

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

Skye Bender-deMoll

January 28, 2014

Contents

1	Introduction to workshop	1
1.1	What is ndtv?	1
1.2	Authors	1
1.3	Workshop prerequisites	2
1.4	A Quick Demo	2
1.5	Limitations	11
2	The basics	11
2.1	Installing ndtv and its external dependencies	11
2.1.1	Installing ndtv and its R package dependencies	11
2.1.2	Installing FFmpeg for saving animations	11
2.1.3	Installing Java and MDSJ setup	12
2.2	Understanding how ndtv works	13
2.2.1	“Wheel” work along example	13
2.2.2	What happened automatically	16
2.2.3	Doing it step by step	17
2.3	Demonstrate output formats	21
2.3.1	saveVideo	21
2.3.2	saveGIF	22
2.3.3	saveLatex	22
3	Using ndtv effectively	22
3.1	Comparisons of Layout algorithms	22
3.1.1	Why we don’t (yet) use Fruchterman-Reingold	23
3.1.2	Kamada-Kawai adaptation	23
3.1.3	MDSJ (Multidimensional Scaling for Java)	23
3.1.4	Graphviz layouts	23
3.1.5	Use a TEA attribute or write your own	24

3.2	Slicing and aggregating time	24
3.2.1	Slicing panel data	24
3.2.2	Slicing streaming data	26
3.2.3	Vertex dynamics	30
3.3	Animating graphic attributes	33
3.3.1	Using dynamic attributes (TEAs)	33
3.3.2	Functional plot arguments	34
3.4	Using and aggregating edge weights	35
3.5	Adjusting isolates and components separation with <code>layout.distance</code>	41
4	Advanced examples	45
4.1	Transmission trees and assembled layouts	45
4.2	A more complete <code>tergm</code> / <code>stergm</code> example	53
4.3	Constructing a movie from external data in matrix form	55
5	Misc topics	58
5.1	Compressing video output	58
5.2	Transparent colors	58
5.3	How to get help	59

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. , 2013) 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 with members of the `statnet` <http://statnet.org> team. skyebend@skyeome.net
- 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 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 look at. A useful way to generate a dynamic network would be to define a STERGM equation giving basic formation and dissolution processes for the edges and then simulate a sequence network states using the `tergm` package. We will demonstrate this in a later example, but for now lets just load in an example dataset. The network `short.stergm.sim` is the output of a toy STERGM model based on Padgett's Florentine Family Business Ties network (`?padgett`).

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

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

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:

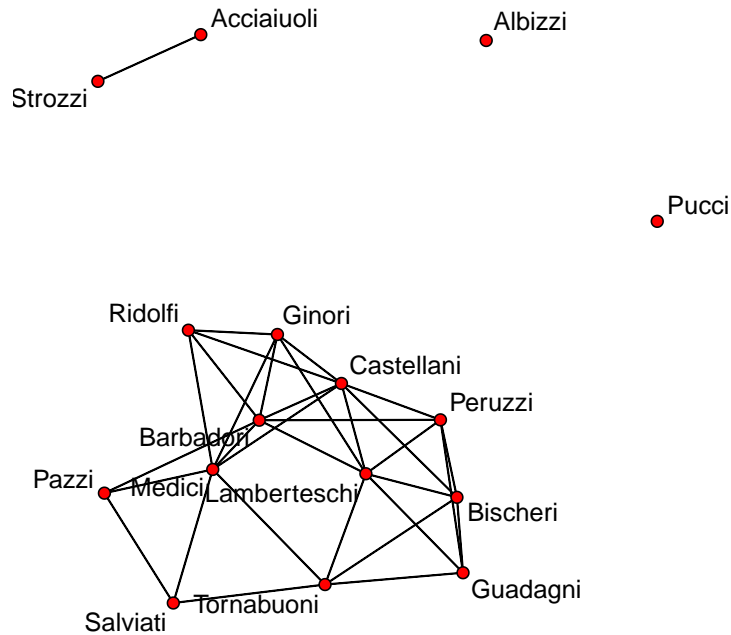
```
Network observation period info:
  Number of observation spells: 1
  Maximal range of observations: 0 to 25
  Temporal mode: discrete
  Time unit: step
  Suggested time increment: 1

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

plot(short.stergm.sim,displaylabels=TRUE)
```



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`.

