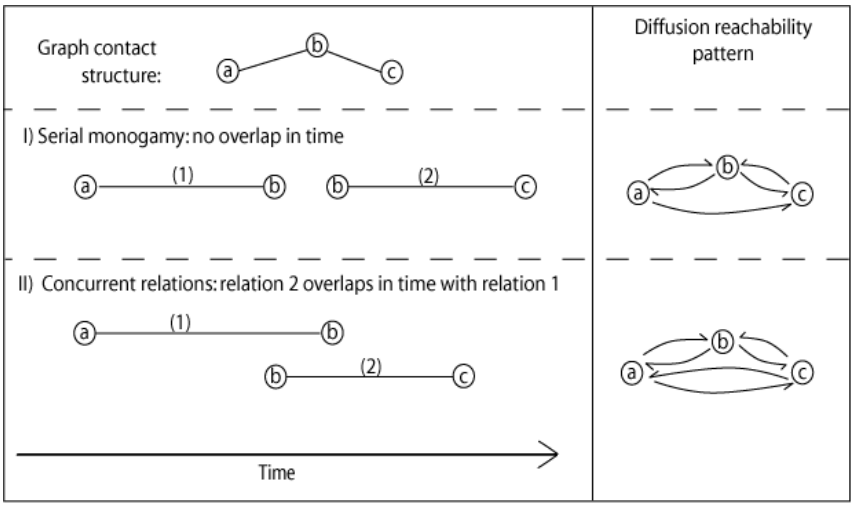

Figure 10: Edgewise connectivity does not necessarily imply temporal connectivity. In panel (I), node a can reach node c, but c cannot reach a since the (ab) edge ends before the (cb) edge starts. When the edges' time intervals overlap (panel II), reachability becomes symmetric. When edge duration is short relative to the full observation window, connectivity can be severely constrained even if the graph appears connected when time is ignored.

8.3. Timing options

In working with the CSDE simulation data we have often found it useful to aggregate networks into time-scales much longer than the simulation time-step in order to show long-term change in structures. The simulation aimed to capture dynamics over ten years and included 3650 daily steps. Since there is very little change in the network each day, we find it effective use bins of length 60 (two months). In some situations we have also manipulated durations of specific kinds of edges in order to highlight important model effects.

Convert the data into a dynamic network but this time does not include the neighbors

```
R> seedNormal <- buildInfectNet(edgeTiming, infectStatus,
+    neighbors = FALSE,
+    keep.infected = FALSE,
+    color.mode = "seed")
```

```
 755 infected nodes
 skipped   19958  out of sample relations
      values
 [1,] "8726" "#FF0000FF"
 (output truncated)
```

Output notes how many relations have been skipped (because we ignore relations among uninfected nodes) and also gives the mapping of color codes to 'strains' of infection. Now set some of the layout options because we will run in batch mode

```
R> soniaApplySettings$kk.opt.dist[2] <- 5
R> soniaApplySettings$kk.comp.connect[2] <- 60
```

Also specify some visual settings to make it look pretty

```
R> soniaGraphicsSettings$layout.height[2] <- 600
R> soniaGraphicsSettings$layout.width[2] <- 600
R> soniaGraphicsSettings$arc.width.fact[2] <- 2
R> soniaGraphicsSettings$node.trans[2] <- 1
R> soniaGraphicsSettings$node.trans[2] <- 1
```

It is a good idea to set the binning criteria here, otherwise **SoNIA** will take a really long time to launch. Set each slice to be two months

```
R> soniaLayoutSettings$slice.duration[2] <- 60
```

But we let the slices overlap by a month

```
R> soniaLayoutSettings$slice.delta[2] <- 30
```

Set the number of in-between frames lower for a faster but jerkier movie

```
R> soniaBrowseSettings$num.frames[2] <- 5
```

and set the render duration to match the slice duration

```
R> soniaBrowseSettings$render.dur[2] <- 60
```

Execute as batch. Note that each movie may take as long as 30 minutes to compute and export

```
R> launchSonia(seedNormal,
+    vertex.col = "color",
+    arc.col = "color",
+    vertex.shape = "shape",
+    vertex.cex = "size",
+    usearrows = FALSE,
+    displaylabels = FALSE,
+    max.iter = 5000,
+    interactive = FALSE,
+    movie.file = file.path(getwd(), "simSeedNormal"))
```

An example of the movie output is simSeedNormal.mov (23MB).

