Workshop materials for learning ndtv: Network Dynamic Temporal Visualizations (Package version 0.5.1)

Skye Bender-deMoll

February 3, 2014

Contents

1 Introduction to workshop

1.1 What is ndtv?

The Network Dynamic Temporal Visualization (`ndtv`) package provides tools for visualizing changes in network structure and attributes over time.

- Uses network information encoded in `networkDynamic` (Butts et al. , 2014) objects as its input
- Outputs animated movies, 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.

, 2014) package in `statnet` (Handcock et al , 2003)) and simulations of disease spread across networks.

- Easy to do basic things, but lots of ability to customize.

1.2 Authors

- Developed by Skye Bender-deMoll (`skyebend@skyeome.net`) with members of the statnet `http://statnet.org` team.
- This work was supported by grant R01HD68395 from the National Institute of Health.

1.3 Workshop prerequisites

- Familiarity with the R statistical software. We are not going to cover basics of how to use and install R.
- Familiarity with general network and SNA concepts
- Experience with statnet packages and `network` data structures preferred but not necessary
- Functioning R installation and basic statnet packages already installed. (Instructions on installing R and statnet are located here `https://statnet.csde.washington.edu/trac/wiki/Sunbelt2013#Downloadingstatnet`.)
- A working internet connection (we will install some libraries and download datasets)

1.4 A Quick Demo

Lets get started with a realistic example. We can render a simple network animation in the R plot window (no need to follow along in this part)

First we load the package and its dependencies.

```
library(ndtv)
```

Now we need some dynamic network data to explore. The package includes an example network data set named `short.stergm.sim` which is the output of a toy STERGM model based on Padgett's Florentine Family Business Ties dataset simulated using the `tergm` package[1].

```
data(short.stergm.sim)
class(short.stergm.sim)
```

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

[1] Using the tergm package to simulate the model is illustrated in a later example.

Notice that the object has a class of both `networkDynamic` and `network`. All `networkDynamic` objects are still `network` objects, they just include additional special attributes to store time information.

```
print(short.stergm.sim)
```

```
NetworkDynamic properties:
  distinct change times: 25
  maximal time range: 0 to 25

Includes optional net.obs.period attribute:
 Network observation period info:
  Number of observation spells: 1
  Maximal range of observations: 0 to 25
  Temporal mode: discrete
  Time unit: step
  Suggested time increment: 1

 Network attributes:
  vertices = 16
  directed = FALSE
  hyper = FALSE
  loops = FALSE
  multiple = FALSE
  bipartite = FALSE
  net.obs.period: (not shown)
  total edges= 32
    missing edges= 0
    non-missing edges= 32

 Vertex attribute names:
    active priorates totalties vertex.names wealth

 Edge attribute names:
    active
```

The print command for `networkDynamic` objects includes some additional info about the time range of the network and then the normal output from `print.network`.

1.4.1 Render a network animation

Or if we wanted to get a quick visual summary of how the structure changes over time, we could render the network as an animation.

```
render.animation(short.stergm.sim)
```

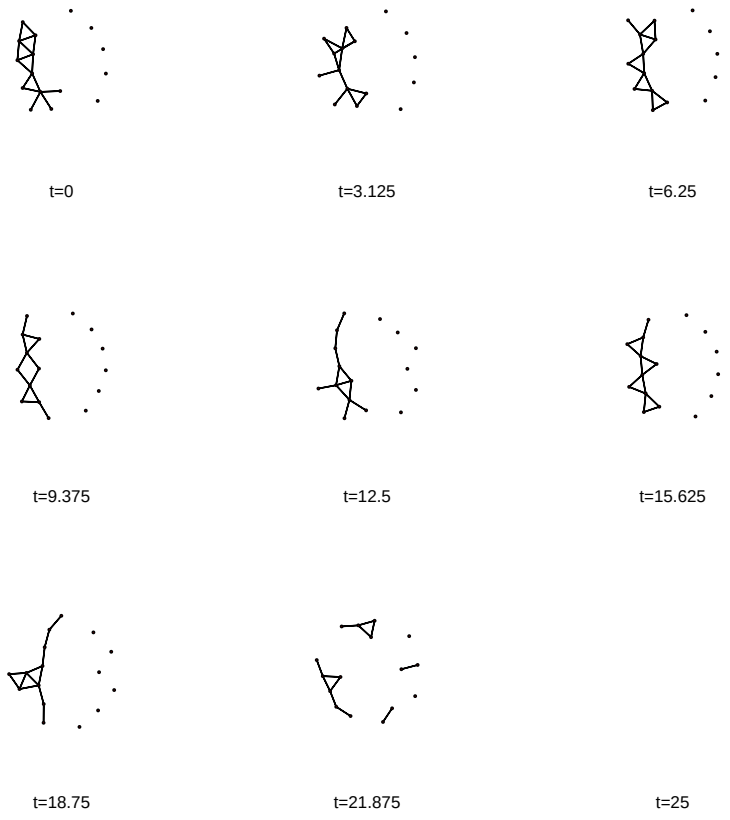
And then play it back in the R plot window

```
ani.replay()
```

1.4.2 Plot a static "filmstrip" sequence

An animation is not the only way to display a sequence of views of the network. We could also use the `filmstrip()` function that will create a "small multiple" plot using frames of the animation to construct a static visual summary of the network changes.

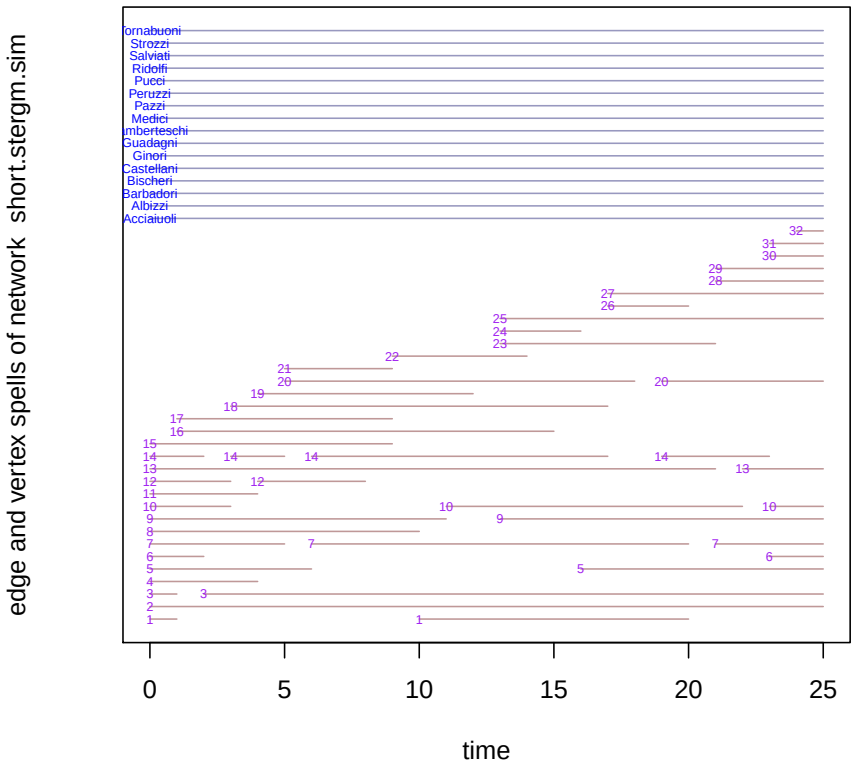
```
filmstrip(short.stergm.sim,displaylabels=FALSE)
```



1.4.3 Plot a timeline

We can view the dynamics as a timeline by plotting the active spells of edges and vertices.

```
timeline(short.stergm.sim)
```

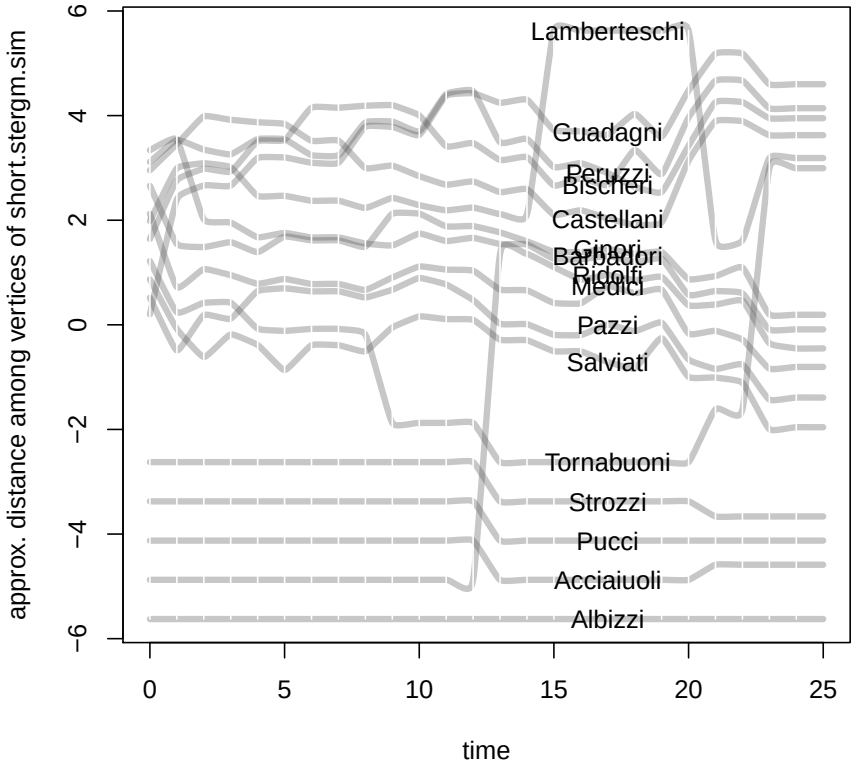


In this view, only the activity state of the network elements are shown–the structure and network connectivity is not visible. The vertices in this network are always active so they trace out unbroken horizontal lines in the upper portion of the plot, while the edge toggles are drawn in the lower portion.

1.4.4 Plot a proximity timeline

We are experimenting with a form of timeline or "phase plot". In this view vertices are positioned vertically by their geodesic distance proximity. This means that changes in network structure deflect the vertices' lines into new positions, attempting to keep closely-tied vertices as neighbors.

```
proximity.timeline(short.stergm.sim,default.dist=6,
          mode='sammon',labels.at=17,vertex.cex=4)
```



Notice how the bundles of vertex lines diverge after time 20, reflecting the split of the large component into two smaller components.

1.4.5 Print tabular data

Of course we can always display the edge (or vertex) spells directly in a tabular form using the various utilities in the `networkDynamic` package.

```
as.data.frame(short.stergm.sim)
```

```
   onset terminus tail head onset.censored terminus.censored duration edge.id
1      0        1    3    5          FALSE            FALSE        1       1
2     10       20    3    5          FALSE            FALSE       10       1
3      0       25    3    6          FALSE            FALSE       25       2
4      0        1    3    9          FALSE            FALSE        1       3
5      2       25    3    9          FALSE            FALSE       23       3
6      0        4    3   11          FALSE            FALSE        4       4
```

7	0	6	4	7	FALSE	FALSE	6	5
8	16	25	4	7	FALSE	FALSE	9	5
9	0	2	4	8	FALSE	FALSE	2	6
10	23	25	4	8	FALSE	FALSE	2	6
11	0	5	4	11	FALSE	FALSE	5	7
12	6	20	4	11	FALSE	FALSE	14	7
13	21	25	4	11	FALSE	FALSE	4	7
14	0	10	5	8	FALSE	FALSE	10	8
15	0	11	5	11	FALSE	FALSE	11	9
16	13	25	5	11	FALSE	FALSE	12	9
17	0	3	6	9	FALSE	FALSE	3	10
18	11	22	6	9	FALSE	FALSE	11	10
19	23	25	6	9	FALSE	FALSE	2	10
20	0	4	7	8	FALSE	FALSE	4	11
21	0	3	8	11	FALSE	FALSE	3	12
22	4	8	8	11	FALSE	FALSE	4	12
23	0	21	9	10	FALSE	FALSE	21	13
24	22	25	9	10	FALSE	FALSE	3	13
25	0	2	9	14	FALSE	FALSE	2	14
26	3	5	9	14	FALSE	FALSE	2	14
27	6	17	9	14	FALSE	FALSE	11	14
28	19	23	9	14	FALSE	FALSE	4	14
29	0	9	9	16	FALSE	FALSE	9	15
30	1	15	3	8	FALSE	FALSE	14	16
31	1	9	14	16	FALSE	FALSE	8	17
32	3	17	7	11	FALSE	FALSE	14	18
33	4	12	3	10	FALSE	FALSE	8	19
34	5	18	4	5	FALSE	FALSE	13	20
35	19	25	4	5	FALSE	FALSE	6	20
36	5	9	6	8	FALSE	FALSE	4	21
37	9	14	5	6	FALSE	FALSE	5	22
38	13	21	3	13	FALSE	FALSE	8	23
39	13	16	5	13	FALSE	FALSE	3	24
40	13	25	10	14	FALSE	FALSE	12	25
41	17	20	5	9	FALSE	FALSE	3	26
42	17	25	6	13	FALSE	FALSE	8	27
43	21	25	1	15	FALSE	FALSE	4	28
44	21	25	8	16	FALSE	FALSE	4	29
45	23	25	4	16	FALSE	FALSE	2	30
46	23	25	7	16	FALSE	FALSE	2	31
47	24	25	9	13	FALSE	FALSE	1	32

Question: What are some strengths and weakness of the various views?