Of course, the standard network plot command ignores the the dynamics of edges and vertices, so it actually plots the network aggregated over time –all the edges and vertices that are ever observed to be active. If we wanted to just see the original starting network we could first extract the network elements active at the beginning and then plot them.

```
plot(network.extract(short.stergm.sim,at=1),displaylabels=TRUE)
```


[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] "Calculating layout for network slice from time 13 to 14"
[1] "Calculating layout for network slice from time 14 to 15"
[1] "Calculating layout for network slice from time 15 to 16"
[1] "Calculating layout for network slice from time 16 to 17"
[1] "Calculating layout for network slice from time 17 to 18"
[1] "Calculating layout for network slice from time 18 to 19"
[1] "Calculating layout for network slice from time 19 to 20"
[1] "Calculating layout for network slice from time 20 to 21"
[1] "Calculating layout for network slice from time 21 to 22"
[1] "Calculating layout for network slice from time 22 to 23"
[1] "Calculating layout for network slice from time 23 to 24"
[1] "Calculating layout for network slice from time 24 to 25"
[1] "Calculating layout for network slice from time 25 to 26"
[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"
[1] "rendering 10 frames for slice 12"
[1] "rendering 10 frames for slice 13"
[1] "rendering 10 frames for slice 14"
[1] "rendering 10 frames for slice 15"
[1] "rendering 10 frames for slice 16"
[1] "rendering 10 frames for slice 17"
[1] "rendering 10 frames for slice 18"
[1] "rendering 10 frames for slice 19"
[1] "rendering 10 frames for slice 20"
[1] "rendering 10 frames for slice 21"
[1] "rendering 10 frames for slice 22"
[1] "rendering 10 frames for slice 23"

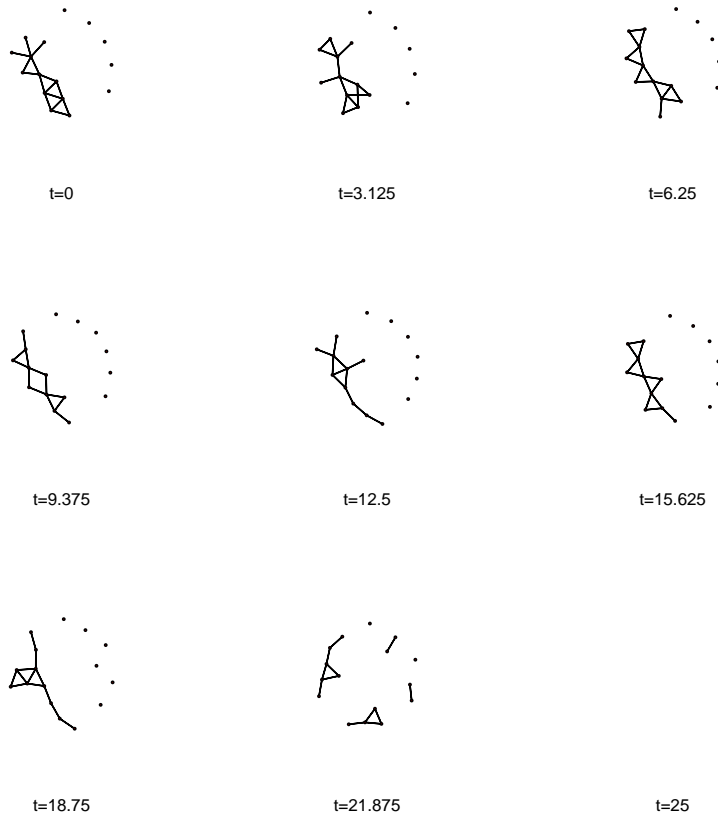
```
[1] "rendering 10 frames for slice 24"
[1] "rendering 10 frames for slice 25"
```

And then play it back in the R plot window

```
ani.replay()
```

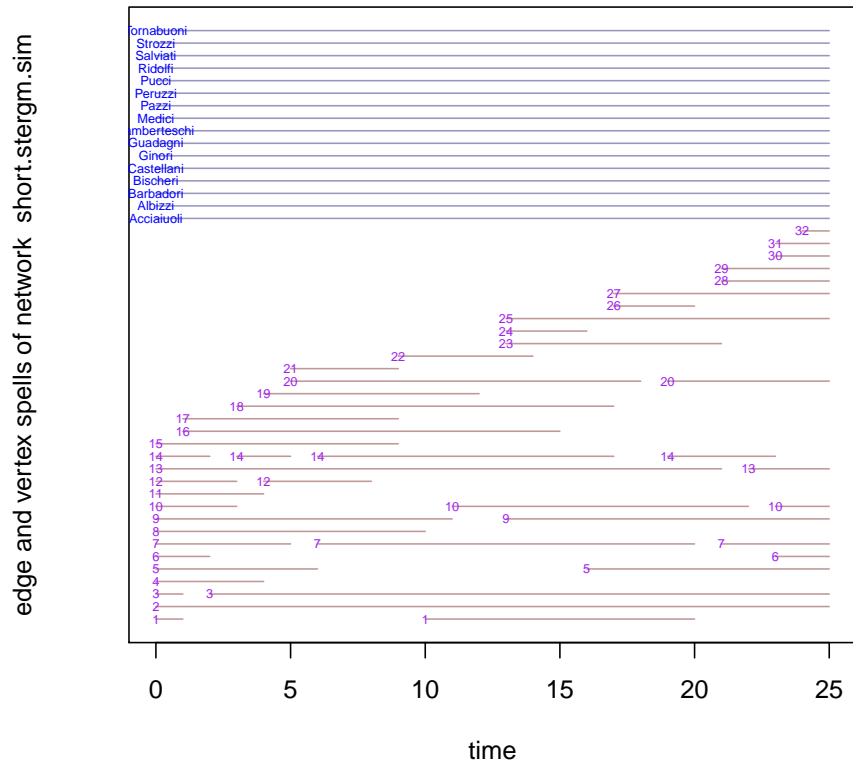
An animation is not the only way to display a sequence of views of the network. We could also use the `filmstrip()` function that will create a “small multiple” plot using frames of the animation to construct a visual summary of the network changes as a static plot.