We can see that contrary to our initial view of the network, when we look at it one month at time, the network is hardly connected at al. Most nodes are in dyads with only a few star-shaped components and no large components at all. We can only infer the the chain of infecting events by looking at the colors (nodes with the same color are infected by the same strain, the larger nodes are the initial seeds). Now that we are using only the infected

nodes we can aggregate the entire time period and look ten years worth of relations as a single network to compare the infected regions around each seed node. There is little point in creating a movie since there will only be a single frame, but we can view this network in **SoNIA** by changing the bin criteria. Relaunch **SoNIA** with `interactive = TRUE`.

Set the duration to the entire time

```
R> soniaLayoutSettings$slice.duration[2] <- 3650
```

And make sure we just make one slice

```
R> soniaLayoutSettings$slice.delta[2] <- 3650
```

Edge weights will have changed due to aggregation so we use rescale instead of adjusting the `opt dist` parameter.

```
R> soniaApplySettings$rescale.layout[2] <- "rescale layout to fit window"
```

Relaunch sonia in interactive mode

```
R> launchSonia(seedNormal, vertex.col = "color",  arc.col = "color",
+    vertex.shape = "shape", vertex.cex = "size",  usearrows = FALSE,
+    displaylabels = FALSE, max.iter = 5000, interactive = TRUE)
```

Click `Apply Layout...`, then click the `Apply` button in the dialog and wait for the layout to finish. This produces an interesting image (Figure 11) in which we can see the infected components of the graph, but we still cannot discern the ordering of the infection events.[18]

Notice also the long loops of the graph linked by gray (non-infecting) edges. Another alternative would be to set `keep.infected = TRUE` in the the network processing script, instructing it to adjust the ending times of infecting edges so that we can see the day-to-day changes against a backbone of the 'history' of infection. We can make another movie to demonstrate this effect.

Generate a new input network, extending the times for infected edges

```
R> seedSustain  <- buildInfectNet(edgeTiming, infectStatus, neighbors = FALSE,
+    keep.infected = TRUE, color.mode = "seed")
```

Set the parameters back as they were for the first movie

```
R> soniaLayoutSettings$slice.duration[2] <- 60
R> soniaLayoutSettings$slice.delta[2] <- 30
R> soniaApplySettings$rescale.layout[2] <- "none"
```

Relaunch sonia in interactive mode

```
R> launchSonia(seedSustain, vertex.col = "color",  arc.col = "color",
+    vertex.shape = "shape",  vertex.cex = "size", usearrows = FALSE,
+    displaylabels = FALSE, max.iter = 5000, interactive = FALSE,
+    movie.file = file.path(getwd(), "simSeedSustain"))
```

[18] Actually we cheated, using a `max.iter=10000` to get a high quality image for the paper. All examples can be run with this option, but running time will be proportionately longer.

