Package Vignette for ndtv: Network Dynamic Temporal Visualizations (Version 0.2)

Skye Bender-deMoll

May 31, 2013

Contents

1 Introduction

The Network Dynamic Temporal Visualization (`ndtv`) package provides tools for visualizing changes in network structure and attributes over time. It works with dynamic network information encoded in `networkDynamic` (Butts et al. , 2013) objects as its input, and outputs animated movies. The package will eventually include timelines and other types of dynamic visualizations of evolving relational structures. The core use-case for development is examining the output of statistical network models (such as those produced by the `tergm` (Krivitsky et al. , 2013) package in `statnet` (Handcock et al , 2003)) and simulations of disease spread across networks. The `ndtv` (Bender-deMoll , 2013) package relies on many other packages to do much of the heavy lifting, especially `animation` (Yihui, Xie et al. , 2013) and `networkDynamic` and requires external libraries (FFmpeg) to save movies out of the R environment. To use `ndtv` effectively you must be already familiar with the functionality and assumptions of `networkDynamic`. This package is intended to eventually replace much of the functionality in the `rSoNIA` package (Bender-deMoll et al., 2008).

This work was supported by grant R01HD68395 from the National Institute of Health.

2 A quick example

2.1 Reinventing the wheel

Lets get started! We can render a trivially simple animation in the R plot window.

```
> library(ndtv) # also loads animation and networkDynamic
> wheel <- network.initialize(10)  # create a toy network
> add.edges.active(wheel,tail=1:9,head=c(2:9,1),onset=1:9, terminus=11)
> add.edges.active(wheel,tail=10,head=c(1:9),onset=10, terminus=12)
> plot(wheel) # peek at the static version
> render.animation(wheel) # compute and render
```

```
[1] "No slice.par found, using"
slice parameters:
  start:1
  end:12
  interval:1
  aggregate.dur:1
  rule:any
```

```
[1] "Calculating layout for network slice from time  1 to 2"
[1] "Calculating layout for network slice from time  2 to 3"
[1] "Calculating layout for network slice from time  3 to 4"
[1] "Calculating layout for network slice from time  4 to 5"
[1] "Calculating layout for network slice from time  5 to 6"
[1] "Calculating layout for network slice from time  6 to 7"
[1] "Calculating layout for network slice from time  7 to 8"
[1] "Calculating layout for network slice from time  8 to 9"
[1] "Calculating layout for network slice from time  9 to 10"
[1] "Calculating layout for network slice from time  10 to 11"
[1] "Calculating layout for network slice from time  11 to 12"
[1] "Calculating layout for network slice from time  12 to 13"
[1] "rendering 10 frames for slice 0"
[1] "rendering 10 frames for slice 1"
[1] "rendering 10 frames for slice 2"
[1] "rendering 10 frames for slice 3"
[1] "rendering 10 frames for slice 4"
[1] "rendering 10 frames for slice 5"
[1] "rendering 10 frames for slice 6"
[1] "rendering 10 frames for slice 7"
[1] "rendering 10 frames for slice 8"
[1] "rendering 10 frames for slice 9"
[1] "rendering 10 frames for slice 10"
[1] "rendering 10 frames for slice 11"
```

```
> ani.replay() # play back in plot window
```

Hopefully, when you ran `ani.replay()` you saw a bunch of labeled nodes moving smoothly around in the R plot window, with edges slowly appearing to link them into a circle. Finally a set of "spoke" edges appear to draw a vertex into the center. If that didn't work, the footnote has a link to an example of the movie [1] you are supposed to see. For some kinds of networks the animated version gives a very different impression of the connectivity of the network that a static plot of the same network (Figure 1)

2.2 What just happened?

Simple right? Yes, but that is because most of the difficult parts happened under the hood using default values. In a nutshell, this is how it worked:

1. We created a `networkDynamic` object named `wheel` containing information about the timing of edge activity.

2. `render.animation()` asked the package to create an animation for `wheel` but we didn't include any arguments indicating what should be rendered or how.