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



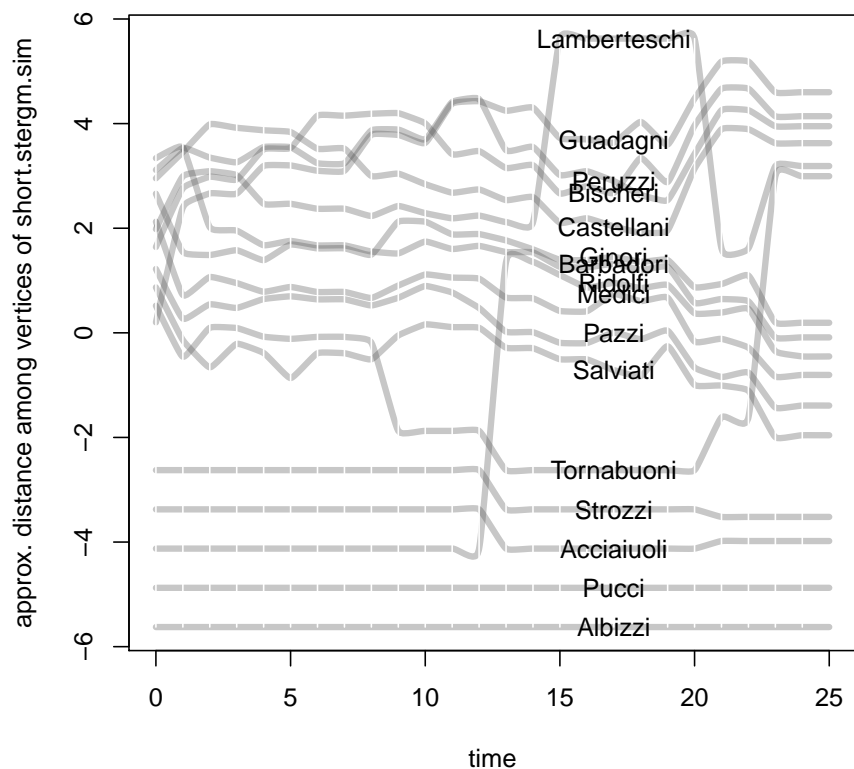
We can view the dynamics as a timeline or where we plot the active spells of edges and vertices over time (but don't show connectivity)

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

We are experimenting with a form of timeline or phase plot where vertices are positioned vertically by their proximity so that changes in network structure deflect their lines into new positions.

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



And of course we can always display the edge spells directly in a tabular form.

```
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

1.5 Limitations

Before we create too high of expectations, we should be clear that you can't just throw any network at this 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.
- Animations of dense networks may not be useful.
- Works best when a relatively small number of ties are changing between time slices,
- When vertex turnover rates (births and death) are high, you 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.

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` and requires external libraries (FFmpeg) to save movies out of the R environment, and Java for some of the better layout algorithms (these will be installed and explained later).

2.1.1 Installing ndtv and its R package dependencies

R should automatically install the dependencies when you install `ndtv`. So open up your R console, and if you don't already have `ndtv` installed do the following.