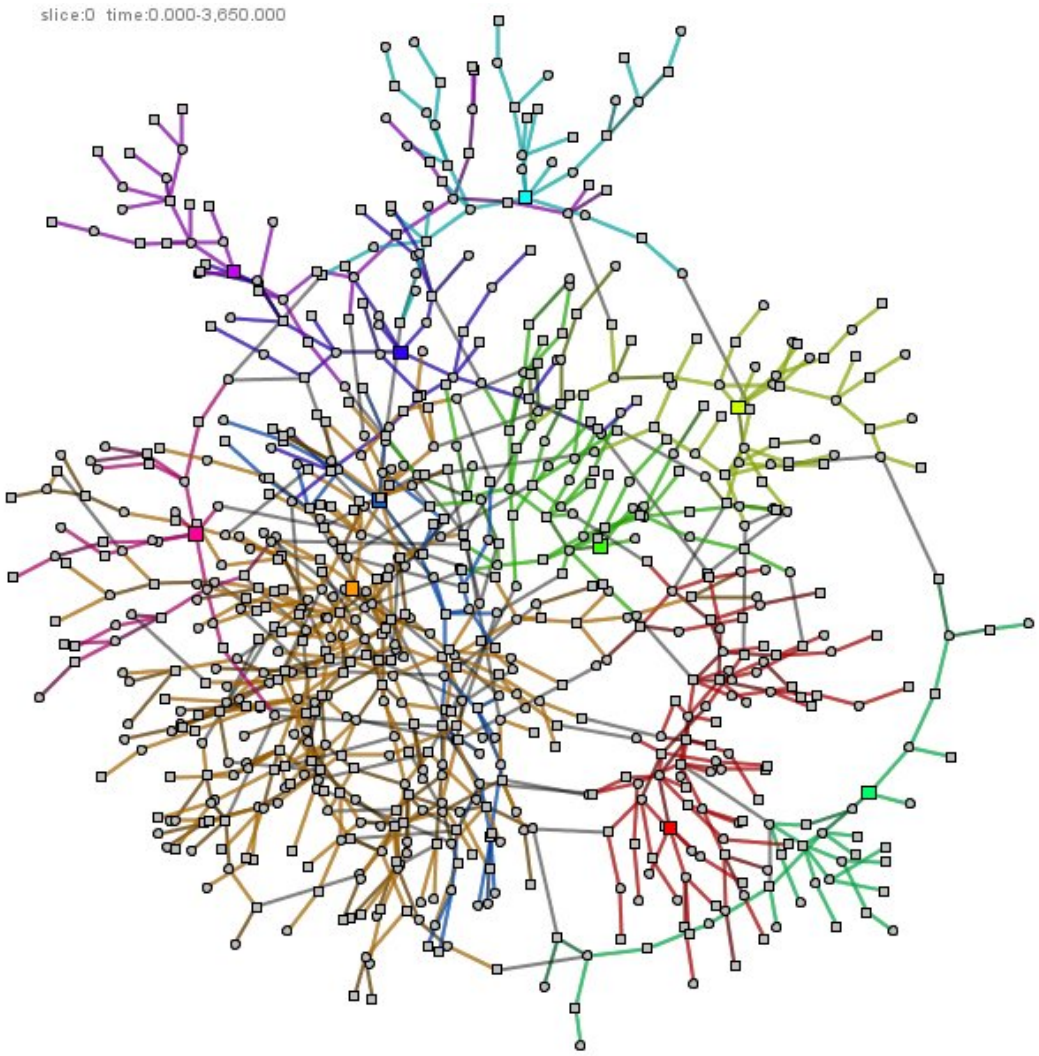

Figure 11: A time-collapsed view of the infected component of the network, with the paths of each infection 'strain' shown in a different color.

An example of the movie output is `simSeedSustain.mov` (38MB).

8.4. Attribute mixing

The model specifications for the simulation included differential mixing on the race attribute, meaning that there should be a lot more connections between nodes with the same race than there are between races. In this graph active edges will be shown in black, and the inactive 'dead' backbone ties will be shown in gray. Race indicated by white and gray node colors, and sex by node shape.

```
R> raceSustain <- buildInfectNet(edgeTiming, infectStatus,
+    neighbors = FALSE, keep.infected = TRUE, color.mode = "race")
```

race attribute	white (8060)	grey (1940)
white (8060)	31072	1151
grey (1940)	1151	8232