Exercise: Load the saved version of short.stergm.sim. Are there any edges that are present for the entire time period from 0 until 25?

```
data(short.stergm.sim)
spls<-as.data.frame(short.stergm.sim)
spls[spls$duration==25,]
```

```
  onset terminus tail head onset.censored terminus.censored duration edge.id
3     0       25    3    6          FALSE             FALSE       25       2
```

2 The basics

2.1 Installing ndtv and its external dependencies

The `ndtv` (Bender-deMoll , 2014) package relies on many other packages to do much of the heavy lifting, especially `animation` (Yihui, Xie, et al. , 2013) and `networkDynamic`. It also requires external libraries (FFmpeg) to save movies out of the R environment, and Java to be able to some of the better layout algorithms.

2.1.1 Installing ndtv and its R package dependencies

R can automatically install the packages `ndtv` depends on when `ndtv` is installed. So open up your R console, and run the following command:

```
install.packages('ndtv',repos='http://cran.us.r-project.org',
                 dependencies=TRUE)
```

```
The downloaded source packages are in
        âĂŸ/tmp/Rtmpy9uaoN/downloaded_packagesâĂŹ
```

```
library(ndtv) # also loads animation and networkDynamic
```

2.1.2 Installing FFmpeg for saving animations

In order to save out animations as video files and use the better-quality layouts, we need to install some additional non-R software dependencies on the computer. 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`

```
?install.ffmpeg   # help page for installing ffmpeg
```

- **Windows instructions**
  - Download the recent 'static' build from `http://ffmpeg.zeranoe.com/builds/`
  - Downloads are compressed with 7zip, so you may need to first install a 7zip decompression program before you can unpack the installer.
  - Decompress the package and store contents on your computer (probably in Program Files)
  - Edit your system path variable to include the path to the directory containing ffmpeg.exe
- **Mac instructions**
  - Download most recent build from `http://www.evermeet.cx/ffmpeg/`
  - The binary files are compressed with 7zip so may need to install an unarchiving utility: `http://wakaba.c3.cx/s/apps/unarchiver.html`
  - Copy ffmpeg to `/usr/local/bin/ffmpeg`
- **Linux/Unix instructions**
  - FFmpeg is a standard package on many Linux systems. You can check if it is installed with a command like `dpkg -s ffmpeg`. If it is not installed, you should be able to install with your system's package manager. i.e. `sudo apt-get install ffmpeg` or search 'ffmpeg' in the Software Center on Ubuntu.

After you have installed FFmpeg on your system, you can verify that R knows where to find it by typing `Sys.which('ffmpeg')` in the R terminal. You many need to first restart R after the install.

2.1.3 Installing Java and MDSJ setup

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.

When java is installed correctly the following command should print out the version information:

```
system('java -version')
```

Due to CRAN's license restrictions, necessary components of the MDSJ layout (which we will use in a minute) are not distributed with `ndtv`. Instead, the first time the MDSJ layout is called after installing or updating the `ndtv` package, it is going to ask to download the library. Lets do that now on a pretend movie to get it out of the way:

```
network.layout.animate.MDSJ(network.initialize(1))
```

This will give a prompt like

```
The MDSJ Java library does not appear to be installed.
The ndtv package can use MDSJ to provide a fast
accurate layout algorithm. It can be downloaded from
http://www.inf.uni-konstanz.de/algo/software/mdsj/
Do you want to download and install the MDSJ Java library? (y/N):
```

Responding `y` to the prompt should install the library and print the following message:

```
MDSJ 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.

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

USE RESTRICTIONS: Creative Commons License 'by-nc-sa' 3.0.
```

And its good to go! (unless you were intending to use the layout for commercial work...)

2.1.4 Installing Graphviz

Graphviz is *not* required for `ndtv` to work. However it does provide several interesting layouts. Instructions for installing Graphviz can be shown with `?install.graphviz`.

2.2 Understanding how ndtv works

Now that we've had a preview of what the package can do, and everything is correctly configured, we can work through some examples in more detail to explain what is going on.

2.2.1 Constructing and rendering a dynamicNetwork

We are going to build a simple `dynamicNetwork` object "by hand" and then visualize its dynamics[2]. Please follow along by running these commands in your R terminal.

[2]The networkDynamic() command provides utilities for constructing dynamic networks from various data sources, see some the later examples

```
# create a static network with 10 vertices
wheel <- network.initialize(10)
# add some edges with activity spells
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)
```

Adding active edges to a network has the side effect of converting it to a `dynamicNetwork`. Lets verify it.

```
class(wheel)  # now it is also a networkDynamic object
```

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

```
print(wheel)
```

```
NetworkDynamic properties:
  distinct change times: 12
  maximal time range: 1 to 12

 Network attributes:
  vertices = 10
  directed = TRUE
  hyper = FALSE
  loops = FALSE
  multiple = FALSE
  bipartite = FALSE
  total edges= 18
    missing edges= 0
    non-missing edges= 18

 Vertex attribute names:
    vertex.names

 Edge attribute names:
    active
```

```
as.data.frame(wheel) # peek at edge dynamics as a data frame
```

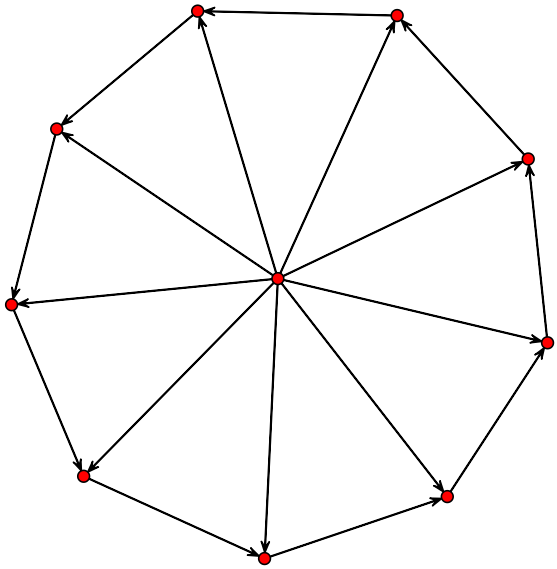
```
   onset terminus tail head onset.censored terminus.censored duration edge.id
1      1       11    1    2          FALSE             FALSE       10       1
2      2       11    2    3          FALSE             FALSE        9       2
3      3       11    3    4          FALSE             FALSE        8       3
4      4       11    4    5          FALSE             FALSE        7       4
5      5       11    5    6          FALSE             FALSE        6       5
```

6	6	11	6	7	FALSE	FALSE	5	6
7	7	11	7	8	FALSE	FALSE	4	7
8	8	11	8	9	FALSE	FALSE	3	8
9	9	11	9	1	FALSE	FALSE	2	9
10	10	12	10	1	FALSE	FALSE	2	10
11	10	12	10	2	FALSE	FALSE	2	11
12	10	12	10	3	FALSE	FALSE	2	12
13	10	12	10	4	FALSE	FALSE	2	13
14	10	12	10	5	FALSE	FALSE	2	14
15	10	12	10	6	FALSE	FALSE	2	15
16	10	12	10	7	FALSE	FALSE	2	16
17	10	12	10	8	FALSE	FALSE	2	17
18	10	12	10	9	FALSE	FALSE	2	18

When we look at the output data frame, we can see that it did what we asked. For example, edge id 1 connects the "tail" vertex id 1 to "head" vertex id 2 and has a duration of 10, extending from the "onset" of time 1 until the "terminus" of time 11.

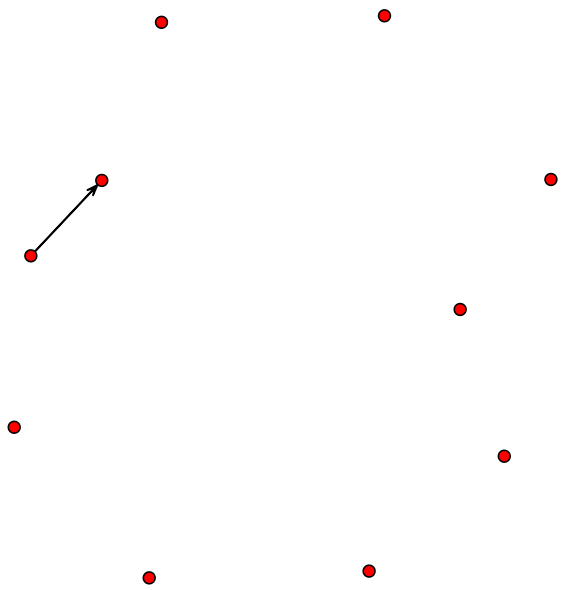
It is important to remember that `dynamicNetwork` objects are also static `network` objects. So all of the `network` functions will still work, they will just ignore the time dimension attached to edges and vertices. For example, if we just `plot` the network, we see all the edges that ever exist (and realize why the network is named "wheel").

```
plot(wheel)
```



If we want to just see the edges active at a specific time point, we could first extract a snapshot view of the network and then plot it.

```
plot(network.extract(wheel,at=1))
```



The `network.extract` function is one of the many tools provided by the `networkDynamic` package for storing and manipulating the time information attached to networks without having to work directly with the low-level data structures. The command `help(package='networkDynamic')` will give a listing of the help pages for all of the functions.

Exercise: Use the help function to determine the difference between the network.extract() and network.collapse() functions

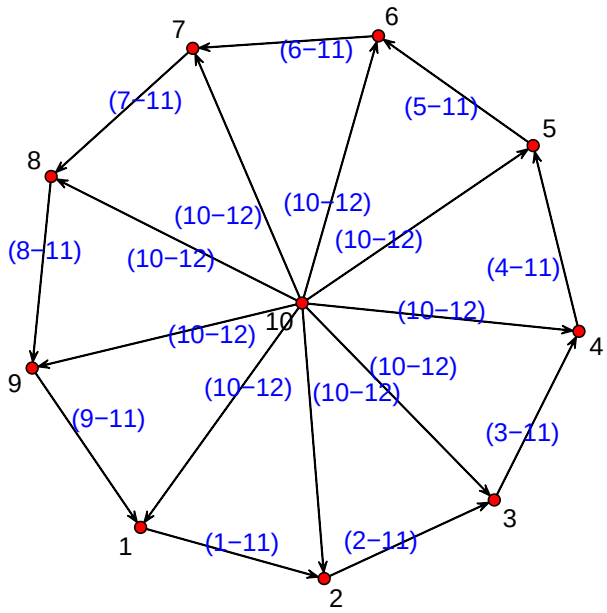
The activity for each edge is stored in an attribute named `activity` as a matrix of starting and ending times (which we refer to as "onset" and "terminus"). The help page `?activity.attribute` is a good place to learn more detail about how the `networkDynamic` package represents dynamics. For many tasks, we would use higher-level methods like `get.edgeIDs.active()` but, we can access timing information directly using `get.edge.activity`.

```
# print edge activity of edge.id 1
get.edge.activity(wheel)[[1]]
```

```
     [,1] [,2]
[1,]    1   11
```

Since the static plot, doesn't show us which edges are active when, lets annotate it by labeling edges with their onset and termination times so we can check that it constructed the network we told it to.

```
# make a list of times for each edge
elabels<-lapply(get.edge.activity(wheel),
                function(spl){
                  paste("(",spl[,1],"-",spl[,2],")",sep='')
                })
# peek at the static version
plot(wheel,displaylabels=TRUE,edge.label=elabels,
     edge.label.col='blue')
```



Question: Why is this edge labeling function not general enough for some networks? (Hint: do edges always have a single onset and terminus time?)

Now lets render the network as a dynamic movie and play it back so that we can visually understand the sequence of edge changes.

```
render.animation(wheel) # compute and render
```

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

[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 [3]. Then a set of "spoke" edges appear to draw a vertex into the center, and finally the rest of the wheel disappears.

[3] An example of the movie is located at `http://statnet.csde.washington.edu/movies/ndtv_vignette/wheel.mp4`

2.2.2 Understanding the default animation process

Simple right? Hopefully most of the complexity was hidden under the hood, but it is still useful to understand what is going on. At its most basic, rendering a movie consists of four key steps:

1. Determining appropriate parameters (time range, aggregation rule, etc)
2. Computing layout coordinates for each time slice
3. Rendering a series of plots for each time slice
4. Replaying the cached sequence of plots (or writing to a file on disk)

When we called `render.animation()` we asked the package to create an animation for `wheel` but we didn't include any arguments indicating what should be rendered or how, so it had to make some educated guesses or use default values. For example, it assumed that the entire time range of the network should be rendered and that we should use the Kamada-Kawai layout to position the vertices.

The process of positioning the vertices was managed by the `compute.animation()` function which stepped through the `wheel` network and called a layout function to compute vertex coordinates for each time step.

Next, `render.animation()` looped through the network and used `plot.network()` to render appropriate slice network for each time step. it calls the `animation` package function `ani.record()` to cache the frames of the animation. Finally, `ani.replay()` quickly redrew the sequence of cached images in the plot window as an animation.

2.2.3 Controlling the animation processing steps

For more precise control of the processes, we can call each of the steps in sequence and explicitly set the parameters we want for the rendering and layout algorithms. First we will define a `slice.par` list of parameters to specify the time range that we want to compute and render.

```
slice.par=list(start=1, end=12, interval=1, aggregate.dur=1,
               rule='latest')
```

Then we ask it to compute the coordinates for the animation. The `animation.mode` argument specifies which algorithm to use.

```
compute.animation(wheel,animation.mode='kamadakawai',slice.par=slice.par)
```

```
[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"
```

```
list.vertex.attributes(wheel)
```

```
[1] "animation.x.active" "animation.y.active" "na"
[4] "vertex.names"
```

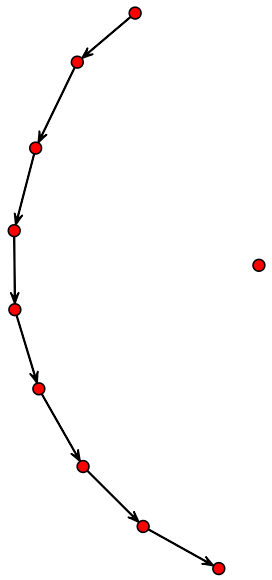
```
# peek at x coords at time 4
get.vertex.attribute.active(wheel,'animation.x',at=4)
```

```
[1]  1.9312637  1.1013854  0.2362666 -0.6577887 -1.5497865 -2.0772677
[7]  1.1123233 -1.2756308  2.0772677 -0.1448341
```

We can see that in addition to the standard vertex attributes of `na` and `vertex.names`, the network now has two dynamic "TEA" attributes for each vertex to describe its position over time. The `slice.par` argument is also cached as a network attribute so that later on `render.animation()` will know what range to render.