```
# uncomment this when the appropriate version is on CRAN
#install.packages('ndtv',repos='http://cran.us.r-project.org',
#                dependencies=TRUE)
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. FFmpeg <http://ffmpeg.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
```

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...)

2.2 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 “Wheel” work along example

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

```
wheel <- network.initialize(10) # create a toy network
add.edges.active(wheel,tail=1:9,head=c(2:9,1),onset=1:9, terminus=11)
add.edges.active(wheel,tail=10,head=c(1:9),onset=10, terminus=12)
```

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"
```

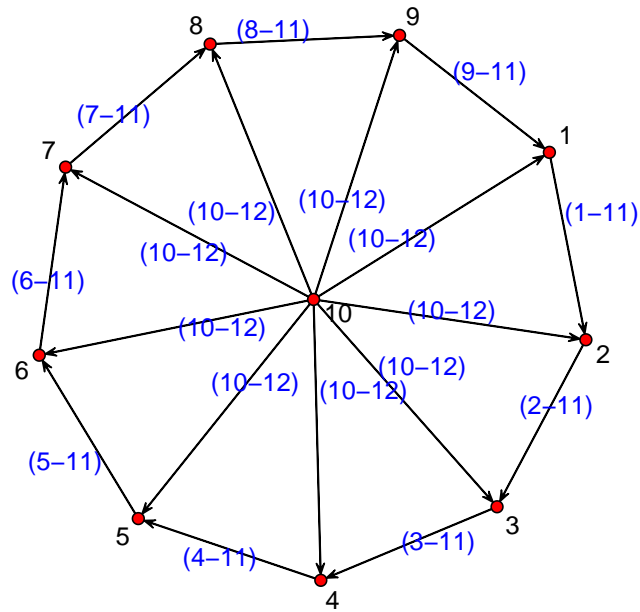
```
as.data.frame(wheel) # peek at edge dynamics as a data frame
```

	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

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. Now lets view it as a static plot, but labeling edges with their onset and termination times so we can check our work.

```
# 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?)

Next, render and view it as a dynamic movie

```
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 ¹. Then a set of “spoke” edges appear to draw a vertex into the center, and finally the rest of the wheel disappears.

2.2.2 What happened automatically

Simple right? Most of the difficult parts happened under the hood using default values.

1. We created a `networkDynamic` object named `wheel` containing information about the timing of edge activity.
2. `render.animation()` asked the package to create an animation for `wheel` but we didn’t include any arguments indicating what should be rendered or how.
3. Since `render.animation()` didn’t find any stored coordinate information about where to draw the vertices and edges, it (invisibly) called `compute.animation()` with default arguments to figure out where to position the vertices at each time step.

¹An example of the movie is located at http://statnet.csde.washington.edu/movies/ndtv_vignette/wheel.mp4

4. Because we didn't tell `compute.animation()` what time points to look at when doing its computations, it reported this, "No `slice.par` found", and made a guess as to when the animation should start and end (the earliest and latest observed times in the network) and how much time should be incremented between each set of layout coordinate calculations.
5. `compute.animation()` then stepped through the `wheel` network, using `network.collapse()` to get the appropriate active network, computing coordinates for each time step – using the previous step's coordinates as starting points – and storing them as a TEA attribute. (This was the "Calculating layout for network slice from time 1 to 2" ... part.)
6. `render.animation()` also stepped through the network, using the stored coordinates, `plot.network()` and `ani.record()` functions to cache snapshots of the network. It also caches a number of "tweening" images between each time step to smoothly interpolate the positions of the vertices. "rendering 10 frames for slice 1" ...
7. `ani.replay()` quickly redraws the sequence of cached images in the plot window as an animation.

2.2.3 Doing it step by step

For more precise control of the processes, layout algorithms, etc, we can call each of the steps in sequence. Lets work through and examine the output. First we will define a `slice.par` list of parameters to specify the time range that should have coordinates computed.

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

Then we ask it to compute the coordinates for the animation.

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

```
[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)

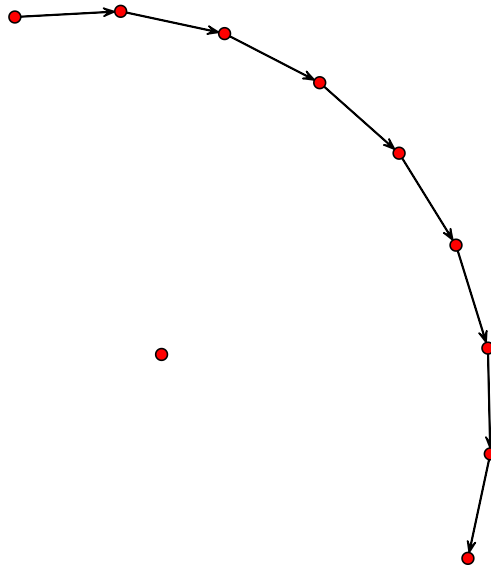
[1] "animation.x.active" "animation.y.active" "na"
[4] "vertex.names"

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

[1] -1.67329444 -0.79855354  0.09333343  0.96767373  1.81731327  2.08797789
[7]  1.12485009 -0.06876024 -1.23821125 -2.08797789
```

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. Since the coordinates are stored in the network, we could always just collapse the network at the time point and plot it with the appropriate values:

```
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, which is why we can pass in the standard network plotting arguments

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

```
[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"
```

We can adjust how many “tweening” frames are rendered. This indirectly impacts the perceived speed of the movie (more means slower and smoother). If we wanted 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

```
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 Demonstrate output formats

Now that we've got ffmpeg installed, we can save out some movies in useful formats.