[1] `http://statnet.csde.washington.edu/movies/ndtv_vignette/wheel.mp4`

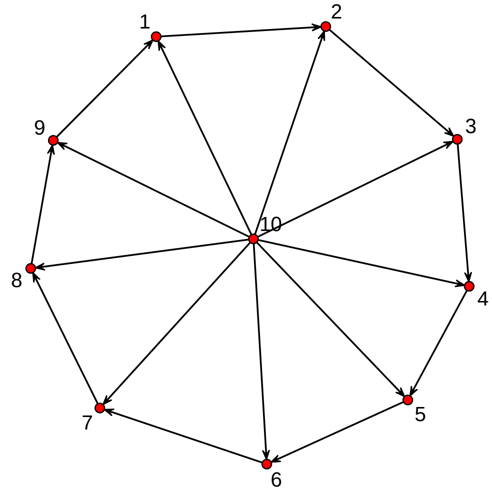


Figure 1: Standard network plot of our trivial "wheel" network does not reveal dynamics. Compare with animated movie version: `http://statnet.csde.washington.edu/movies/ndtv_vignette/wheel.mp4`

3. Since `render.animation()` didn't find any stored coordinate information about where to draw the vertices and edges, it (invisibly) called `compute.animation()` with default arguments to figure out where to position the vertices at each time step.

4. Because we didn't tell `compute.animation()` what time points to look at when doing its computations, it reported this, `"No slice.par found"`, and made a guess as to when the animation should start and end (the earliest and latest observed times in the network) and how much time should be incremented between each set of layout coordinate calculations.

5. `compute.animation()` then stepped through the `wheel` network, computing coordinates for each time step and storing them. (This was the `"Calculating layout for network slice from time 1 to 2"` ... part.)

6. `render.animation()` also stepped through the network, using the stored coordinates, `plot.network()` and `ani.record()` functions to cache snapshots of the network. It also caches a number of "tweening" images between each time step to smoothly interpolate the positions of the vertices. `"rendering 10 frames for slice 1"` ...

7. `ani.replay()` quickly redraws the sequence of cached images in the plot window as an animation.

Of course, using defaults doesn't give much control of what should be rendered and how it should look. For more precise control of the processes, layout algorithms, etc, we can call each of the steps in sequence.

3 A tergm simulation example

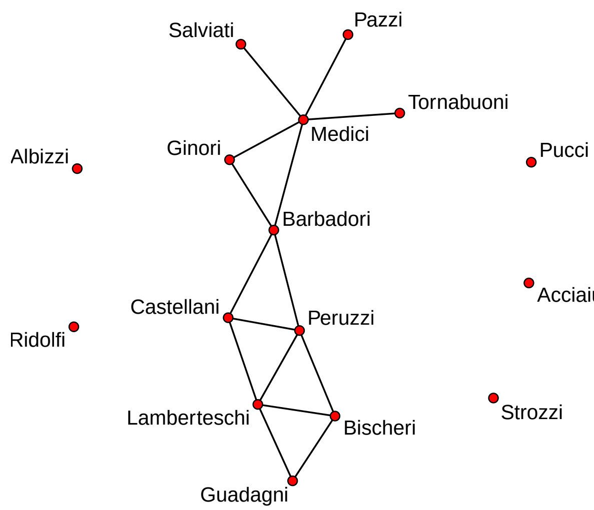
Lets look at a more realistic example using output from the simulation of a crude dynamic model. This uses the statnet `tergm` package to estimate the parameters for an edge formation and dissolution process which produces a network similar to the Florentine business network (`?ergm::flobusiness`) given as input. Once the model has been estimated, we can take a number of sequential draws from it to see how the network might "evolve" over time. When we generate the movie, we can include the model statistics on screen to see how they are influenced by edge additions and deletions. This example also assumes you have some of the external libraries working (Java and FFmpeg) so you run into problems, try skipping to Dependencies (section 8) and come back.