Since the coordinates are stored in the network, we can collapse the dynamics at any time point, extract the coordinates, and plot it:

```
wheelAt8<-network.collapse(wheel,at=8)
coordsAt8<-cbind(wheelAt8%v%'animation.x',wheelAt8%v%'animation.y')
plot(wheelAt8,coord=coordsAt8)
```



This is essentially what `render.animation()` does internally. The standard network plotting arguments are accepted by `render.animation` (via `...`) and will be passed to `plot.network()`:

```
render.animation(wheel,vertex.col='blue',edge.col='gray',
                 main='A network animation')
```

```
[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"
```

`render.animation()` also plots a number of in-between frames for each slice to smoothly transition the vertex positions between successive points. We can adjust how many "tweening" interpolation frames will be rendered which indirectly impacts the perceived speed of the movie (more tweening means a slower and smoother movie). For no animation smoothing at all, set `tween.frames=1`.

```
render.animation(wheel,render.par=list(tween.frames=1),
                 vertex.col='blue',edge.col='gray')
```

```
[1] "rendering 1 frames for slice 0"
[1] "rendering 1 frames for slice 1"
[1] "rendering 1 frames for slice 2"
[1] "rendering 1 frames for slice 3"
[1] "rendering 1 frames for slice 4"
[1] "rendering 1 frames for slice 5"
[1] "rendering 1 frames for slice 6"
[1] "rendering 1 frames for slice 7"
[1] "rendering 1 frames for slice 8"
[1] "rendering 1 frames for slice 9"
[1] "rendering 1 frames for slice 10"
[1] "rendering 1 frames for slice 11"
```

```
ani.replay()
```

Or bump it up to 30 for a slow-motion replay:

```
render.animation(wheel,render.par=list(tween.frames=30),
                 vertex.col='blue',edge.col='gray')
```

```
[1] "rendering 30 frames for slice 0"
[1] "rendering 30 frames for slice 1"
[1] "rendering 30 frames for slice 2"
[1] "rendering 30 frames for slice 3"
[1] "rendering 30 frames for slice 4"
[1] "rendering 30 frames for slice 5"
[1] "rendering 30 frames for slice 6"
[1] "rendering 30 frames for slice 7"
[1] "rendering 30 frames for slice 8"
[1] "rendering 30 frames for slice 9"
[1] "rendering 30 frames for slice 10"
[1] "rendering 30 frames for slice 11"
```

```
ani.replay()
```

If you are like me, you probably forget what the various parameters are and what they do. You can use `?compute.animation` or `?render.animation` to display the appropriate help files. and `?plot.network` to show the list of plotting control arguments.

Question: Why is all this necessary? Why not just call plot.network over and over at each time point?

2.3 Animation Output Formats

We have been only playing back animations in the R plot window. But what if you want to share your animations with collaborators or post them on the web? Assuming that the external dependencies are correctly installed, we can save out some movies in multiple useful formats supported by the `animation` package:

- `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's `convert` is installed)
- `saveLatex()` creates an animation embedded in a PDF document

Please see `?animation` and each function's help files for more details and parameters. We will quickly demonstrate some useful options below.

2.3.1 Video files

Since we just rendered the "wheel" example movie, it is already cached so we can capture the output of `ani.replay` into a movie file. Try out the various output options below.

```
saveVideo(ani.replay(),video.name="wheel_movie.mp4")
```

You will probably see a lot output on the console from ffmpeg reporting its status, and then it should open the movie in an appropriate viewer on your machine.

Sometimes we may want to change the pixel dimensions of the movie output to make the plot (and the file size) much larger.

```
saveVideo(ani.replay(),video.name="wheel_movie.mp4",
          ani.width=800,ani.height=800)
```

We can increase the video's image quality (and file size) by telling ffmpeg to use a higher bit-rate (less compression)[4].

[4]This bit-rate setting seems to mostly impacts jpeg output more than the PNG default.

```
saveVideo(ani.replay(),video.name="wheel_movie.mp4",
          other.opts="-b 5000k")
```

Because the `ani.record()` and `ani.replay()` functions cache each plot image in memory, they are not very speedy and the rendering process will tend to slow to a crawl down as memory fills up when rendering large networks or long movies. We can avoid this by saving the output of `render.animation` directly to disk by wrapping it inside the `saveVideo()` call and setting `render.cache='none'`.

```
saveVideo(render.animation(wheel,vertex.col='blue',
          edge.col='gray',render.cache='none'),
          video.name="wheel_movie.mp4")
```

2.3.2 Animated GIF files

We can also export it as an animated GIF image. GIF animations will be very large files, but very portable for sharing on the web (assuming you happen to have ImageMagick installed...).

```
saveGIF(render.animation(wheel,vertex.col='blue',
          edge.col='gray',render.cache='none'),
         movie.name="wheel_movie.gif")
```

The `animation` package supports including animations insdie PDF documents if you have the appropriate Latex utilities installed. However, the animations will only play inside Adobe Acrobat PDF viewers so it is probably less portable than using GIF or video renders.

2.3.3 Animated PDF/Latex files

```
saveLatex(render.animation(wheel,vertex.col='blue',
          edge.col='gray',render.cache='none'))
```

Exercise: Using the list of options from the help page ?ani.options, locate the option to control the time delay interval of the animation, and use it to render a video where each frame stays on screen for 2 seconds.

```
saveVideo(render.animation(wheel,vertex.col='blue',
          edge.col='gray',render.cache='none',
          render.par=list(tween.frames=1),
          ani.options=list(interval=2)),
         video.name="wheel_movie.mp4")
```

3 Using ndtv Effectively

3.1 Animated Layout Algorithms

First some background about graph layouts. Producing "good" (for an admittedly ambiguous definition of good) layouts of dynamic networks is generally a computationally difficult problem.

Common goals:

- Layouts should remain as visually stable as possible over time.
- 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.

So far, in ndtv we are using variations of Multidimensional Scaling (MDS) layouts. MDS algorithms use various numerical optimization techniques to find a configuration of points (the vertices) in a low dimensional space (the screen) where the distances between the points are as close as possible to the desired distances (the edges). This is somewhat analogous to the process of squashing a 3D world globe onto a 2D map: there are many useful ways of doing the projection, but each introduces some type of distortion. For networks, we are attempting to define a high-dimensional "social space" to project down to 2D.

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. Using the previous coordinates allows us to "chain" the layouts together. This means that each visualization step can often avoid some computational work by using a previous solution as its starting point, and it is likely to find a solution that is spatially similar to the previous step.

3.1.1 Why we avoid Fruchterman-Reingold

The Fruchterman-Reingold algorithm has been one of the most popular layout algorithms for graph layouts (it is the default for `plot.network`). For larger networks it can be tuned to run much more quickly than most MDS algorithms. Unfortunately, its default optimization technique introduces a lot of randomness, so the "memory" of previous positions is usually erased each time the layout is run[5], producing very unstable layouts when used for animations..

[5] Various authors have had useful animation results by modifying FR to explicitly include references to vertices' positions in previous time points. Hopefully we will be able to include such algorithms in future releases of ndtv.

3.1.2 Kamada-Kawai adaptation

The Kamada-Kawai network layout algorithm is often described as a "force-directed" or "spring embedder" simulation, but it is mathematically equivalent to some forms of MDS[6]. 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 using `layout.distance` (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. Isolates often move around for no clear reason. 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.

```
compute.animation(short.stergm.sim,animation.mode='kamadakawai')
saveVideo(render.animation(short.stergm.sim,render.cache='none',
                           main='Kamada-Kawai layout'),
          video.name='kamadakawai_layout.mp4')
```

3.1.3 MDSJ (Multidimensional Scaling for Java)

MDSJ is a very efficient implementation of "SMACOF" stress-optimization Multidimensional Scaling. The `network.layout.animate.MDSJ` layout 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. It also produces very smooth layouts with less of the wobbling and flipping which can sometimes occur with Kamada-Kawai. Like Kamada-Kawai, it computes a symmetric geodesic distance matrix from the input network using `layout.distance` (replacing infinite values with `default.dist`), and seeds the initial coordinates for each slice with the results of the previous slice.

As noted earlier, 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. More information about the MDSJ library and its licensing can be found at `http://www.inf.uni-konstanz.de/algo/software/mdsj/`.

```
compute.animation(short.stergm.sim,animation.mode='MDSJ')
saveVideo(render.animation(short.stergm.sim,render.cache='none',
                           main='MDSJ layout'),
          video.name='MDSJ_layout.mp4')
```

[6]Kamada-Kawai uses Newton-Raphson optimization instead SMACOF stress-minimization

3.1.4 Graphviz layouts

The Graphviz (John Ellson et al , 2001) external layout library includes a number of excellent algorithms for graph layout, including `neato`, an stress-optimization variant, and `dot` a hierarchical layout (for trees and DAG networks). As Graphviz is not an R package, you must first install the the software on your computer following the instructions at `?install.graphviz` before you can use the layouts. The layout uses the `export.dot` function to create temp file which it then passes to Graphviz.

```
compute.animation(short.stergm.sim,animation.mode='Graphviz')
saveVideo(render.animation(short.stergm.sim,render.cache='none',
                           main='Graphviz-neato layout'),
          video.name='gv_neato_layout.mp4')
compute.animation(short.stergm.sim,animation.mode='Graphviz',
                  layout.par=list(gv.engine='dot'))
```

3.1.5 Existing coordinates or customized layouts

The `network.layout.animate.useAttribute` layout is useful if you already know exactly where each vertex should be drawn at each time step (based on external data such as latitude and longitude), and you just want `ndtv` to render out the network. It just needs to know the names of the dynamic TEA attribute holding the x coordinate and the y coordinate for each time step. If no suitable attributes are found, it will just produce an error.

```
compute.animation(short.stergm.sim,animation.mode='useAttribute')
```

It is also possible to write your own layout function and easily plug it in by defining a function with a name like `network.layout.animate.MyLayout`. See the `ndtv` package vignette for a circular layout example `browseVignette(package='ndtv')`.

Excercise: Play the videos for each layout type in adjacent video player windows to compare the algorithms. Or watch this composite version `http://statnet.org/movies/layout_compare.mp4`. What differences are apparent

If playing multiple videos is difficult, we've also posted a single video that shows all three side-by-side on a longer simulation: `http://` More detailed information about each of the layouts and their parameters can be found on the help pages, for example `?network.layout.animate.kamadakawai`.

3.2 Slicing and aggregating 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 to be fed a static set of relationships which can be used to compute a set of

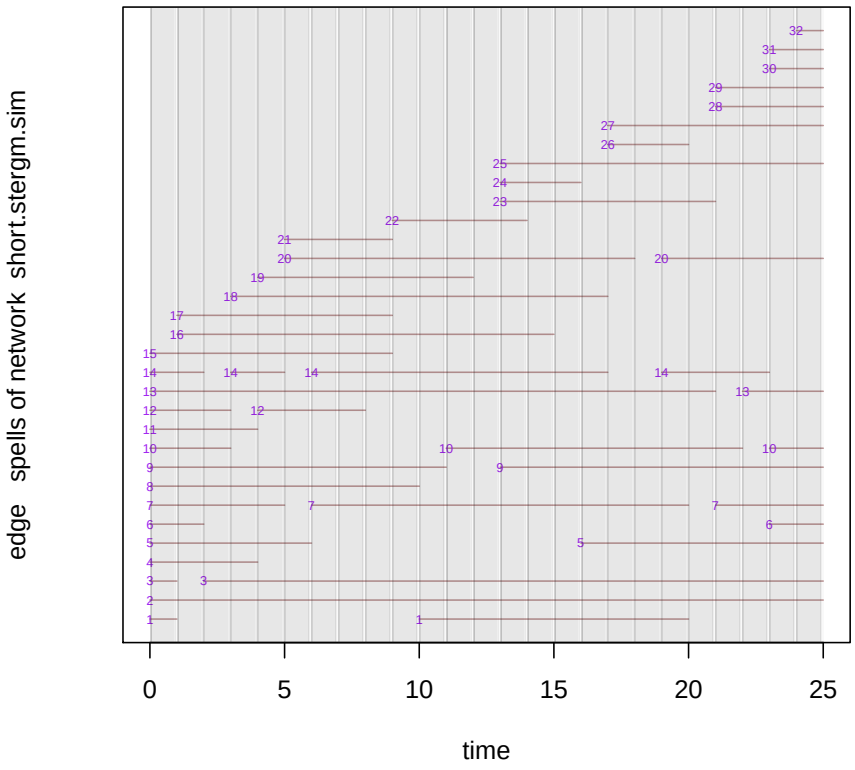
distances in a Euclidean space suitable for plotting. A common way to apply static metrics to a time-varying object is to sample it, taking a sequence static observations at a series of time points and using these to describe the changes over time. In the case of networks, we often call this this sampling process "extracting" or "slicing"–cutting through the dynamics to obtain a static snapshot. The processes of determining a set of slicing parameters appropriate for the network phenomena and data-set of interest requires careful thought and experimentation. As mentioned before, it is somewhat similar to the questions of selecting appropriate bin sizes for histograms.

3.2.1 Slicing panel data

In both the `wheel` and `short.stergm.sim` examples, we've been implicitly slicing up time in discrete way, extracting a static network at each unit time step.