Table 1: 'Mixing matrix' for the infection network, showing counts of the numbers of ties between nodes with various combinations of race attributes.

```
R> launchSonia(raceSustain, vertex.col = "color", arc.col = "color",
+    vertex.shape = "shape",vertex.cex = "size", usearrows = FALSE,
+    displaylabels = FALSE, max.iter = 5000, interactive = FALSE,
+    movie.file = file.path(getwd(), "simRaceSustain"))
```

An example of the movie output is SimRaceSustain.mov (37MB). The movie suggests visually that there are few ties between grey and white colored nodes, implying some degree of differential mixing, but it would be better to see some numbers before making this conclusion.

Table 1 confirms that there is strong segregation by race in our simulated network, although there are enough cross-race ties to allow the infections to cross race boundaries as some infection paths can be seen to contain both gray and white nodes.

8.5. Age / generation time

We can also specify that the edges should be colored according to the age (in terms of number of transmission steps) of their infecting strain. This could also be thought of as an indication of the relative probability that each successive event would occur, assuming that the transmission probability was set to a realistic value < 1.0.

```
R> ageSustain <- buildInfectNet(edgeTiming, infectStatus, neighbors = FALSE,
+    keep.infected = TRUE, color.mode = "age")

R> launchSonia(ageSustain, vertex.col = "color",  arc.col = "color",
+    vertex.shape = "shape", vertex.cex = "size", usearrows = FALSE,
+    displaylabels = FALSE, max.iter = 5000, interactive = FALSE,
+    movie.file = file.path(getwd(), "simAgeSustain"))
```

An example of the movie output is simAgeSustain.mov (34MB).

8.6. Concurrency

But the real question we are trying to explore is the relative effects of the different kinds of concurrency in the final prevalence. To get a picture of this, we set up a coloring scheme that maps the relative amounts of 'concurrent' (S, A, R, U) and 'monogamous' (M) transmissions in the the infection history to the blue and red components of the edge colors. So chains that are primarily blue indicate that most of the transmission events took place in monogamous relationships. A red chain indicates that individuals had multiple partners when the transmission occurred. An early concurrent transmission followed by a series of monogamous events results in a chain with a purplish hue.