First load in the main necessary libraries (each of which loads a bunch of additional R libraries).

```
> require(ndtv)      # dynamic network animations
> require(tergm)     # dynamic ergm simulations
```

Load in the original Florentine business network.

```
> data("florentine") # an example network
> plot(flobusiness,displaylabels=T)
```



Define basic `stergm` model with formation and dissolution parameters.

```
> theta.diss <- log(9)
> stergm.fit.1 <- stergm(flobusiness,
+   formation= ~edges+gwesp(0,fixed=T),
+   dissolution = ~offset(edges),
+         targets="formation",
+   offset.coef.diss = theta.diss,
+   estimate = "EGMME"        )
```

(time passes, lots simulation status output hidden)

Now we can simulate 100 discrete time steps from the model and save them as a `dynamicNetwork` object.

```
> stergm.sim.1 <- simulate.stergm(stergm.fit.1,
+                     nsim=1, time.slices = 100)
```

Since this isn't a terribly exciting simulation, lets only calculate coordinates for part of the simulated time period by using the `start` and `end` parameters of `slice.par` to specify a time range. We can also ask it to use the MDSJ layout (assuming MDSJ and Java are installed).

```
> slice.par<-list(start=75,end=100,interval=1,
+                 aggregate.dur=1,rule="any")
> compute.animation(stergm.sim.1,slice.par=slice.par,
+                                 animation.mode='MDSJ')
```

Now that we have all the coordinates stored, we can define some parameters for `render.par` to specify how many `tween.frames` to render, and tell it to display the time and the summary statistics formula.

```
> render.par=list(tween.frames=5,show.time=T,
+                 show.stats="~edges+gwesp(0,fixed=T)")
```

Then we ask it to build the animation, passing in some of the standard `plot.network` graphics arguments to change the color of the edges and show the labels with a smaller size and blue color.

```
> render.animation(stergm.sim.1,render.par=render.par,
+                  edge.col="darkgray",displaylabels=T,
+                  label.cex=.6,label.col="blue")
```

This takes some time and produces many lines output which we are not showing. The output could also be suppressed by adding a `verbose=FALSE` argument.

After it has finished, replay the movie in an R plot window.

```
> ani.replay()
```

Notice that in addition to the labels on the bottom of the plot indicating which time step is being viewed, it also displays the network statistics of interest for the time step. When the 'edges' parameter increases up, you can see the density on the graph increase and the number of isolates decrease. Eventually the model corrects, and the parameter drifts back down.

We can also use the `animation` library to save out the movie in `.mp4` format (assuming that the FFmpeg library is installed on your machine).

```
> saveVideo(ani.replay(),video.name="stergm.sim.1.mp4",
+                      other.opts="-b 5000k",clean=TRUE)
```

This should produce a movie[2] in an R working directory on disk. The `other.opts` parameter is set here to generate a higher-quality video than the default, but this will result in a large file size. For more information on compressing videos for the web, see Compressing Video (section 9).

4 Slicing time

The basic network layout algorithms we are using, like most "traditional' network metrics, don't really know what to do with dynamic networks. They need

[2] `http://statnet.csde.washington.edu/movies/ndtv_vignette/stergm.sim.1.mp4`

to be fed a static set of relationships which can be used to compute a set of distances in a Euclidean space suitable for plotting. A common way to apply static metrics to a time-varying object is to sample it, taking a sequence static observations at a series of time points and using these to describe the changes over time. In the case of networks, we call this “extracting’ or “slicing’.

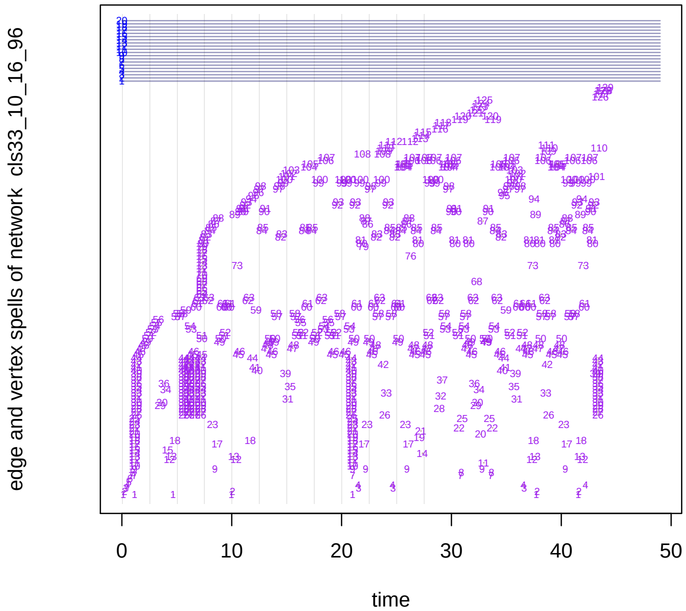
Slicing up a dynamic network created from discrete panels may be fairly straightforward but it is much less clear how to do it when working with continuous time or streaming relations. How often should we slice? Should the slices measure the state of the network at a specific instant, or aggregate over a longer time period? The answer probably depends on what the important features to visualize are in your data-set. The `slice.par` parameters make it possible to experiment with various slicing options. In many situations we have even found (Bender-deMoll and McFarland , 2006) it useful to let slices mostly overlap – incrementing each one by a small value to help show fluid changes on a moderate timescale instead of the rapid changes happening on a fine timescale.