We can plot the slice bins against the timeline of edges to visualize the "observations" of the network that the rendering process is using. When the horizontal line corresponding to an edge spell crosses the vertical gray bar corresponding to a bin, the edge would be included in that network. If this was a social network survey, each slice would correspond to one data collection panel of the network.

```
timeline(short.stergm.sim,slice.par=list(start=0,end=25,interval=1,
                     aggregate.dur=1,rule='latest'),
                     plot.vertex.spells=FALSE)
```



Looking at the first slice, which extends from time zero *until*[7] time 1, it appears that it should have 15 edges in it. Let's check:

```
# extract network and count edges
network.edgecount(network.extract(short.stergm.sim,onset=0,terminus=1))
```

```
[1] 15
```

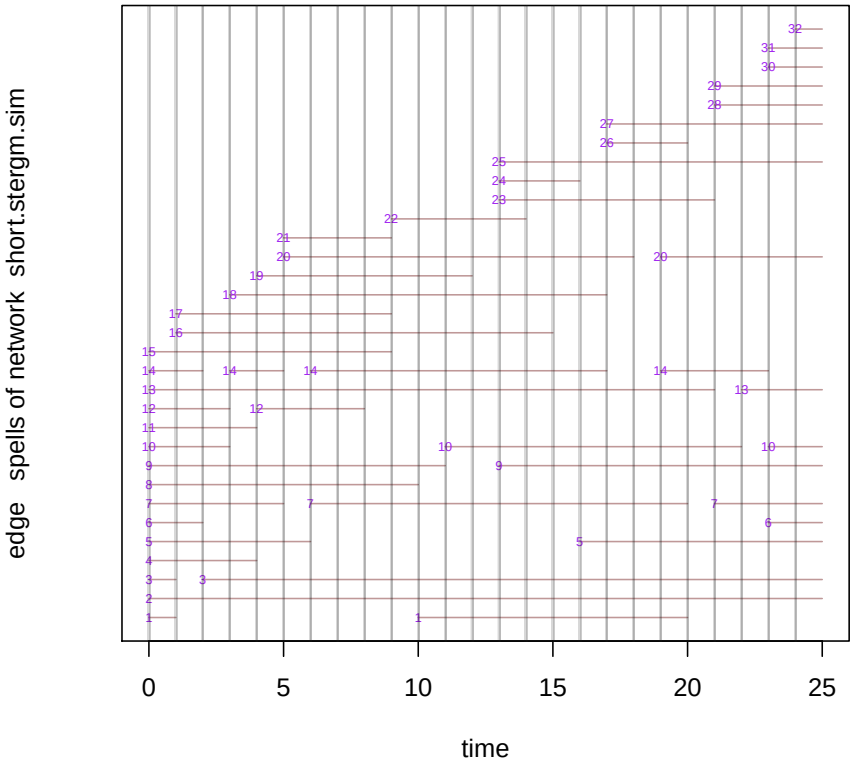
```
# or just count active edges directly
network.edgecount.active(short.stergm.sim,onset=0,terminus=1)
```

```
[1] 15
```

Because this is a discrete time network, the edge spells always extend the full duration of the slice, so it wouldn't actually matter if we used a shorter `aggregate.dur`. Each slice will still intersect with the same set of edges.

[7] notice the right-open interval definition!

```
timeline(short.stergm.sim,slice.par=list(start=0,end=25,interval=1,
                        aggregate.dur=0,rule='latest'),
                        plot.vertex.spells=FALSE)
```



Some data-sets that are collected as panels the time units many not be integers, so the slicing parameters might need to be adjusted to the natural time units. And, as we will see later in the windsurfers example, there are situations where using longer aggregation durations can be helpful even for panel data.

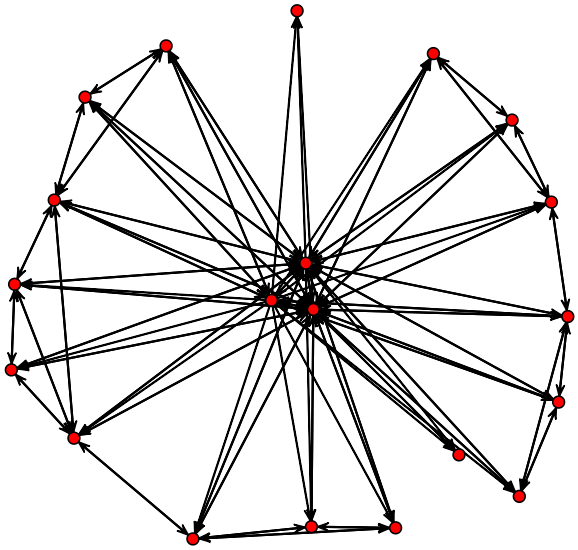
3.2.2 Slicing streaming data

Slicing up a dynamic network created from discrete panels may appear to 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 aggregation options. In many situations we have even found (Bender-deMoll and McFarland , 2006) it useful to let slices mostly overlap – increment 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.

```
data(McFarland_cls33_10_16_96)
# plot the time-aggregated network
plot(cls33_10_16_96)
```



First, we can animate at the fine time scale, viewing the first half-hour of class using instantaneous slices.

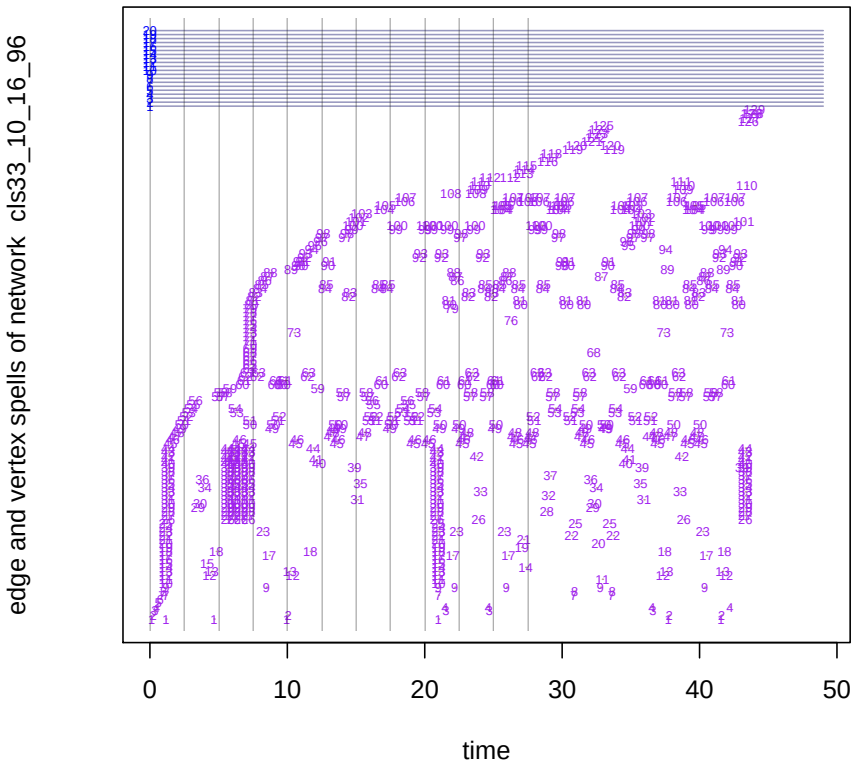
```
slice.par<-list(start=0,end=30,interval=2.5,
                aggregate.dur=0,rule="latest")
```

```
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()
```

Notice that although the time-aggregated plot shows a fully-connected structure, in the animation most of the vertices are isolates, occasionally linked into brief pairs or stars by speech acts[8]. Once again we can get an idea of what is going on by slicing up the network by using the `timeline()` function to plot the `slice.par` parameters against the vertex and edge spells. Although the vertices have spells spanning the entire time period, in this data set the edges are recorded as instantaneous "events" with no durations. The very thin slices (gray vertical lines) (`aggregate.dur=0`) are not intersecting many edge events (purple numbers) at once so the momentary degrees are mostly low.

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

[8] `http://statnet.csde.washington.edu/movies/ndtv_vignette/cls33_10_16_96v1.mp4`



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[9].

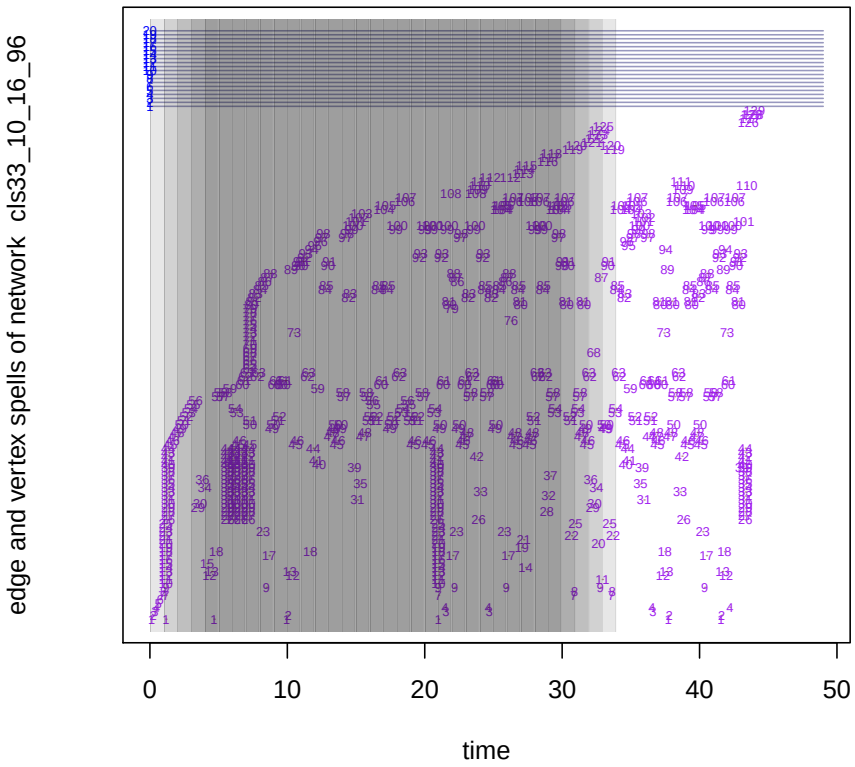
```
slice.par<-list(start=0,end=30,interval=2.5,
                aggregate.dur=2.5,rule="latest")
compute.animation(cls33_10_16_96,
                slice.par=slice.par,animation.mode='MDSJ')
render.animation(cls33_10_16_96,
                 displaylabels=FALSE,vertex.cex=1.5)
ani.replay()
```

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 same edge may appear in sequential slices and the changes will be less dramatic between successive views[10].

[9] `http://statnet.csde.washington.edu/movies/ndtv_vignette/cls33_10_16_96v2.mp4`
[10] `http://statnet.csde.washington.edu/movies/ndtv_vignette/cls33_10_16_96v3.mp4`

Question: How would measurements of a network's structual properties (degree, etc.) at each time point differ in each of the aggregation scenarios?

```
slice.par<-list(start=0,end=30,interval=1,
                aggregate.dur=5,rule="latest")
timeline(cls33_10_16_96,slice.par=slice.par)
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 the edge between a pair of vertices has more than one active period. How should this condition be handled? If the edge has attributes, which ones should be shown?

Ideally we might want to aggregate the edges in some way, perhaps adding the weights together. Currently edge attributes are not aggregated and the `rule`

element of the `slice.par` argument controls which attribute should be returned for an edge when multiple elements are encountered. Generally `rule='latest'` gives reasonable results, returning the most recent value found within the query spell.

Exercise: Define a slice.par and render an animation of the first 15 minutes of classroom interactions using 5 minute non-overlapping slices

3.2.3 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 good example of this. In this data-set there are different people present on the beach on successive days, and there is even a day of missing data.

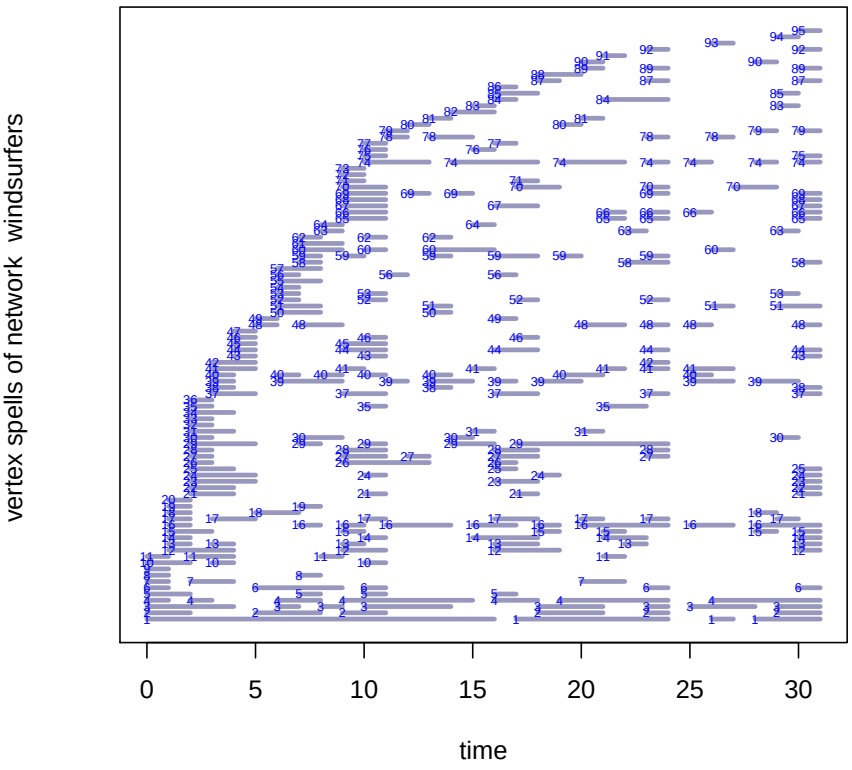
```
data(windsurfers)
slice.par<-list(start=1,end=31,interval=1,
                aggregate.dur=1,rule="latest")
windsurfers<-compute.animation(windsurfers,slice.par=slice.par,
                               default.dist=3,
                               animation.mode='MDSJ',
                       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

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 the lower `default.dist` above can help with this.