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

- `ani.replay()` plays the animation back in the R plot window. (see `?ani.options` for more parameters)
- `saveVideo()` saves the animation as a movie file on disk (if the FFmpeg library is installed).
- `saveGIF()` creates an animated GIF (if ImageMagick's `convert` installed)
- `saveLatex()` creates an animation embedded in a pdf

Please see `?animation` and each function's help files for more details.

2.3.1 saveVideo

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")
```

Sometimes we may want to change the pixel dimensions of the movie output.

```
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), although this setting mostly impacts jpeg output.

```
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 will tend to bog 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 saveGIF

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")
```

2.3.3 saveLatex

```
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 interval of the animation, and use it to render a gif where each frame stays on screen for 2 seconds.

3 Using ndtv effectively

3.1 Comparisons of Layout algorithms

Producing “good” (for an admittedly ambiguous definition of good) layouts of networks is generally a computationally difficult problem.

Common goals:

- Layouts should remain as 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 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.

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. Each step can often avoid some work by using a previous solution as its starting point, and it is likely to find a solution that is similar to previous step.

3.1.1 Why we don’t (yet) use Fruchterman-Reingold

The Fruchterman-Reingold (CITE) algorithm has been one of the most popular layout algorithms (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 is based on a sort of simulated annealing approach, so “memory” of previous positions is usually erased each time the layout is run², producing very unstable layouts.

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

3.1.3 MDSJ (Multidimensional Scaling for Java)

MDSJ is a very efficient implementation of “SMACOF” stress-optimization Multidimensional Scaling so `network.layout.animate.MDSJ` gives the best performance of any of the algorithms tested so far – despite the overhead of writing matrices out to a Java program and reading coordinates back in. It also produces very smooth layouts with less of the wobbling and flipping which can sometimes occur with Kamada-Kawai

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/>.

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).

²Various 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`.

3.1.5 Use a TEA attribute or write your own

The `useAttribute` layout is useful if you already know exactly where each vertex should be drawn at each time step (based on external data, like latitude and longitude), and you just want to render out the network³. It just needs to know the names of the dynamic attribute holding the x coordinate and the y coordinate for each time step.

TODO: include layout time comparison plots

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 might call this this “extracting” or “slicing”.

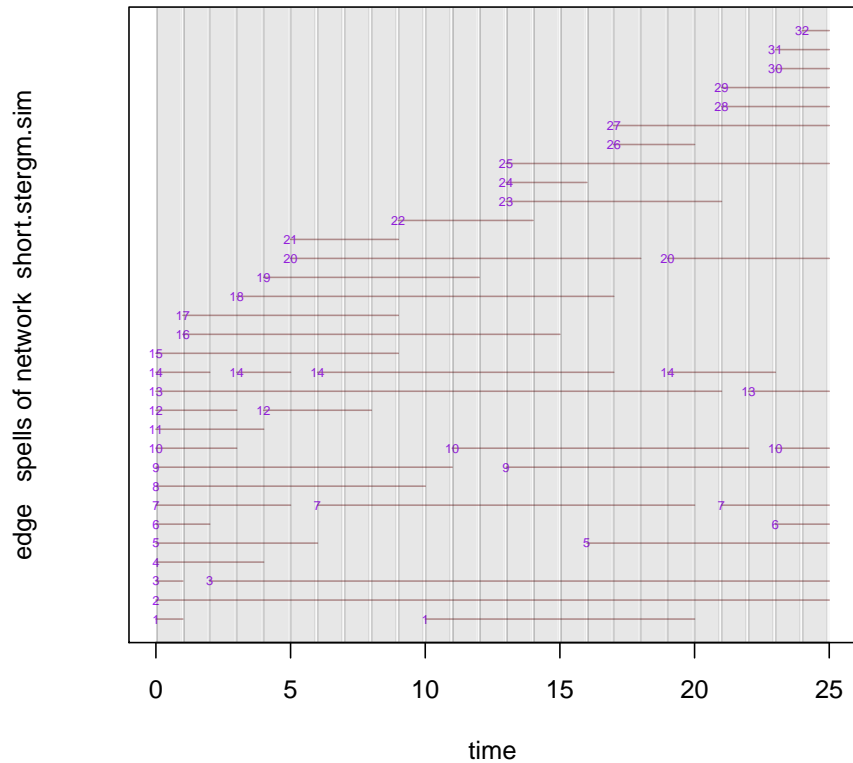
3.2.1 Slicing panel data

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

We can plot the slice bins against the timeline of edges to visualize our “observations” of the network. 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)
```

³It is also possible to write your own layout function and easily plug it in. See the `ndtv` package vignette for an example circular layout.