As an example, lets look at the McFarland (McFarland , 2001) data-set of streaming classroom interactions and see what happens when we chop it up in various ways. First, we can animate at the fine time scale, viewing the first half-hour of class using instantaneous slices.

```
> data(McFarland_cls33_10_16_96)
> slice.par<-list(start=0,end=30,interval=2.5,
+                 aggregate.dur=0,rule="any")
> compute.animation(cls33_10_16_96,
+                 slice.par=slice.par,animation.mode='MDSJ')
> render.animation(cls33_10_16_96,
+                  displaylabels=FALSE,vertex.cex=1.5)
> ani.replay()
```

We can also get an idea of how we are slicing up the network by using the `timeline()` function to plot the `slice.par` parameters against the vertex and edge spells. Our very thin slices (gray virtical lines) (`aggregate.dur=0`) are not intersecting may edge events (purple numbers) at once.

```
> timeline(cls33_10_16_96,slice.par=slice.par)
```



Notice that in the animation most of the vertices are isolates, occasionally linked into brief pairs or stars by speech acts[3]. However, if we aggregate over a longer time period of 2.5 minutes we start to see the individual acts form into triads and groups[4].

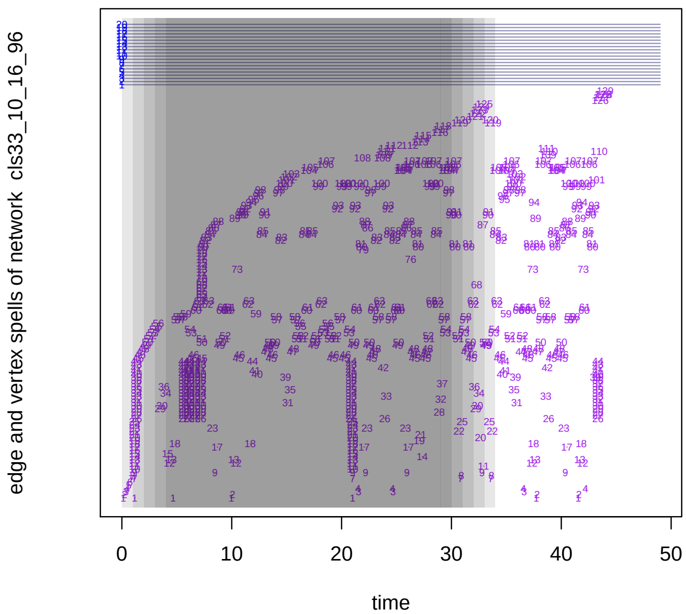
```
> slice.par<-list(start=0,end=30,interval=2.5,
+                 aggregate.dur=2.5,rule="any")
> compute.animation(cls33_10_16_96,
+                 slice.par=slice.par,animation.mode='MDSJ')
> render.animation(cls33_10_16_96,
+                  displaylabels=FALSE,vertex.cex=1.5)
> ani.replay()
```

To reveal slower structural patterns we can make the aggregation period even longer, and let the slices overlap (by making `interval` less than `aggregate.dur`) so that the changes will be less dramatic between successive views[5].

```
> slice.par<-list(start=0,end=30,interval=1,
+                 aggregate.dur=5,rule="any")
> timeline(cls33_10_16_96,slice.par=slice.par)
```