In this example[11] the turnover of people on the beach is so great that structure appears to change chaotically, and it is quite hard to see what is going on. Vertices enter and exit frequently and are often not available for observation on successive days. For example, look at vertex 74 in contrast to vertex 1 on the timeline plot.

```
timeline(windsurfers,plot.edge.spells=FALSE,lwd=3)
```

[11] `http://statnet.csde.washington.edu/movies/ndtv_vignette/windsurfers_v1.mp4`



Notice 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 changes against the aggregate weekly structure.

```
slice.par<-list(start=0,end=24,interval=1,
                aggregate.dur=7,rule="latest")
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[12] is larger and more dense (which is to be expected) and also far more stable. There is still some turnover due to people who don't make it to the beach every week but it is now possible to see some of the sub-groups and the the various bridging individuals.

Question: How would a researcher determine the "correct" aggregation period or data collection interval for a dynamic process? When the "same" network is sampled repeatedly, wow can we distinguish sampling noise from network dynamics?

3.3 Animating network attributes

Vertices and edges are not the only things that change over time, how do we display dynamic attributes (such as infection status) and changes to structural properties of the network?

3.3.1 Controlling plot properties using dynamic attributes (TEAs)

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. The help file listing and explaining dynamic TEA attributes can be displayed with `?dynamic.attributes?`.

We can attach an attribute named "width" to the edges which will take values of 1, 5, and 10 for various ranges of times.

```
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=7,terminus=Inf)
```

And check what we created.

```
list.edge.attributes(wheel)
```

```
[1] "active"       "na"           "width.active"
```

```
get.edge.attribute.active(wheel,'width', at=2)
```

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

```
get.edge.attribute.active(wheel,'width', onset=5,terminus=6)
```

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

[12] http://statnet.csde.washington.edu/movies/ndtv_vignette/windsurfers_v2.mp4

```
get.edge.attribute.active(wheel,'width', at=300)
```

```
[1] 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

There are many features and complexities that come up when working with dynamic attributes. For example, with the commands above, we've defined values for edges even when those edges are inactive. Compare:

```
get.edge.attribute.active(wheel,'width', at=2)
```

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

```
get.edge.attribute.active(wheel,'width', at=2,require.active=TRUE)
```

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

When using an attribute to control a plot parameter we must make sure the attributes are always defined for every time period that the network will be plotted or else an error will occur. So it is often good practice first set a default value from `-Inf` to `Inf` before overwritting with later commands 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 parameters 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.

Exercise: Using the wheel network, create a dynamic vertex attributed named "group". Define the TEA so that initially most of the vertices will be in group "A", but over time more and more will be in group "B"

3.3.2 Controlling plot properties with special functions

Sometimes it is awkward or inefficient to pre-generate dynamic attribute values. Why create and attach another attribute for color if it is just a simple transformation of an existing attribute or measure? The `render.animation` function has the ability to accept the `plot.network` arguments as R functions with specially-named arguments. These functions will be evaluated "on the fly" at each time point as the network is rendered and can operate on the attributes and properties of the collapsed network.

For example, if we wanted to use our previously created "width'" attribute to control the color of edges along with their width, we could define a function to extract the value of the "width" edge attribute from the momentary slice network and map it to the red component of an RGB color. We can render the network, setting `edge.col` equal to this function.

```
render.animation(wheel,edge.lwd=3,
    edge.col=function(slice){rgb(slice%e%'width'/10,0,0)},
    verbose=FALSE)
ani.replay()
```

Notice the `slice` function argument and the use of `slice` instead of the original name of the network in the body of the function. The arguments of plot control functions must draw from a specific set of named arguments which will be substituted in and evaluated at each time point before plotting. The set of valid argument names that can be used in special functions is:

- `net` is the original (un-collapsed) network
- `slice` is the static network slice to be rendered, collapsed with the appropriate onset and terminus
- `s` is the slice number in the sequence to be rendered
- `onset` is the onset (start time) of the slice to be rendered
- `terminus` is the terminus (end time) of the slice to be rendered

We can also define functions based on calculated network measures such as betweenness rather than network attributes:

```
require(sna)
wheel%n%'slice.par'<-list(start=1,end=10,interval=1,
                          aggregate.dur=1,rule='latest')
render.animation(wheel,
      vertex.cex=function(slice){(betweenness(slice)+1)/5},
      verbose=FALSE)
ani.replay()
```

Notice that in this example we had to modify the start time using the `slice.par` setting to avoid time 0 because the `betweenness` function in the `sna` package will give an error for a network with no edges.

Exercise: write a functional plot argument that scales vertex size in proportion to momentary degree

The main plot commands accept functions as well, so it is possible to do fun things like implement a crude zoom effect by setting `xlim` and `ylim` parameters to be dependent on the time.

```
render.animation(wheel,
      xlim=function(onset){c(-5/(onset*0.5),5/(onset*0.5))},
      ylim=function(onset){c(-5/(onset*0.5),5/(onset*0.5))},
      verbose=FALSE)
ani.replay()
```

3.4 Using and aggregating edge weights

In the previous examples we have treated the networks we are visualizing as un-weighted. But for some applications it may be useful to incorporate edge weight information in a layout. For example, when we are rendering an aggregate network, should an edge that exists for a single moment be treated the same way as an edge that exists the entire time? Even if the original network data does not contain weights, we may wish to perform operations using edge duration information.

By default, the `collapse.network()` function returns an ordinary static network. But setting the argument `rm.time.info=FALSE` will cause it to add some attributes with crude summary information about the activity of the edges and vertices over the time period of aggregation.

```
simAgg <-network.collapse(short.stergm.sim,rm.time.info=FALSE,
                          rule='latest')
list.edge.attributes(simAgg)
```

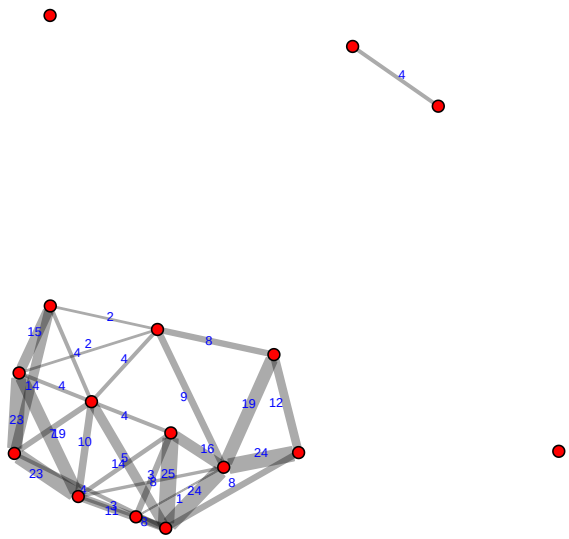
```
[1] "activity.count"    "activity.duration" "na"
```

Notice the `activity.count` and `activity.duration` that are now attached to the edges.

By default the layout algorithms will assume that all edges should have the same "ideal" length, any weights or values attached to the edge will be ignored. But if we do have edge values, either from raw data or by aggregating the edges over time, we might want to have the layout attempt to map that information the desired lengths of the edges. The `weight.attr` argument makes it possible to pass in the name of a numeric edge attribute to be used in constructing the `layout.dist` matrix of desired distances between vertices.

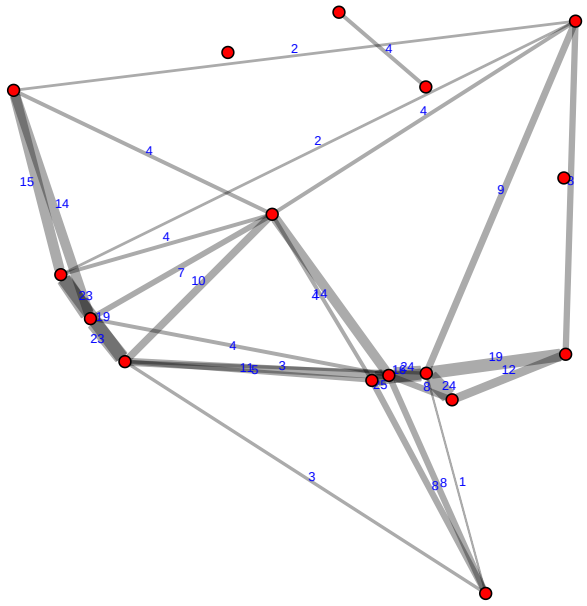
Lets plot the aggregate network with edge weights ignored. But instead of using the normal `plot.network` command alone, we are going to extract the coordinates created by `compute.animation` and feed them into the plot (this is just for constructing the examples, no need to do this for real movies). We will also use the `activity.duration` attribute to control the drawing width and labels for the edges.

```
# set slice par to single slice for entire range
slice.par<-list(start=0,end=25,interval=25,
                aggregate.dur=25,rule='latest')
compute.animation(short.stergm.sim,animation.mode='MDSJ',
                  slice.par=slice.par)
# extract the coords so we can do a static plot
coords<-cbind(get.vertex.attribute.active(short.stergm.sim,
                                          'animation.x',at=0),
              get.vertex.attribute.active(short.stergm.sim,
                                          'animation.y',at=0))
plot(simAgg,coord=coords,
     edge.lwd='activity.duration', edge.col='#00000055',
     edge.label='activity.duration',edge.label.col='blue',
     edge.label.cex=0.5,main='Edge weights ignored')
```



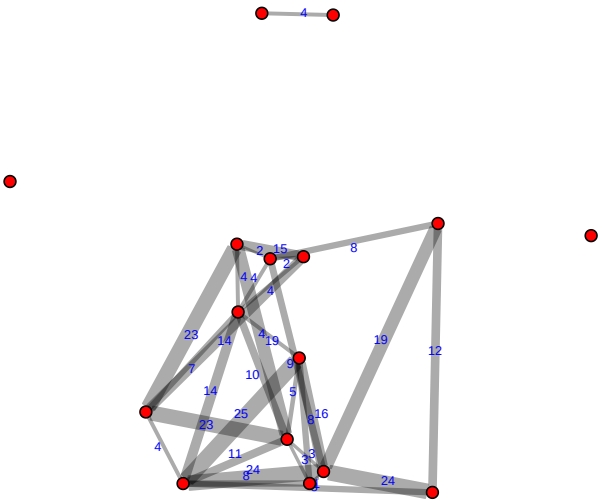
For comparison, we will compute the layout so that the higher-valued edges draw vertices closer together (the default `weight.dist=FALSE` considers edge weights to be similarities)

```
compute.animation(short.stergm.sim,weight.attr='activity.duration',
          animation.mode='MDSJ',seed.coords=coords,default.dist=20)
coords<-cbind(get.vertex.attribute.active(short.stergm.sim,
                                          'animation.x',at=0),
              get.vertex.attribute.active(short.stergm.sim,
                                          'animation.y',at=0))
plot(simAgg,coord=coords,
     edge.lwd='activity.duration', edge.col='#00000055',
     edge.label='activity.duration',edge.label.col='blue',
     edge.label.cex=0.5,main='Edge weights as similarity')
```



Now compute layout with `weight.dist=FALSE` so weights are treated as distance and higher-valued edges push vertices further apart.

```
compute.animation(short.stergm.sim,weight.attr='activity.duration',
                  weight.dist=TRUE, animation.mode='MDSJ',
                  seed.coords=coords,default.dist=20)
coords<-cbind(
  get.vertex.attribute.active(short.stergm.sim, 'animation.x',at=0),
  get.vertex.attribute.active(short.stergm.sim, 'animation.y',at=0)
)
plot(simAgg,coord=coords,
     edge.lwd='activity.duration', edge.col='#00000055',
     edge.label='activity.duration',edge.label.col='blue',
     edge.label.cex=0.5,main='Edge weights as distance')
```



Although we can clearly see that the layout is trying to do what we ask it, the result is a graph that is visually hard to read. Part of the problem is that we are asking the layout to achieve something difficult: the edge weights (and corresponding desired lengths) differ by more than an order of magnitude.

```
range(simAgg%e%'activity.duration')
```

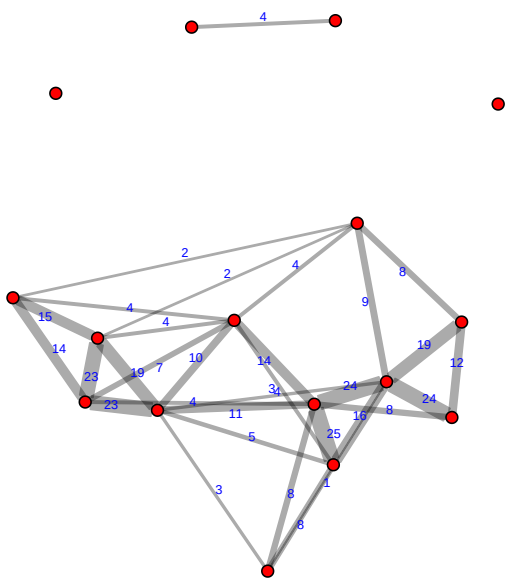
```
[1]  1 25
```

```
range(log(simAgg%e%'activity.duration'+1))
```

```
[1] 0.6931472 3.2580965
```

This suggests a solution of attaching a new edge attribute containing the logged duration, and using that as the `weight.attr`

```
set.edge.attribute(short.stergm.sim,'logDuration',
                   log(simAgg%e%'activity.duration'+1))
compute.animation(short.stergm.sim,
              weight.attr='logDuration',
                  animation.mode='MDSJ',seed.coords=coords)
coords<-cbind(
  get.vertex.attribute.active(short.stergm.sim, 'animation.x',at=0),
  get.vertex.attribute.active(short.stergm.sim, 'animation.y',at=0))
plot(simAgg,coord=coords,
     edge.lwd='activity.duration', edge.col='#00000055',
     edge.label='activity.duration',edge.label.col='blue',
     edge.label.cex=0.5,main='Edge weights as log similarity')
```



This gives us something better looking–big edges are a bit shorter, but not so much that they squinch up the layout.