Looking at the first slice, which extends from time zero *until*⁴ 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
```

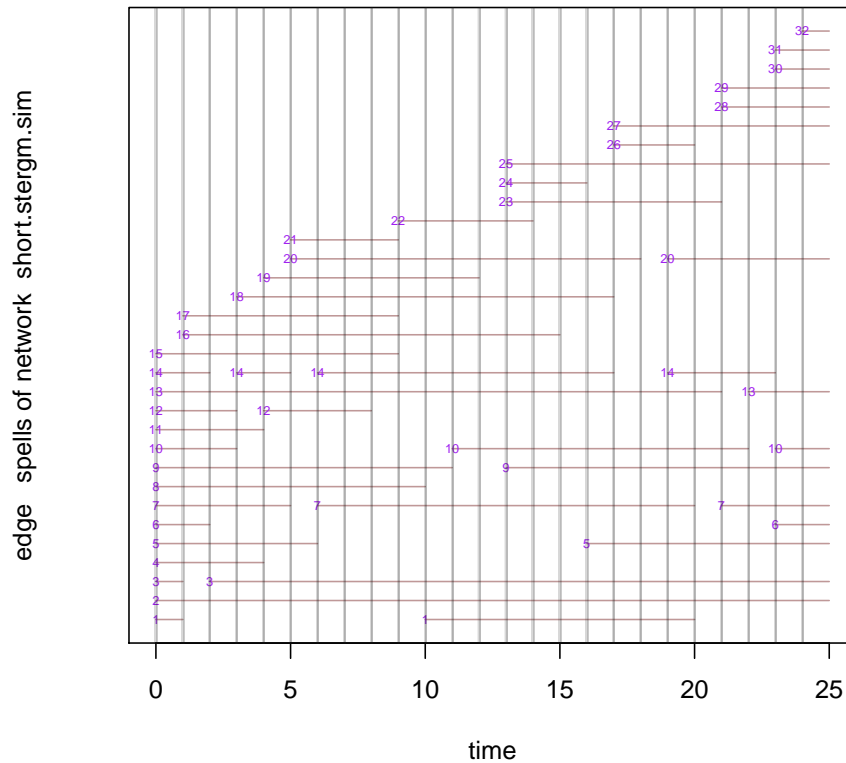
Because this is a discrete time network, the edge spells always extend the full duration of the slice, so it wouldn't actually matter if we used a shorter `aggregate.dur`. Each slice will still intersect with the same set of edges.

⁴notice 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 may 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.

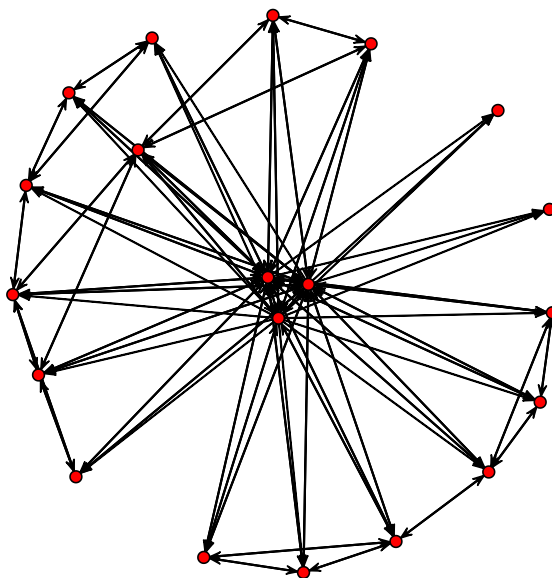
3.2.2 Slicing streaming data

Slicing up a dynamic network created from discrete panels may 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()

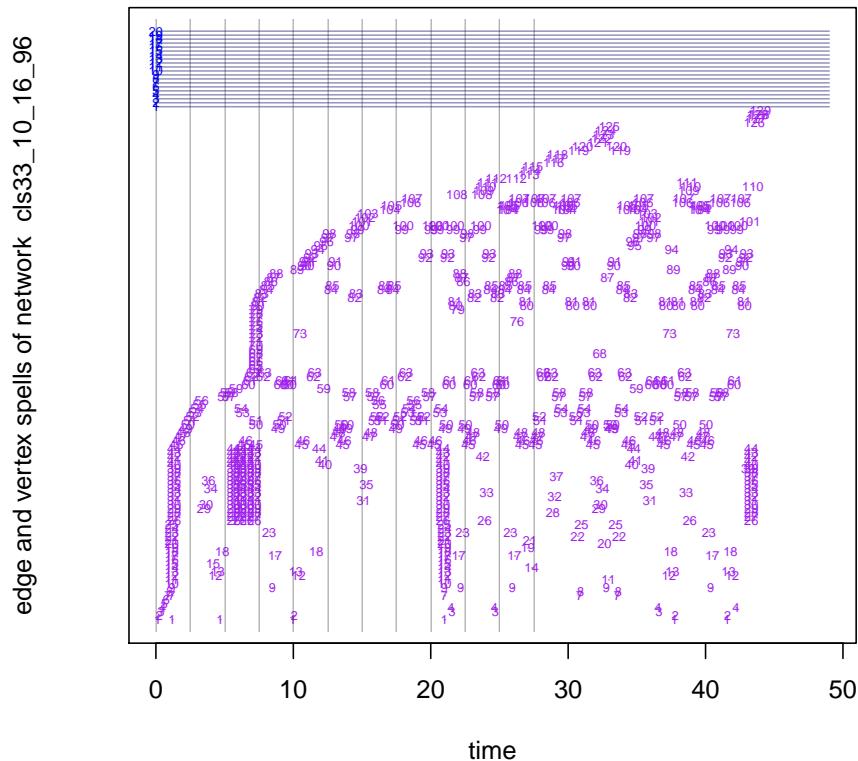
```