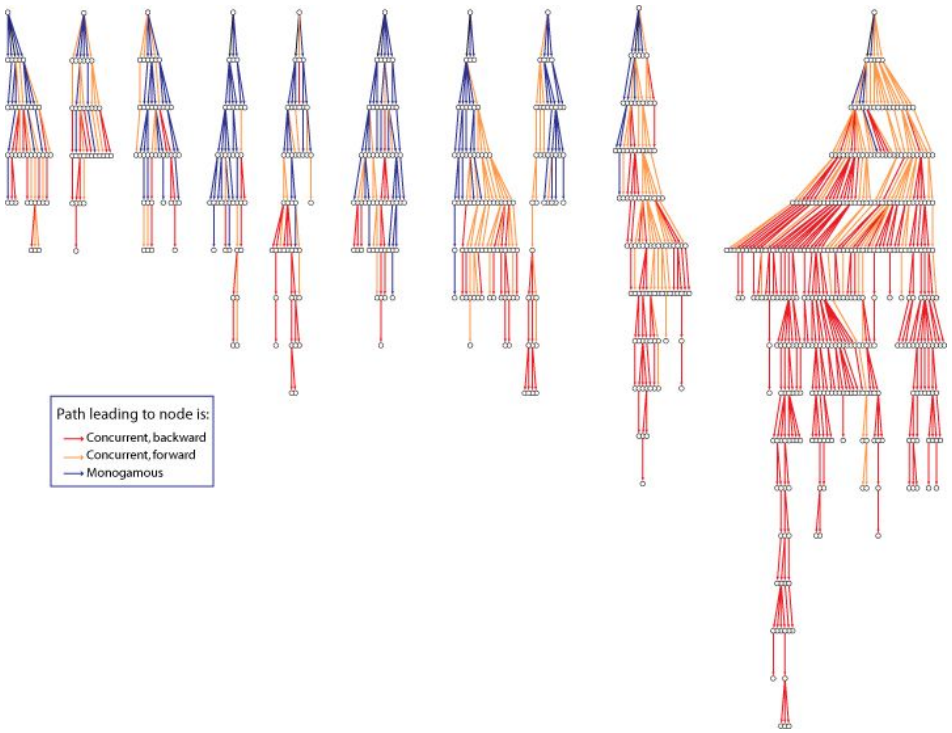

Figure 12: A representation of the infection 'trees' in occurring in the simulated network. Time advances vertically down the page, so the seed nodes for each infection appear at the top and the 'depth' of the tree indicates the time step of the infection. The color of each edge indicates the concurrency status of the corresponding relationship when the transmission occurred.

```
R> concurSustain <- buildInfectNet(edgeTiming, infectStatus, neighbors = FALSE,
+    keep.infected = TRUE, color.mode = "chainletter")

R> launchSonia(concurSustain, vertex.col = "color",  arc.col = "color",
+    vertex.shape = "shape", vertex.cex = "size", usearrows = FALSE,
+    displaylabels = FALSE, max.iter = 5000, interactive = FALSE,
+    movie.file = file.path(getwd(), "simConcurSustain"))
```

For the final movie we animated a transition from the last time step of the movie to a set of coordinates that position the nodes as a time-ordered tree with the seed node at the top (Figure 12) to show an alternate view that reinforces our understanding of the network's time structure and the ordering of infection events.

An example of the movie output is `simConcurSustain.mov` (32MB). For a more refined example, see `10seed05take3.mov` (77MB).

9. Discussion and next steps

These packages require a number of improvements before they are ready for regular research

use. The functionality demonstrated here needs to be made far more robust and several design and theoretical issues need to be solved. In addition to the speed and efficiencies that could be gained by moving most of the heavy lifting from R into C library code, much more sophisticated C-based structures can be used in the back-end. This would also help with some of the ordering and update/deletion problems that can arise in the existing **network** and **dynamicnetwork** packages. The examples shown here all used momentary slices through a dynamic network to create cross-sectional networks. A more advanced package should include **SoNIA**-like features to permit aggregation over arbitrary intervals and generate appropriate values for edges weights and attributes. More sophisticated definitions of equality and equivalence relations between corresponding edges and relations in network objects would be useful. We do not have a system of notation for directly referring to an instance of an edge between a pair of nodes at giving time. This is also an issue when dealing with multiplex edges in static networks. Eventually the features should be integrated into the basic network package.

References

Adobe Systems Incorporated (1999). *Adobe **Flash** Player*. URL http://www.adobe.com/shockwave/download/.

Apple Inc (1999). ***QuickTime** Multimedia Software Architecture*. URL http://developer.apple.com/quicktime/.

Batagelj V, Mrvar A (2007). ***Pajek**: Package for Large Network Analysis*. University of Ljubljana, Slovenia. URL http://vlado.fmf.uni-lj.si/pub/networks/pajek/.

Bender-deMoll S, McFarland DA (2003). ***SoNIA**: Social Network Image Animator*. URL http://sonia.stanford.edu/.

Bender-deMoll S, McFarland DA (2006). "The Art and Science of Dynamic Network Visualization." *Journal of Social Structure*, **7**(2).

Borgatti SP, Everett MG, Freeman LC (1999). ***UCINET** 6.0 for Windows: Software for Social Network Analysis*. Analytic Technologies, Natick. URL http://www.analytictech.com/.

Butts CT (2006). ***diffusion**: Tools for Simulating Network Diffusion Processes*. R package, URL http://erzuli.ss.uci.edu/R.stuff/.

Butts CT (2007). ***sna**: Tools for Social Network Analysis*. Statnet Project http://statnetproject.org/, Seattle, WA. R package version 1.5, URL http://CRAN.R-project.org/package=sna.