[3] http://statnet.csde.washington.edu/movies/ndtv_vignette/cls33_10_16_96v1.mp4
[4] http://statnet.csde.washington.edu/movies/ndtv_vignette/cls33_10_16_96v2.mp4
[5] http://statnet.csde.washington.edu/movies/ndtv_vignette/cls33_10_16_96v3.mp4

```
> compute.animation(cls33_10_16_96,
+                  slice.par=slice.par,animation.mode='MDSJ')
> render.animation(cls33_10_16_96,
+                   displaylabels=FALSE,vertex.cex=1.5)
> ani.replay()
```



Note that when we use a long duration slice, it is quite likely that there is more than one edge between a pair of vertices. How should this condition be handled? Ideally we might want to aggregate the two edges in some way, perhaps adding the weights together or drawing them in some overlapping way. Although we anticipate adding these features soon, currently it is handled in the kludgy way of only considering the first (earliest) edge when multiple edges are encountered.

5 Layout algorithms

Producing "good' (for an admittedly ambiguous definition of 'good') layouts of networks is generally a computationally difficult problem. There are a wide variety of algorithms and approaches being developed. Doing layouts for animations adds additional challenges because it is usually desirable that the layouts remain stable over time. Ideally this means that the layouts don't change much unless the network structure changes, and that small changes in the network structure should lead to small changes in the layouts. Many otherwise excellent static layout algorithms are not stable in this sense, or they may require very specific parameter settings to improve their results for animation applications.

The `network.layout.animate.*` layouts included in `ndtv` are adaptations or wrappers for existing static layout algorithms with some appropriate parameter presets. They all accept the coordinates of the previous layout as an argument so that they can try to construct a suitably smooth sequence of node positions. They also include the `default.dist` parameter which can be tweaked to increase or decrease the spacing between isolates and disconnected components. The default value for `default.dist` is `sqrt(network.size(net))`, see `?layout.dist` for more information.

It is important to remember that there are many types of networks for which these methods will probably not produce useful visualizations. We've had the most success with networks that are fairly sparse, where a relatively small number of ties are changing between time slices, and node turnover is not too high.

5.1 Kamada-Kawai adaptation

The function `network.layout.animate.kamadakawai` is essentially a wrapper for `network.layout.kamadakawai`. It computes a symmetric geodesic distance matrix from the input network (replacing infinite values with `default.dist`), and seeds the initial coordinates for each slice with the results of the previous slice in an attempt to find solutions that are as close as possible to the previous positions. It is not as fast as MDSJ, and the layouts it produces are not as smooth. But it has the advantage of being written entirely in R, so it doesn't have the pesky external dependencies of MDSJ. For this reason it is the default layout algorithm.

5.2 MDSJ (Multidimensional Scaling for Java)

According to its authors:

> MDSJ (MDSJ , 2009) is a free Java library for Multidimensional Scaling (MDS). It is a free, non-graphical, self-contained, lightweight implementation of basic MDS algorithms and intended to be used both as a standalone application and as a building block in Java based data analysis and visualization software.

MDSJ is a very efficient implementation of MDS so `network.layout.animate.MDSJ` gives the best performance of any of the algorithms tested so far – despite the overhead of writing matrices out to a Java program and reading coordinates back in. Like all of the MDS-variants, MDSJ will check and give errors if you try to call it with a non-symmetric distance matrix. Currently `max_iter` is the only user argument that is passed through to the Java wrapper. It controls the maximum number of optimization steps. The default value is 50 which is usually sufficient. But it can be increased for layouts that appear to be not entirely converging, or perhaps decreased to save some speed on simpler layouts.

Please note that the MDSJ library is released under Creative Commons License "by-nc-sa" 3.0. This means using the algorithm for commercial purposes would be a violation of the license. Due to CRAN's license restrictions, the MDSJ binary is not distributed along with the (GPL-licensed) `ndtv` package. Instead, the first time the layout is called, it will ask if you want to automatically download and install the library. More information about the MDSJ library and its licensing can be found at `http://www.inf.uni-konstanz.de/algo/software/mdsj/`.