Notice that although the plot shows a start structure, in the animation most of the vertices are isolates, occasionally linked into brief pairs or stars by speech acts⁵. 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, 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)

```



⁵http://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 groups⁶.

```
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()
```

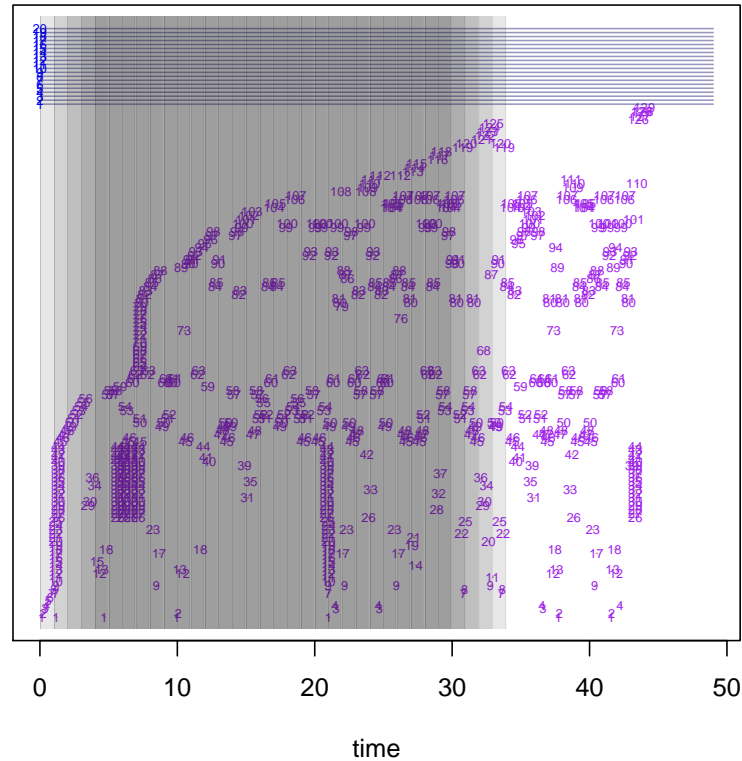
To reveal slower structural patterns we can make the aggregation period even longer, and let the slices overlap (by making `interval` less than `aggregate.dur`) so that the same edge may appear in sequential slices and the changes will be less dramatic between successive views⁷.