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

Butts CT, Handcock MS, Hunter DR (2007). ***network**: Classes for Relational Data*. Statnet Project http://statnetproject.org/, Seattle, WA. R package version 1.3, URL http://CRAN.R-project.org/package=network.

CERN (1999). ***COLT*** *Open Source Libraries for High Performance Scientific and Technical Computing in Java*. European Organization for Nuclear Research. URL http://dsd.lbl.gov/~hoschek/colt/.

Ferreira A, Viennot L (2002). "A Note on Models, Algorithms, and Data Structures for Dynamic Communication Networks." *Rapport de recherche 4403*, Institute National de Recherche en Informatique et en Automatique.

FreeHEP (2000). ***FreeHEP*** *Vector Graphics Java Package*. URL http://java.freehep.org/vectorgraphics/.

Handcock MS, Hunter DR, Butts CT, Goodreau SM, Morris M (2003). ***statnet****: Software Tools for the Statistical Modeling of Network Data*. Statnet Project http://statnetproject.org/, Seattle, WA. R package version 2.0, URL http://CRAN.R-project.org/package=statnet.

Hunter DR (2007). "Curved Exponential Family Models for Social Networks." *Social Networks*, **29**, 216–230.

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

Kamada T, Kawai S (1989). "An Algorithm for Drawing General Undirected Graphs." *Information Processing Letters*, **31**(1), 7–15.

Moody J, McFarland D, Bender-deMoll S (2005). "Dynamic Network Visualization." *American Journal of Sociology*, **110**(4), 1206–1241.

Morris M, Kretzschmar M (1997). "Concurrent Partnerships and the Spread of HIV." *AIDS*, **11**(641–648).

Morris M, Kretzschmar M (2000). "A Micro-Simulation Study of the Effect of Concurrent Partnerships on HIV Spread in Uganda." *Mathematical Population Studies*, **8**(2), 109–133.

Newcomb TM (1961). *The Acquaintance Process*. Holt, Rinehart, Winston., New York.

R Development Core Team (2007). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, Version 2.6.1, URL http://www.R-project.org/.

Resnick MD, Bearman PS, Blum RW, Bauman KE, Harris KM, Jones J, Tabor J, Beuhring T, Sieving RE, Shew M, Ireland M, Bearinger LH, Udry JR (1997). "Protecting Adolescents from Harm. Findings from the National Longitudinal Study on Adolescent Health." *Journal of the American Medical Association*, **278**(10), 823–832.

Sampson SF (1968). *A Novitiate in a Period of Change: An Experimental and Case Study of Social Relationships*. Ph.d. thesis (university micofilm, no 69-5775), Department of Sociology, Cornell University, Ithaca, New York.

Tsvetovat M, Reminga J, Carley KM (2004). *DyNetML: Interchange Format for Rich Social Network Data*. Carnegie Mellon University, School of Computer Science, Institute for Software Research International.

Udry JR (2003). "The National Longitudinal Study of Adolescent Health (Add Health), Waves I & II, 1994–1996; Wave III, 2001–2002." *Technical report*, Carolina Population Center, University of North Carolina at Chapel Hill.

Zachary WW (1977). "An Information Flow Model for Conflict and Fission in Small Groups." *Journal of Anthropological Research*, **33**(4), 452–473.

A. Major new functions

A.1. Descriptions and purpose of the major new functions added

In order to give an idea of the kinds of functions we found immediately useful or necessary when working with dynamics, we have included names and short descriptions of the functions in **dynamicnetwork**

`as.dynamic.network()` Takes a network object, makes a copy and adds the necessary elements to make it a `dynamicnetwork` object describing a single region of time. The default is for edges to have the valid interval [0,1].