Question: What types of data have edge weights that would best be thought of as distances? Similarities?

3.5 Adjusting spacing of isolates and components

We've briefly mentioned the `default.dist` parameter earlier. Most of the layouts use the `layout.dist` function to compute a matrix of geodesic path distances between vertices. Because there isn't a way to compute a path distance between vertices that have no connections (or between disconnected components) the empty cells of the matrix are filled in with a default value provided by `default.dist`. This value can be tweaked to increase or decrease the spacing between isolates and disconnected components. However, because this works by essentially introducing invisible edges between all of the elements, it can also introduce some distortion in the layout. The default value for `default.dist` is `sqrt(network.size(net))`, see `?layout.dist` for more information.

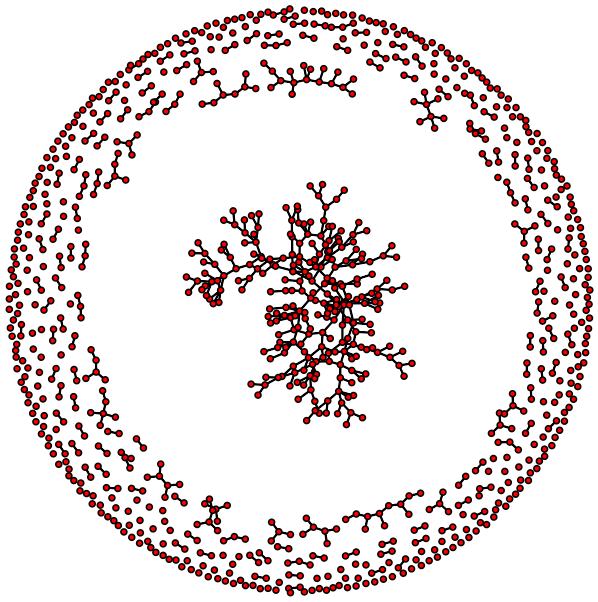
To demonstrate this we will work with a single static slice of the network and call the animation layout directly so we can avoid rendering out the entire movie for each test.

```
data(msm.sim)
msmAt50<-network.extract(msm.sim,at=50)
network.size(msmAt50)
```

```
[1] 1000
```

```
plot(msmAt50,coord=network.layout.animate.MDSJ(msmAt50),vertex.cex=0.5)
```

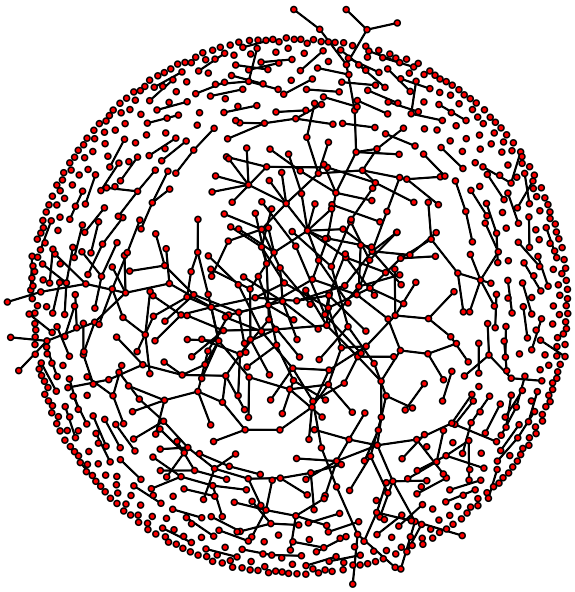
```
[1] "MDSJ starting stress: 960554.5827592497"
[2] "MDSJ ending stress: 153000.49609808027"
```



In this case, the default distance must have been set to about 31 (square root of 1000). This results in the giant component being well separated from the smaller components and isolates. Although this certainly focuses visual attention nicely on the big component, it squishes up the rest of the network. We can try setting the value lower.

```
plot(msmAt50,coord=network.layout.animate.MDSJ(msmAt50,default.dist=10),
     vertex.cex=0.5)
```

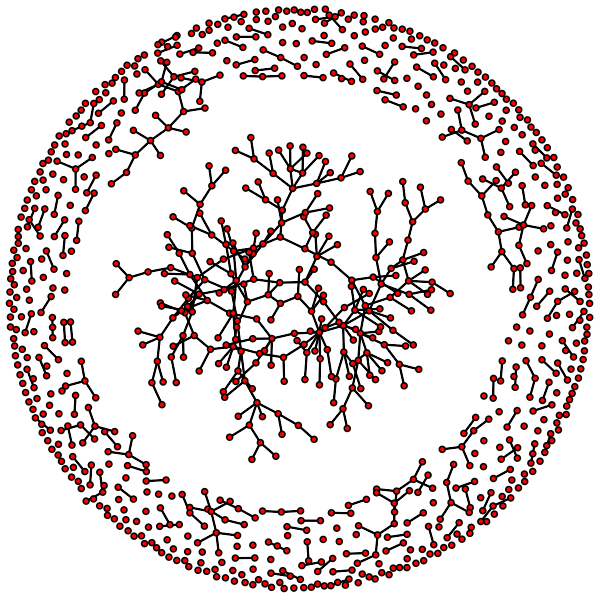
```
[1] "MDSJ starting stress: 897225.3819929918"
[2] "MDSJ ending stress: 173729.60379015136"
```



But then the `default.dist` value was too small to effectively separate the components, resulting in a lot of unnecessary edge crossing. So lets try an intermediate value.

```
plot(msmAt50,coord=network.layout.animate.MDSJ(msmAt50,default.dist=18),
     vertex.cex=0.5)
```

```
[1] "MDSJ starting stress: 937958.5392749084"
[2] "MDSJ ending stress: 160693.1602308798"
```



For this network, `default.dist=18` seems to give a reasonable compromise between spacing and scaling, but it can still lead to some edge overlaps. We can now compute the overall movie to see how it works at various points in time. And then peek at four time points to see if the parameter is going to give reasonable values over the time range of the movie.

```
# calculating for this network size will be slow
# expect this command to take 5 minutes
compute.animation(msm.sim,animation.mode='MDSJ',default.dist=18)
```

```
[1] "No slice.par found, using"
slice parameters:
  start:1
  end:10
  interval:1
  aggregate.dur:1
  rule:latest
```

```
[1] "Calculating layout for network slice from time  1 to 2"
[1] "MDSJ starting stress: 940218.7535148488"
[2] "MDSJ ending stress: 173298.2404100899"
[1] "Calculating layout for network slice from time  2 to 3"
[1] "MDSJ starting stress: 205250.86590475857"
[2] "MDSJ ending stress: 172355.41573471975"
[1] "Calculating layout for network slice from time  3 to 4"
[1] "MDSJ starting stress: 204707.509375172"
[2] "MDSJ ending stress: 171122.34096101898"
[1] "Calculating layout for network slice from time  4 to 5"
[1] "MDSJ starting stress: 214421.40201962698"
[2] "MDSJ ending stress: 169532.67245937898"
[1] "Calculating layout for network slice from time  5 to 6"
[1] "MDSJ starting stress: 199460.35092812043"
[2] "MDSJ ending stress: 168552.52868660947"
[1] "Calculating layout for network slice from time  6 to 7"
[1] "MDSJ starting stress: 205554.93324315475"
[2] "MDSJ ending stress: 169041.9782091806"
[1] "Calculating layout for network slice from time  7 to 8"
[1] "MDSJ starting stress: 207395.22259980315"
[2] "MDSJ ending stress: 168208.10683705707"
[1] "Calculating layout for network slice from time  8 to 9"
[1] "MDSJ starting stress: 219593.6251910919"
[2] "MDSJ ending stress: 161388.82583111903"
[1] "Calculating layout for network slice from time  9 to 10"
[1] "MDSJ starting stress: 194620.62854219033"
[2] "MDSJ ending stress: 163508.9422386753"
[1] "Calculating layout for network slice from time  10 to 11"
[1] "MDSJ starting stress: 226624.03269790206"
[2] "MDSJ ending stress: 160631.21828665287"
```

```
filmstrip(msm.sim,frames=4,displaylabels=FALSE,vertex.cex=0.5)
```

So it seems like it will work acceptably, but by the end of the movie the giant component will have grown enough to start squishing the rest of the network.

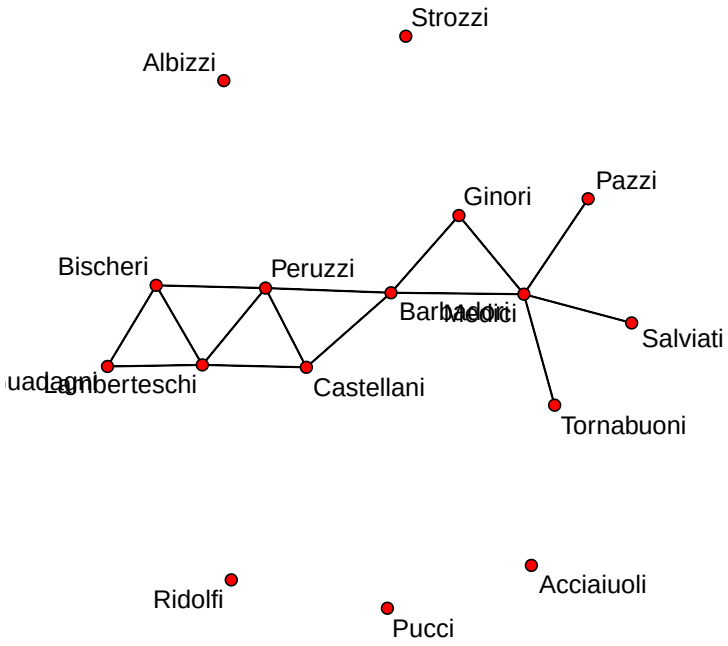
4 Advanced examples

4.1 A more complete tergm/stergm example

This is an expanded version of the demo from the beginning of the tutorial. It illustrates how to run a `tergm` simulation and some additional useful features for working with `tergm` model output.

Let say we want to construct a basic but plausible statistical model of a dynamic network defined as a STERGM, and we want to see what one possible realization of the network might look like. Using statnet's `tergm` package, we can 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. First we load up the data.

```
library(tergm)    # lib for temporal ergm simulations
data("florentine") # an example network
plot(flobusiness,displaylabels=T)
```



Then we ask `stergm` to fit a model with a random edge probability ("edges") plus a type of clustering defined as a decreasing marginal return on the number of edge-wise shared partners ("gwesp"). We also include a random dissolution process on the edges to keep the density from simply increasing at each time step. For more background on the modeling process please see the `tergm` workshop materials at `https://statnet.csde.washington.edu/trac/wiki/Sunbelt2013#WorkshopMaterials`.

```
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"  )
```

```
Iteration 1 of at most 20:
Convergence test P-value: 3.5e-226
The log-likelihood improved by 0.06815
Iteration 2 of at most 20:
Convergence test P-value: 2.7e-55
The log-likelihood improved by 0.01733
Iteration 3 of at most 20:
Convergence test P-value: 9.5e-08
The log-likelihood improved by 0.00288
Iteration 4 of at most 20:
Convergence test P-value: 6.9e-02
The log-likelihood improved by 0.0004833
Iteration 5 of at most 20:
Convergence test P-value: 1.2e-01
The log-likelihood improved by 0.0003976
Iteration 6 of at most 20:
Convergence test P-value: 2.8e-01
The log-likelihood improved by 0.0002161
Iteration 7 of at most 20:
Convergence test P-value: 8.4e-02
The log-likelihood improved by 0.0004489
Iteration 8 of at most 20:
Convergence test P-value: 7.8e-01
Convergence detected. Stopping.
The log-likelihood improved by < 0.0001

This model was fit using MCMC.  To examine model diagnostics and check for degeneracy, use t
========  Phase 1: Burn in, get initial gradient values, and find a configuration under whic
Burning in... Done.
Attempt 1 :
All parameters have some effect and all statistics are moving. Proceeding to Phase 2.
========  Phase 2: Find and refine the estimate. ========
Subphase 2.1 ////////\\\\\
Subphase 2.2 \/\/\\\/\\\\\
Subphase 2.3 \\\\/\\\\\
Subphase 2.4 \\\\\
========  Phase 3: Simulate from the fit and estimate standard errors. ========
```

```
Subphase 2.5 \\//\\\\\
========  Phase 3: Simulate from the fit and estimate standard errors. ========
```

After the model has been estimated, we can take a number of sequential draws from it to see how the network might "evolve" over time, and output these discrete networks as a single `dynamicNetwork` object.

```
stergm.sim.1 <- simulate.stergm(stergm.fit.1,
                    nsim=1, time.slices = 25)
```

Now that we have our dataset, it is time to render the movie. We will specify some render parameters using `render.par`. The `show.stats` parameter accepts a tergm summary formula to be evaluated, and print the model statistics for each slice on the appropriate frame of the movie. We can specify the formula we used for formation: `~edges+gwesp(0,fixed=T)`

```
render.par=list(tween.frames=5,show.time=TRUE,
                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. As we are fairly familiar with the output by now, we can suppress it by adding a `verbose=FALSE` argument.

```
render.animation(stergm.sim.1,render.par=render.par,
                 edge.col="darkgray",displaylabels=TRUE,
                 label.cex=.6,label.col="blue",verbose=FALSE)
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, 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.

4.2 Constructing a movie from external data in matrix form

At this point you might be thinking: "All of this dynamic stuff is well and good, but my data were collected in panels and stored as matrices. Can I still make a network animation?"