5.3 Use a TEA attribute

The `useAttribute` layout is useful if you already know exactly where each vertex should be drawn at each timestep, and you just want to render out the network. It just needs to know the names of the dynamic attribute holding the x coordinate and the y coordinate for each time step.

5.4 User-generated layout functions

We can define new layout functions by following the appropriate naming structure. For example, if we wanted a layout that just arranged all the active vertices in a circle we could define a new function `network.layout.animate.circle`.

```
> network.layout.animate.circle <- function(net, dist.mat = NULL,
+         default.dist = NULL, seed.coords = NULL, layout.par = list(),
+         verbose=FALSE){
+
+     n<-network.size(net)
+     x<-10*cos( seq(0,2*pi, length.out=n))
+     y<-10*sin( seq(0,2*pi, length.out=n))
+     return(cbind(x,y))
+   }
```

We can then re-compute a new animation for the simulation output using our new "circle' layout function.

```
> stergm.sim.1<-compute.animation(stergm.sim.1,
+               slice.par=slice.par,animation.mode='circle')
> render.animation(stergm.sim.1)
> ani.replay()
```

5.5 Other techniques

We have tested some layouts using R libraries for doing SMACOF (de Leeuw , 2009) and standard MDS optimization. The former gave high-quality results but was extremely slow, the later often didn't give stable results. Both may be included in future releases of `ndtv` if the performance issues improve.

6 Vertex dynamics

Edges are not the only things that can change in networks. In some dynamic network data-sets vertices also enter or leave the network (become active or inactive). Lin Freeman's windsurfer social interaction data-set (Almquist et all, 2011) is a a good example of this. In this data-set there are different people are present on the beach on different days, and there is even a day of missing data. These networks also have a lot of isolates, which tends to scrunch up the rest of the components so they are hard to see. Setting a lower `default.dist` can help with this.

```
> data(windsurfers)
> slice.par<-list(start=1,end=31,interval=1,
+                 aggregate.dur=1,rule="any")
> windsurfers<-compute.animation(windsurfers,slice.par=slice.par,
+                                default.dist=3,
+                                animation.mode='MDSJ',
+                                verbose=FALSE)
> render.animation(windsurfers,vertex.col="group1",
+                  edge.col="darkgray",
+                  displaylabels=TRUE,label.cex=.6,
+                  label.col="blue", verbose=FALSE)
> ani.replay()
```

In this example[6] the turnover of people on the beach so so great that structure appears to change chaotically, and it is quite hard to see what is going on. Notice also the blank period at day 25 where the network data is missing. There is also a lot of periodicity, since a lot more people go to the beach on weekends. So in this case, lets try a week-long slice by setting `aggregate.dur=7` to try to smooth it out so we can see some structure.

```
> slice.par<-list(start=0,end=24,interval=1,
+                 aggregate.dur=7,rule="any")
> windsurfers<-compute.animation(windsurfers,slice.par=slice.par,
+                                default.dist=3,
+                                animation.mode='MDSJ',
+                                verbose=FALSE)
> render.animation(windsurfers,vertex.col="group1",
+                  edge.col="darkgray",
+                  displaylabels=TRUE,label.cex=.6,
+                  label.col="blue", verbose=FALSE)
> ani.replay()
```

This new rolling–"who interacted this week" network[7] is larger and more dense (which is to be expected) and also far more stable. There is still some

[6] http://statnet.csde.washington.edu/movies/ndtv_vignette/windsurfers_v1.mp4
[7] http://statnet.csde.washington.edu/movies/ndtv_vignette/windsurfers_v2.mp4

turnover due to people who don't make it to the beach every week but is possible to see some of the sub-groups and the the various bridging individuals.