`as.dynamic.list()` Converts a list of networks to a dynamic network. First copies the data in the first network on the list, then looks through all the other networks, adding edges and modifying edges' valid intervals as necessary. Assumes that the edges in the first network have the valid interval [0,1], that all intervals have unit duration, and that the appearance of an edge in immediately adjacent networks means that the edge was continuously active.

`as.dynamic.intervals()` Converts a set of valid intervals in the form of four-column matrix with columns source, target, start, end, into a `dynamicnetwork` object.

`as.dynamic.changelist()` Converts a change list matrix (set of toggles) into a dynamic network. The input is a (possibly empty) network object giving the initial state of the graph, followed by a three column matrix (time step, head node, tail node) There is one row for each changed dyad over the entire series (i.e., time to the last time). Values listed under time step i are the changes made to the network at time step i to make it the network at time i+1. They are listed in time step order. This is the method used to convert the output of a ERGM simulation.

When the `subsample` argument is set to a vector of node ids, only relations involving those ids will be inserted into the the network, making it possible to easily extract a sub-sample of relations from a much larger network.

`add.dynamic.edges()` Adds an edge to the network that is only 'active' during the specified time interval. Edge are added using the normal `add.edge` internal call, but the time values `c(start,end)` are stored on an Edge Time List (`x$etl`) that has entries in the same order as the mel list.

`get.edge.start(), get.edge.end(), set.edge.start,set.edge.end()` returns or sets the timing information attached to an edge

`get.slice.network()` Returns a network object that contains the appropriate edges and attribute values for the given time. Dynamic attributes will be removed and replaced with a static attribute with same name and appropriate value.

`get.edge.ids.at()` A means of getting the ids of edges in the neighborhood of a specified node at a specified time. May return multiple edges if the network is multiplex. Kind of a work around for the fact that we do not have a good way of addressing edges in multiplex dynamic networks.

`get.edge.ids.before()` Same as get edges at, but returns the most recent edge before or intersecting with the specified time.

`get.matching.edgeIDs()` Given two networks and an edge ID in the first, finds the id(s) of edges in the second with matching head- and tail-sets (i.e. source and target nodes for most networks)

`get.edges.difference()` Given two networks, returns a vector of edgeIDs for edges in the second network that do not have a corresponding edge in the first. Mostly used for comparing the same network at two points in time in order to get a list of newly created edges.

`get edges intersection()` Given two networks, returns a vector of edgeIDs for edges in the second network that have a corresponding edge in the first network. Mostly used for comparing the same network at two points in time to find the edges that have not changed.

`set.dynamic.edge.attribute(), set.dynamic.vertex.attribute()` Specifies that the named attribute in the dynamic network should take on the given value at the given time and for all later times until the next value.

This is implemented by adding attributes to `val` that are two-column matrices (`[value, valid.time]`) with rows equal to the number of changes. So adding additional values adds more rows to the matrix (deletion behavior not defined, but should be OK). These attributes are considered dynamic attributes, defined by having the vertex name listed in the graph-level attribute `dynam.attr.names`. Uses the internal `set.vertex.attribute` method to set the value. The attribute is assumed to have the same value from `valid.time` until the next specified time for the named attribute on a vertex. This is a very naive implementation that assumes the variables will be added in increasing time order. Violating this should give a warning and then strange results.

`is.dynamic.edge.attribute(), is.dynamic.vertex.attribute()` checks if an attribute name is dynamic, by looking in it the appropriate list of attributes.

`get.verts.with()` gets a set of ids corresponding to vertices that have the specified value for the named attribute at the specified time.

Affiliation:

Skye Bender-deMoll
Developer & Network Visualization Consultant
E-mail: skyebend@skyeome.net
URL: http://skyeome.net/wordpress/

Martina Morris
University of Washington
URL: http://faculty.washington.edu/morrism/

James Moody
Duke University
URL: http://www.soc.duke.edu/~jmoody77/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 24, Issue 7 *Submitted:* 2007-06-01
February 2008 *Accepted:* 2007-12-25