The answer is yes! We will use the example Harry Potter Support Networks of Goele Bossaert and Nadine Meidert (Bossaert, G. and Meidert, N. , 2013). They have coded the peer support ties between 64 characters appearing in the text of each of the well-known fictional J. K. Rowling novels and made the data available for general use in the form of 6 text formatted adjacency matrices and several attribute files. You can download and unzip the data files from `http://www.stats.ox.ac.uk/~snijders/siena/HarryPotterData.html`, or use the R code below to load in directly from the zip file.

```
# tmp filename for the data
webLoc<-"http://www.stats.ox.ac.uk/~snijders/siena/bossaert_meidert_harrypotter.zip"
temp_hp.zip <- tempfile()
download.file(webLoc,temp_hp.zip)
# read in first matrix file, unziping in the process
hp1 <- read.table(unz(temp_hp.zip, "hpbook1.txt"),
                  sep=" ",stringsAsFactors=FALSE)
```

Notice the `stringsAsFactors=FALSE` argument to `read.table`. This prevents the strings from being converted into factors, which then may unexpectedly appear as integers causing all kinds of headaches. Now we should confirm that the data have the shape we expect. Since there are 64 characters, we expect a 64x64 matrix.

```
dim(hp1)
```

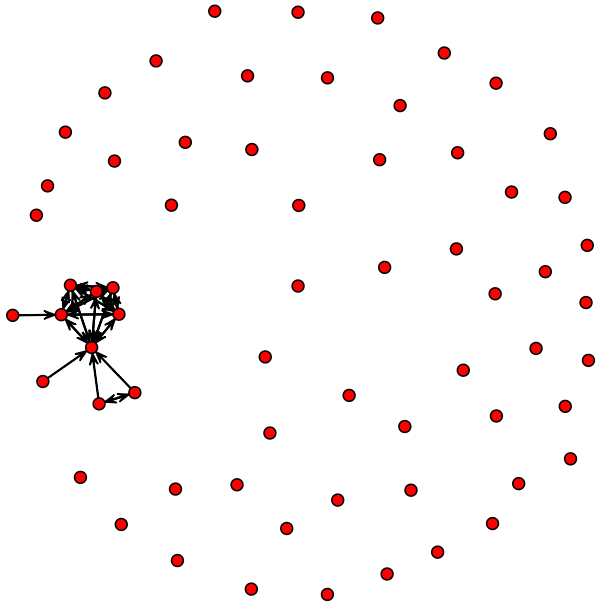
```
[1] 64 64
```

```
# peek at "upper-left" corner of file
hp1[1:12,1:12]
```

```
   V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12
1   0  0  0  0  0  0  0  0  0   0   0   0
2   0  0  0  0  0  0  0  0  0   0   0   0
3   0  0  0  0  0  0  0  0  0   0   0   0
4   0  0  0  0  0  0  0  0  0   0   0   0
5   0  0  0  0  0  0  0  0  0   0   0   0
6   0  0  0  0  0  0  0  0  0   0   0   0
7   0  0  0  0  0  0  0  0  0   0   0   0
8   0  0  0  0  0  0  0  0  0   0   0   0
9   0  0  0  0  0  0  0  0  0   0   0   0
10  0  0  0  0  0  0  0  0  0   0   0   0
11  0  0  0  0  0  0  0  0  0   0   1   0
12  0  0  0  0  0  0  0  0  0   0   0   0
```

And lets try quickly converting to a network static and plotting to make sure it works.

```
plot(as.network(hp1))
```



Since that seems to work, lets load in the rest of the files.

```
# tmp filename for the data
hp2 <- read.table(unz(temp_hp.zip, "hpbook2.txt"),
                  sep=" ",stringsAsFactors=FALSE)
hp3 <- read.table(unz(temp_hp.zip, "hpbook3.txt"),
                  sep=" ",stringsAsFactors=FALSE)
hp4 <- read.table(unz(temp_hp.zip, "hpbook4.txt"),
                  sep=" ",stringsAsFactors=FALSE)
hp5 <- read.table(unz(temp_hp.zip, "hpbook5.txt"),
                  sep=" ",stringsAsFactors=FALSE)
hp6 <- read.table(unz(temp_hp.zip, "hpbook6.txt"),
                  sep=" ",stringsAsFactors=FALSE)
```

To construct a dynamicNetwork, we will arrange them in a list and then convert it with the `dynamicNetwork()` utility function.

```
hpList<-list(hp1,hp2,hp3,hp4,hp5,hp6)
# convert adjacency matrices to networks
```

```
hpList<-lapply(hpList,as.network.matrix,matrix.type='adjacency')
# convert list of networks to networkDynamic
harry_potter_support<-networkDynamic(network.list=hpList)
```

```
Neither start or onsets specified, assuming start=0
Onsets and termini not specified, assuming each network in network.list should have a discre
Argument base.net not specified, using first element of network.list instead
Created net.obs.period to describe network
 Network observation period info:
  Number of observation spells: 1
  Maximal range of observations: 0 to 6
  Temporal mode: discrete
  Time unit: step
  Suggested time increment: 1
```

Next we will read in the names from the auxillary file and attach them to the network as vertex names.

```
# read in and assign the names
names<-read.table(unz(temp_hp.zip, "hpnames.txt"),
                  sep="\t",stringsAsFactors=FALSE,header=TRUE)
network.vertex.names(harry_potter_support)<-names$name
```

And similarly for the other attributes.

```
# read in and assign the attributes
attributes<-read.table(unz(temp_hp.zip, "hpattributes.txt"),
                       sep="\t",stringsAsFactors=FALSE,header=TRUE)
harry_potter_support%v%'id'<-attributes$id
harry_potter_support%v%'schoolyear'<-attributes$schoolyear
harry_potter_support%v%'gender'<-attributes$gender
harry_potter_support%v%'house'<-attributes$house
```

As a courtesy to other users (or our future selves) we will add a special `net.obs.period` attribute to provide some meta data about the temporal model for this dataset. See `?net.obs.period` for more information.

```
harry_potter_support%n%'net.obs.period'<-list(
  observations=list(c(0,6)),mode="discrete",
  time.increment=1,time.unit="book volume")
```

Now for the important question: which vertex is Harry Potter?

```
which(network.vertex.names(harry_potter_support)=="Harry James Potter")
```

```
[1] 25
```

When we render out the movie it is going to look like *Harry Potter and the Philosopher's Stone*, right?

```
render.animation(harry_potter_support)
```

Lets tweak it a bit for some more refinement

```
compute.animation(harry_potter_support,
                  animation.mode='MDSJ',
                  default.dist=2)
render.animation(harry_potter_support,
            render.par=list(tween.frames=20),
            vertex.cex=0.8,label.cex=0.8,label.col='gray',
            # make shape relate to school year
            vertex.sides=harry_potter_support%v%'schoolyear'-1983,
            # color by gender
            vertex.col=ifelse(harry_potter_support%v%'gender'==1,'blue','green'),
            edge.col="#CCCCCC55"
)
```

One challenge of constructing movies from matrices is that (as the authors of this dataset note) there is often a great deal of change between network survey panels.

Question: How could dynamic network data (like in the example above) be collected differently to support animations and more flexible analysis of dynamics?

4.3 Transmission trees and constructed animations

We will simulate a fictitious rumor transmission network using code from the "Making Lin Freeman's windsurfers gossip" section of the `networkDynamic` vignette, and then examine various ways to look at the transmission. Finally, we will construct a new movie and animated transition using the output of several networks and algorithms.

The code below defines a function to run the simulation, sets initial seeds (starts the rumor) and then runs the simulation The EpiModel package (Jenness S, et. al , 2014) includes much better utilities for simulating transmission networks with various realistic properties. If you don't care about the details of the simulation, just execute the entire block of code below to load in the funciton.

```
# function to simulate transmission
runSim<-function(net,timeStep,transProb){
  # loop through time, updating states
  times<-seq(from=0,to=max(get.change.times(net)),by=timeStep)
  for(t in times){
    # find all the people who know and are active
    knowers <- which(!is.na(get.vertex.attribute.active(
      net,'knowsRumor',at=t,require.active=TRUE)))
```

```
    # get the edge ids of active friendships of people who knew
    for (knower in knowers){
      conversations<-get.edgeIDs.active(net,v=knower,at=t)
      for (conversation in conversations){
        # select conversation for transmission with appropriate prob
        if (runif(1)<=transProb){
          # update state of people at other end of conversations
          # but we don't know which way the edge points so..
          v<-c(net$mel[[conversation]]$inl,
                 net$mel[[conversation]]$outl)
          # ignore the v we already know and people who already know
          v<-v[!v%in%knowers]
          if (length(v)){
            activate.vertex.attribute(net,"knowsRumor",TRUE,
                                      v=v,onset=t,terminus=Inf)
            # record who spread the rumor
            activate.vertex.attribute(net,"heardRumorFrom",knower,
                                      v=v,onset=t,length=timeStep)
            # record which friendships the rumor spread across
            activate.edge.attribute(net,'passedRumor',
                      value=TRUE,e=conversation,onset=t,terminus=Inf)
          }
        }
      }
    }
    return(net)
}
```

We next need to load in the `networkDynamic` data object, set up the initial state of the simulation, and then use the function we defined above to propagate the rumor.

```
data(windsurfers)    # let's go to the beach!
# set initial params...
timeStep <- 1  # units are in days
transProb <- 0.2 # how likely to tell in each conversation/day
# start the rumor out on vertex 1
activate.vertex.attribute(windsurfers,"knowsRumor",TRUE,v=1,
                          onset=0-timeStep,terminus=Inf)
activate.vertex.attribute(windsurfers,"heardRumorFrom",1,v=1,
                          onset=0-timeStep,length=timeStep)
activate.edge.attribute(windsurfers,'passedRumor',value=FALSE,
onset=-Inf,terminus=Inf)
# run the sim!
windsurfers<-runSim(windsurfers,timeStep,transProb)
```

Now the windsurfers network should have dynamic attributes indicating who knows the rumor, who they heard it from, and which edges passed it.

```
list.vertex.attributes(windsurfers)
```

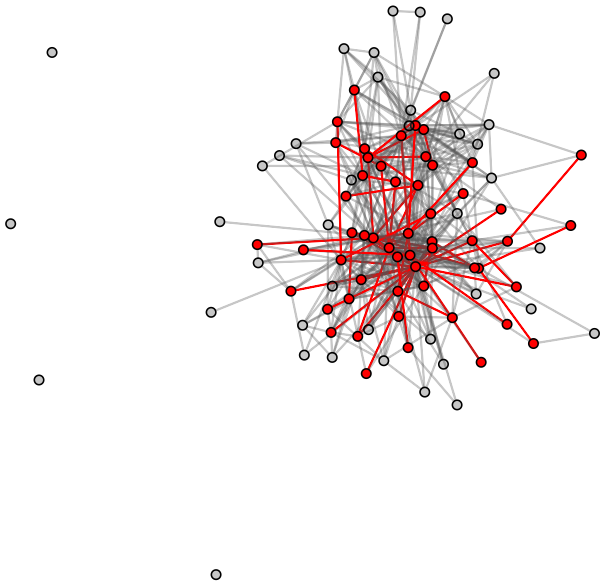
```
[1] "active"                "group1"                "group2"
[4] "heardRumorFrom.active" "knowsRumor.active"     "na"
[7] "regular"               "vertex.names"
```

```
list.edge.attributes(windsurfers)
```

```
[1] "active"             "na"                 "passedRumor.active"
```

Lets plot the time-aggregate network with the infected vertices and edges highlighted by their status in the last timepoint

```
# plot the aggregate network, hiliting infected
plot(windsurfers,
     vertex.col=ifelse(
       !is.na(get.vertex.attribute.active(windsurfers,'knowsRumor',at=31)),
       'red','#55555555'
     ),
     edge.col=ifelse(
       get.edge.attribute.active(windsurfers,'passedRumor',at=31),
       'red','#55555555'
     ),
     vertex.cex=0.8
)
```



From the plot, it appears the rumor stayed mostly in one of the communities. But without being able to see the timing of the infections, it is difficult to tell what is going on. Since we know that the high vertex turnover makes it hard to render this as a basic movie, we can create a "flip-book" style movie, where we will keep the vertex positions fixed and just animate the dynamics.

```
# record the coords produced by plot
coords<-plot(windsurfers)
# set them as animation coords directly,without layout
activate.vertex.attribute(windsurfers, 'animation.x',coords[,1],
                          onset=-Inf, terminus=Inf)
activate.vertex.attribute(windsurfers, 'animation.y',coords[,2],
                          onset=-Inf,terminus=Inf)
# construct slice par to indicate time range to render
windsurfers%n%'slice.par'<-list(start=-31,end=0,interval=1,
                                aggregate.dur=31,rule='latest')
```

Now we will render it out to file, using functional plot arguments to color edges and vertices by their infection status at each time point.

```
saveVideo(
  render.animation(windsurfers,
            render.par=list(initial.coords=coords),
            # color edges by rumor status
            edge.col=function(slice){
              ifelse(slice%e%'passedRumor','red','#00000055')
            },
            # color vertices by rumor status
            vertex.col=function(slice){
              ifelse(!is.na(slice%v%'knowsRumor'),'red','gray')
            },
            # change text of label to show time and total infected.
            xlab=function(slice,terminus){
              paste('time:',terminus,' total infected:',
                    sum(slice%v%'knowsRumor',na.rm=TRUE))
            },
            vertex.cex=0.8,label.cex=0.8,render.cache='none'
  )
  ,video.name='windsurferFlipbook.mp4'
)
```

Notice that we did something really funky with the `slice.par` parameters. We are using `aggregate.dur=31`–equal to the entire duration of the network– and starting at -31. This makes it so we are effectively sliding along a giant bin which is gradually accumulating more of the network edges with each step until it contains the whole thing. We also used an initial coordinate setting for the vertices (otherwise they would appear at zero when first entering) and functional attribute definitions for vertex and edge colors.

Question: Why does the infection fail to spread across many of the edges to infected vertices visable in this animation?