7 Animating attributes

If a network has dynamic attributes defined, they can be used to define graphic properties of the network which change over time. We can activate some attributes on our earlier "wheel" example, setting a dynamic attribute for edge widths:

```
> activate.edge.attribute(wheel,'width',1,onset=0,terminus=3)
> activate.edge.attribute(wheel,'width',5,onset=3,terminus=7)
> activate.edge.attribute(wheel,'width',10,onset=3,terminus=Inf)
```

We must make sure the attributes are always defined for each time period that the network will be plotted or else an error will occur. So we first set a default value from `-Inf` to `Inf` before defining which elements we wanted to take a special value.

```
> activate.vertex.attribute(wheel,'mySize',1, onset=-Inf,terminus=Inf)
> activate.vertex.attribute(wheel,'mySize',3, onset=5,terminus=10,v=4:8)
```

We can set values for vertex colors.

```
> activate.vertex.attribute(wheel,'color','gray',onset=-Inf,terminus=Inf)
> activate.vertex.attribute(wheel,'color','red',onset=5,terminus=6,v=4)
> activate.vertex.attribute(wheel,'color','green',onset=6,terminus=7,v=5)
> activate.vertex.attribute(wheel,'color','blue',onset=7,terminus=8,v=6)
> activate.vertex.attribute(wheel,'color','pink',onset=8,terminus=9,v=7)
```

Finally we render it, giving the names of the dynamic attributes to be used to control the plotting paramters for edge with, vertex size, and vertex color.

```
> render.animation(wheel,edge.lwd='width',vertex.cex='mySize',
+                  vertex.col='color',verbose=FALSE)
> ani.replay()
```

The attribute values for the time points are defined using `network.collapse`, which controls the behavior if multiple values are active for the plot period.

8 Dependencies

8.1 Java (for MDSJ)

In order to use the MDSJ layout algorithm, you must have Java installed on your system. Java should be already installed by default on most Mac and Linux systems. If it is not installed, you can download it from `http://www.java.com/en/download/index.jsp`. On Windows, you may need to edit your 'Path' environment variable to make Java executable from the command-line.

8.2 FFmpeg

FFmpeg `http://ffmpg.org` is a cross-platform tool for converting and rendering video content in various formats. It is used as an external library by the `animation` package to save out the animation as a movie file on disk. (see `?saveVideo` for more information.) Since FFmpeg is not part of R, you will need to install it separately on your system for the save functionality to work. The instructions for how to do this will be different on each platform. You can also access these instructions using `?install.ffmpeg`

9 Compressing video

The saved video output of the animation often produces very large files. These may cause problems for your viewers if you upload them directly to the web. It is almost always a good idea to compress the video, as a dramatically smaller file can usually be created with little or no loss of quality. Although it may be possible to give `saveVideo()` various `other.opts` to control video compression, determining the right settings can be a trial and error process. Handbrake `http://handbrake.fr/` is an excellent and easy to use tool for doing video compression into the web-standard H.264 codec with appropriate presets.

10 Reference for the main command

Included here are more complete explanations of the main function. You can also refer to the man pages `?compute.animation` and `?render.animation`.

10.1 compute.animation()

The `compute.animation()` function computes a sequence of vertex layouts suitable for rendering a network animation. It steps through a `networkDynamic` object and applies layout algorithms at specified intervals, storing the calculated coordinates in the network for later use by the `render.animation` function. Generally the layouts are done in a sequence with each using the previously calculated positions as initial seed coordinates in order to smooth out the resulting movie.

The command takes several important arguments as named elements of the `slice.par` list. The parameters indicate it how "slice up' the network when computing layouts (`start`, `end`, `aggregate.dur` and `rule`), what type of layout algorithm to use (`animation.mode`), possible parameters to control the layouts (as a list named `layout.par`) and how much to try to separate nodes or disconnected components (`default.dist`). The computed coordinates are stored as dynamic vertex attributes named `animation.x.active` and `animation.y.active`. The slice `slice.par` list is stored as a standard network attribute. The network argument is modified in place, and returned invisibly

For each time slice, new coordinates are only computed for the active set of vertices, so the function usually behaves appropriately for networks with changing vertex sets.

10.2 render.animation()

This function is designed to step through a network object extracting slice networks according to the previously cached `slice.par` settings. It retrieves the `animation.x` and `animation.y` coordinates for each slice and passes them to `plot.network` to render the frame. If no `slice.par` network attribute is found to define the time range to render it will make one up using the smallest and largest non-Inf time values and unit-length non-overlapping time steps. If no stored coordinates are found it will call `compute.animation`. Additional `plot.network` control parameters (to set colors, line widths, etc) can be passed in via the `...` arguments. See `?plot.network` for the full list.