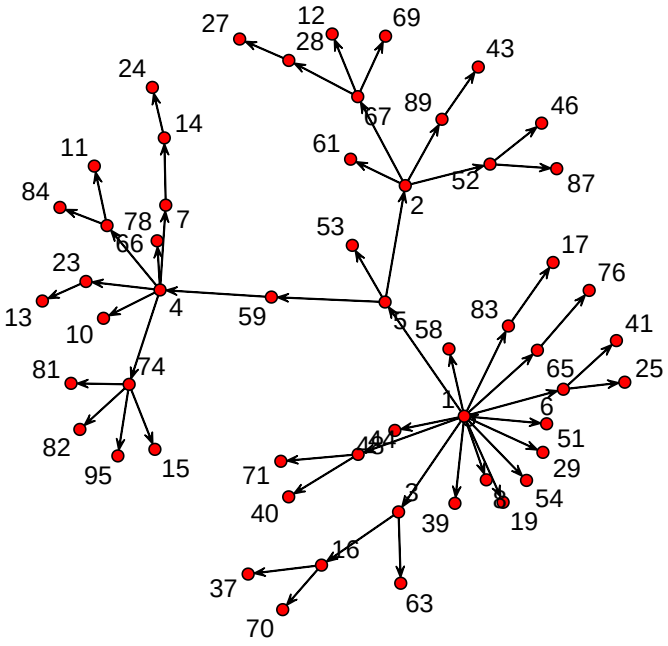
In this view, it is still quite difficult to see the sequence of infections and the infection path. We can try extracting the infection so that we can visualize it directly. The function below will extract a network consisting only of the rumor-infected vertices and edges in the original network that passed the rumor. The edges will be directed, so we can see it as a tree. Don't need to look at this in detail, just load it up.

```
# function to extract the transmission tree
# as a directed network
transTree<-function(net){
  # for each vertex in net who knows
  knowers <- which(!is.na(get.vertex.attribute.active(net,
```

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

Now lets use the newly-created `transTree()` function to find the transmission tree, and plot it.

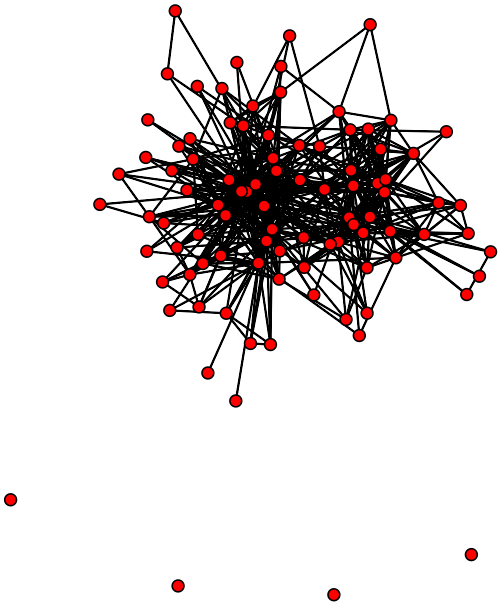
```
windTree<-transTree(windsurfers)
plot(windTree,displaylabels=TRUE)
```



We don't necessarily have to use `compute.animation` to construct the sequence of coordinates we are going to render in a movie. We can assemble a new synthetic network to visualize and, if we are careful, we can even use coordinates from one layout and apply them to another. In the next example, we will assemble an animated transition from the full cumulative network into a hierarchical representation of the transmission tree (this part requires a working installation of Graphviz).

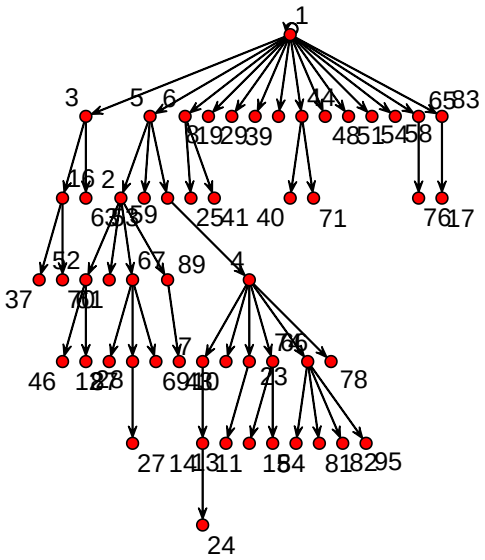
First, lets compute a layout for the cumulative across-time network

```
# calculate coord for aggregate network
windAni<-network.collapse(windsurfers,onset=-Inf,terminus=Inf,
                          rule='latest')
cumCoords<-plot.network(windAni)
cumCoords<-layout.normalize(cumCoords)
```



Next, compute a hierarchical tree layout of the transmission tree.

```
# calculate coords for transmission tree
treeCoords<-network.layout.animate.Graphviz(windTree,
              layout.par=list(gv.engine='dot',
                              gv.args='-Granksep=3'))
treeCoords<-layout.normalize(treeCoords)
# peek at it
plot(windTree,coord=treeCoords,displaylabels=TRUE,jitter=FALSE)
```



Now lets assemble a dynamic network on `windAni` frame-by-frame applying the coordinates we saved from the previous layouts of the cumulative and transmission tree networks.

For the first frame, we want all vertices and edges active and we attach the coordinates we calculated for the cumulative network to position the vertices.

```
activate.vertices(windAni,onset=0,terminus=1)
activate.edges(windAni,onset=0,terminus=1)
# store the plain network coords for cumulative network
activate.vertex.attribute(windAni,'animation.x',cumCoords[,1],
                          onset=0,terminus=Inf)
activate.vertex.attribute(windAni,'animation.y',cumCoords[,2],
                          onset=0,terminus=Inf)
```

For the second frame, we activate vertices that "know" and all the edges that passed the rumor.

```
activate.vertices(windAni,onset=1,terminus=3,
                  v=which(windAni%v%'knowsRumor'))
```

```
activate.edges(windAni,onset=1,terminus=3,
               e=which(windAni%e%'passedRumor'))
```

For the third frame, the edges and vertices will still be active, but we want to transition the positions to the "tree", so we attach the tree coordinates at that time.

```
activate.vertex.attribute(windAni,'animation.x',treeCoords[,1],
          onset=2,terminus=Inf,v=network.vertex.names(windTree))
activate.vertex.attribute(windAni,'animation.y',treeCoords[,2],
          onset=2,terminus=Inf,v=network.vertex.names(windTree))

# construct slice par to indicate time range to render
windAni%n%'slice.par'<-list(start=0,end=2,interval=1,
                            aggregate.dur=1,rule='latest')
# render it
saveVideo(
  render.animation(windAni,
            edge.col=function(slice){
              ifelse(!is.na(slice%e%'passedRumor'),
                     'red','#00000055')
            },
            vertex.col=function(slice){
              ifelse(!is.na(slice%v%'knowsRumor'),
                     'red','gray')
            },
            vertex.cex=0.8,label.cex=0.8,label.pos=1,
                   render.cache='none'
  )
 , video.name='windsurferTreeTransition.mp4')
```

```
[1] "rendering 10 frames for slice 0"
[1] "rendering 10 frames for slice 1"
[1] "rendering 10 frames for slice 2"
```

Exercise: Generate a similar movie, but use the coordinates of the non-hierarchical tree layout (i.e. don't use Graphviz)

Exercise: Choose one of the dynamic dataset (perhaps one from the networkDynamicData package) and construct an animation.

5 Misc topics

5.1 Compressing video output

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[13], determining the right settings can be a trial and error process. As an alternative, Handbrake `http://handbrake.fr/` is an excellent and easy to use graphical tool for doing video compression into the web-standard H.264 codec with appropriate presets.

5.2 Transparent colors

Using a bit of transparency can help a lot with readability for many visualizations. Transparency makes it so that overlapping edges can show through each other and the less saturated colors tend to be less distracting. Many of the R plot devices support transparency, but specifying the color codes with transparency can be a bit awkward. One way we demonstrated is to include an alpha parameter to the `rgb()` function which defines a color given numeric values for proportions of red, green, blue, and alpha (the computer graphics term for transparency). If you want to just make one of R's existing named colors more transparent, try `grDevices::adjustcolor()`. The most concise way is to specify colors as HTML color codes with an alpha channel. These are 8-"digit" hexadecimal strings in the format`"#RRGGBBAA"`. Each hexadecimal "digit" has 16 possible values, ranging from 0 to F. The first pair of digits (RR) gives the percent of red, the second (BB) blue, third (GG) green, and last (AA) gives the alpha percent. For example, 50% black is `"#00000088"`, 50% green would be `"#00FF0088"`. These can be hard to remember, but useful once you have written down the colors you want.

```
# 50% blue
rgb(0,0,1,0.5)
```

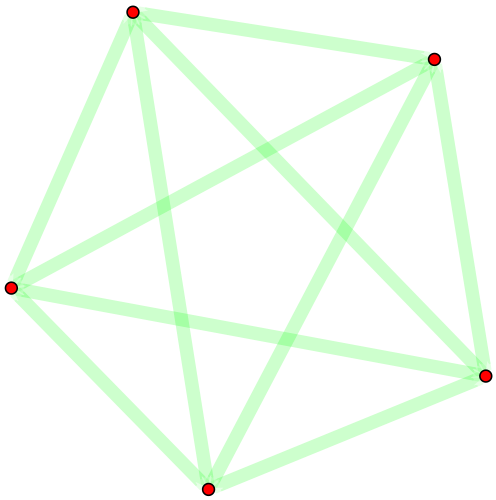
```
[1] "#0000FF80"
```

```
# 50% pink
grDevices::adjustcolor('pink',alpha.f=0.5)
```

```
[1] "#FFC0CB80"
```

```
# plot example net with 10% green
colorNet<-network.initialize(5)
colorNet[,]<-1
plot(colorNet,edge.lwd=20,edge.col=rgb(0,1,0,0.1))
```

[13] The default settings for ffmpeg differ quite a bit depending on platform, some installations may give decent compression without tweaking the settings



5.3 Setting background colors and margins

You may have noticed that the network plots we have seen so far have fairly wide margins which allow extra room for labels and annotations. It is possible to adjust the margins, and other generic plot commands, by passing in `par` arguments to `render.animation` via `plot.par`. For example, we can set all of the margins to zero with the `mar` command and change the background to gray with `bg='gray'`. See `?par` for a list of high-level plot parameters suitable for `plot.par`.

```
render.animation(wheel,plot.par=list(bg='gray',mar=c(0,0,0,0)),
                 edge.col='white')
```

5.4 Tips for working with large networks

As the `msm.sim` example probably demonstrated, rendering animations of large networks can be very time consuming. Here are a number of solutions we have found effective for working with networks that take a lot of time or are simply too large to render directly as animations.

- Test and refine algorithm settings and graphics parameters on a short sequence or single time slice of larger network before running the animation for its full duration
- Render full animations of large networks directly to disk instead previewing in the R plot window. See the use of the `saveVideo` and `render.cache='none'` arguments in example in the Output Formats section.
- Extract and visualize a relevant subset of the network. For example, for this movie illustrating the effects of concurrency, we able were effectively visualize the relationships among the 800 vertex infected component of simulation with a population of 10,000: `http://statnet.csde.washington.edu/movies/`. This does introduce some bias into the viewer's perception of the network's overall properties.

5.5 How to get help

Here are some suggestions on where to look for help if you run into problems or have questions:

- R's built in documentation feature. Each function has a help page, usually with some examples: i.e. `?render.animation`. The listing of all the documentation pages can be found with `help(package='ndtv')` or `help(package='networkDynamic')`
- The ndtv package vignette (includes much of the material from this workshop): `browseVignettes(package='ndtv')`
- The statnet wiki has various FAQ pages and resources: `https://statnet.csde.washington.edu/trac/wiki`
- You can always subscribe to the statnet mailing list and post your problem there: `https://mailman2.u.washington.edu/mailman/listinfo/statnet_help`

6 Limitations

Before we create unreasonable expectations, we should be clear that you can't just throw any network into `render.animation` and expect good results.

- Not for "big data". So far we've mostly used it successfully with networks of 1k vertices and several hundred time steps, but that takes a long time to render.
- Animations of dense or fully-connected networks may not be useful.
- The animation techniques work best when a relatively small number of ties are changing between time slices and vertex turnover rates (births and death) are low.
- You may need to think carefully about how to aggregate and smooth the dynamics over time in order to reveal the underlying patterns of interest. This is analogous to choosing bin-widths for histograms in order to reveal the distributional form of the data.
- Good results may require considering how the weights and measures of your input data can be translated meaningfully to the implicit "space" you are embedding your data in.

As the `ndtv` package is still very much under development, we would greatly appreciate your suggestions and feedback.

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.

Bossaert, G. and Meidert, N. (2013) 'We are only as strong as we are united, as weak as we are divided'. A dynamic analysis of the peer support networks in the Harry Potter books. *Open Journal of Applied Sciences*, Vol. 3 No. 2, pp. 174-185. DOI: `http://dx.doi.org/10.4236/ojapps.2013.32024`

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 (2014). *networkDynamic: Dynamic Extensions for Network Objects*. R package version 0.7, 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 (2014). *ndtv: Network Dynamic Temporal Visualizations*. R package version 0.5.1, http://CRAN.R-project.org/package=ndtv.

John Ellson et al (2001) Graphviz – open source graph drawing tools *Lecture Notes in Computer Science*. Springer-Verlag. p483-484 http://www.graphviz.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/.

Jenness S, Goodreau S, Wang L and Morris M (2014). *EpiModel: Mathematical Modeling of Infectious Disease*. The Statnet Project (http://www.statnet.org). R package version 0.95, CRAN.R-project.org/package=EpiModel.

Krivitsky P and Handcock M (2014). *tergm: Fit, Simulate and Diagnose Models for Network Evolution based on Exponential-Family Random Graph Models*. The Statnet Project (http://www.statnet.org). R package version 3.1.4, 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/.