As mentioned earlier, a number of "tweening" animation frames are generated between each network slice with the positions of the vertices interpolated between the slices. This creates the illusion of smooth motion as the vertices change position, making it much easier to visually track changes in the network structure. As each slice (and tweening slice) is plotted, `ani.record` is called to store the image as a frame of the animation for later output.

Parameters to control the animation are read from a list passed in via the `render.par` argument.

- `tween.frames` is the number of interpolated frames to generate between each pre-calculated network layout. Default is 10. Increasing this will make the animation appear smoother and slower, but will make the file sizes much larger.
- `show.time` defaults to `TRUE`, in which case the x-axis of the plot will be labeled with the onset and terminus time for each slice as it is shown.
- `show.stats` does nothing with its default value of `NULL`. But if it is set to a string, it is assumed to be a formula and will passed to `summary.stergm` and the results used to display the network statistics for the current slice on the plot.
- `extraPlotCmds` provides a way present additional information (such as annotations) on the plot. The value of this argument will be passed to `eval()` after each frame has been plotted, so drawing commands can be added here.

There are also several lists of arguments that give default values that will be passed to the appropriate lower-level commands. The `plot.par` list is passed to the `par()` command and provides a way to configure some of the general plot details such as background color, margins, fonts, etc. Similarly, the `ani.options` list is passed to the `ani.options()` command to configure settings for the animation package such as `interval` to control the time between frames in playback.

10.3 saveVideo()

The `animation` package provides several neat tools for storing animations once they have been rendered.

- `ani.replay()` plays the animation back in the R plot window. (see `?ani.options` for more parameters)
- `saveVideo()` saves the animation as a movie file on disk (if the FFmpeg library is installed).
- `saveGIF()` creates an animated GIF (if ImageMagick installed)
- `saveLatex()` creates an animation embedded in a pdf (didn't work for me...)

Please see `?animation` and each function's help files for more details. With the exception of `ani.replay()` each of these requires the presence of some external library software which may need to be installed on your system as described in Dependencies (section 8).

11 Limitations

11.1 Size limits

Like most network algorithms, the layouts tend to scale quite badly. We generally have only had enough patience to generate moves for networks of less than 1000 vertices. There also seems to be quite a bit of overhead in the `animation` package, so the generation process seems to slow down considerably for longer duration networks or when slice or render parameters cause lots of slices to be generated.

References

Algorithmics Group, University of Konstanz (2009) *MDSJ: Java Library for Multidimensional Scaling (Version 0.2).* `http://www.inf.uni-konstanz.de/algo/software/mdsj/`.

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, Skye and McFarland, Daniel A. (2006) The Art and Science of Dynamic Network Visualization. *Journal of Social Structure. Volume 7, Number 2* `http://www.cmu.edu/joss/content/articles/volume7/deMollMcFarland/`

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/`.

Butts C, Leslie-Cook A, Krivitsky P and Bender-deMoll S (2013). *networkDynamic: Dynamic Extensions for Network Objects*. R package version 0.5, `http://statnet.org`.

de Leeuw J and Mair P (2009). "Multidimensional Scaling Using Majorization: SMACOF in R." *Journal of Statistical Software*, **31**(3), pp. 1–30. `http://www.jstatsoft.org/v31/i03/`

Bender-deMoll S (2013). *ndtv: Network Dynamic Temporal Visualizations*. R package version 0.2, `http://statnet.org`.

Handcock MS, Hunter DR, Butts CT, Goodreau SM, Morris M (2003b). statnet: Software tools for the Statistical Modeling of Network Data. Statnet Project, Seattle, WA. Version 3, `http://www.statnetproject.org`.

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/`.

Krivitsky P and Handcock M (2013). *tergm: Fit, Simulate and Diagnose Models for Network Evoluation based on Exponential-Family Random Graph Models*. The Statnet Project (`http://www.statnet.org`). R package version 3.1-0, `CRAN.R-project.org/package=tergm`.

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

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

Xie Y (2013). "animation: An R Package for Creating Animations and Demonstrating Statistical Methods." *Journal of Statistical Software*, **53**(1), pp. 1–27. `http://www.jstatsoft.org/v53/i01/`.