```
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()
```

⁶http://statnet.csde.washington.edu/movies/ndtv_vignette/cls33_10_16_96v2.mp4

⁷http://statnet.csde.washington.edu/movies/ndtv_vignette/cls33_10_16_96v3.mp4

edge and vertex spells of network cls33_10_16_96



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 different days, and there is even a day of missing data.

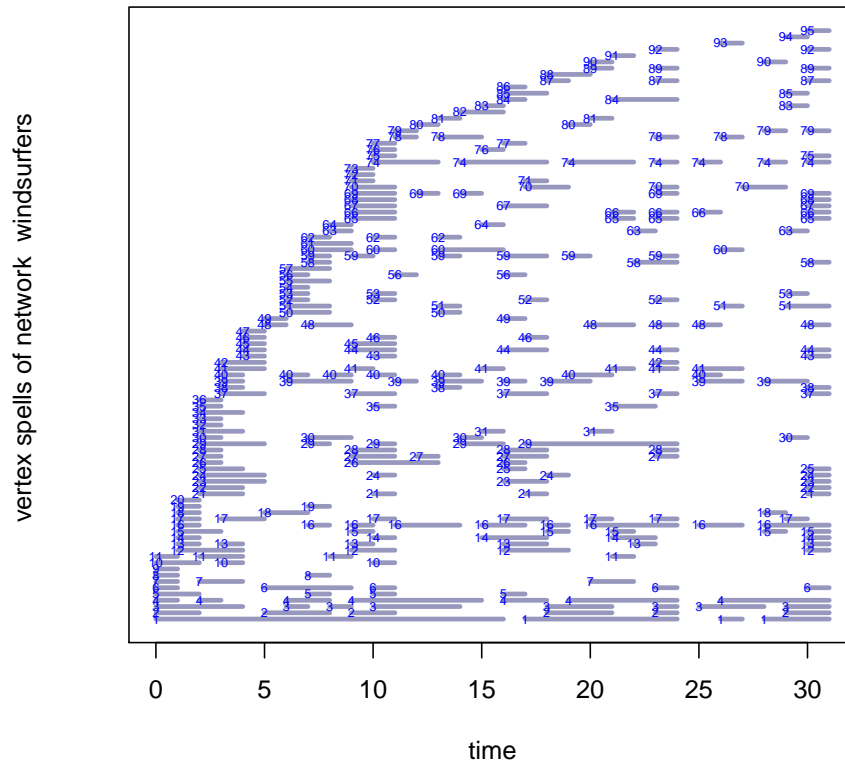
```
data(windsurfers)
slice.par<-list(start=1,end=31,interval=1,
               aggregate.dur=1,rule="latest")
windsurfers<-compute.animation(windsurfers,slice.par=slice.par,
                              default.dist=3,
                              animation.mode='MDSJ',
                              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 example⁸ 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 or the timeline plot below.

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

⁸http://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, let's try a week-long slice by setting `aggregate.dur=7` to try to smooth it out so we can see some structure.

```
slice.par<-list(start=0,end=24,interval=1,
                aggregate.dur=7,rule="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” network⁹ 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 is possible to see some of the sub-groups and the the various bridging individuals.

3.3 Animating graphic attributes

Vertices and edges are not the only things that change over time, how do we show dynamic attributes and changes to structural properties of the network?

3.3.1 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:

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

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

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

We can set values for vertex colors.

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

Finally we render it, giving the names of the dynamic attributes to be used to control the plotting parameters for edge width, 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”

⁹http://statnet.csde.washington.edu/movies/ndtv_vignette/windsurfers_v2.mp4

3.3.2 Functional plot arguments

Sometimes it is awkward or inefficient to pre-generate dynamic attribute values. Why create and 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 functions with special arguments to be evaluated on the fly at each time point as the network is rendered. So, for example, if we wanted to use our previously created “width” attribute to control the color of edges along with their width:

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

Notice the use of the `slice` argument to the function instead of the original name of the network. 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 is:

- `net` is the original (un-collapsed) network
- `slice` is the network collapsed to be rendered 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

So in the example above, at each time point the edge attribute “width” is extracted and used to control the red component of the rgb color. We can also define functions based on network measures such as betweenness:

```
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()
```

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

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

When we are looking at 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?

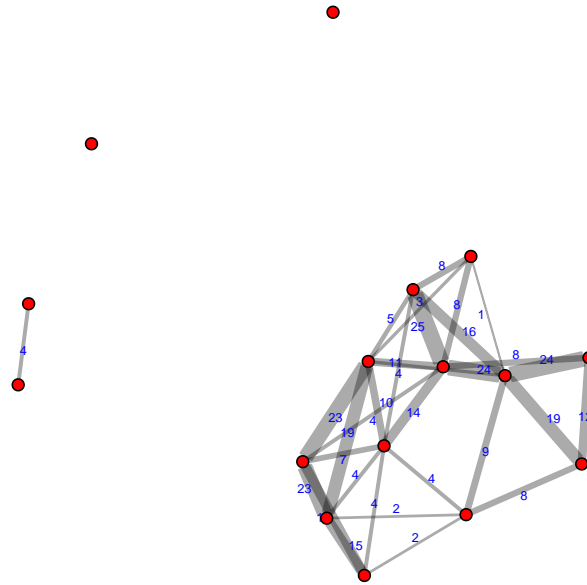
Collapse a network, and tell it not to remove the time related info

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

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

Notice the `activity.count` and `activity.duration` that are now attached to the edges. Lets plot it, using those durations to control the edge width.

```
plot(simAgg,edge.lwd='activity.duration',edge.col='#00000055',
  edge.label='activity.duration',edge.label.col='blue',
  edge.label.cex=0.5)
```



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.

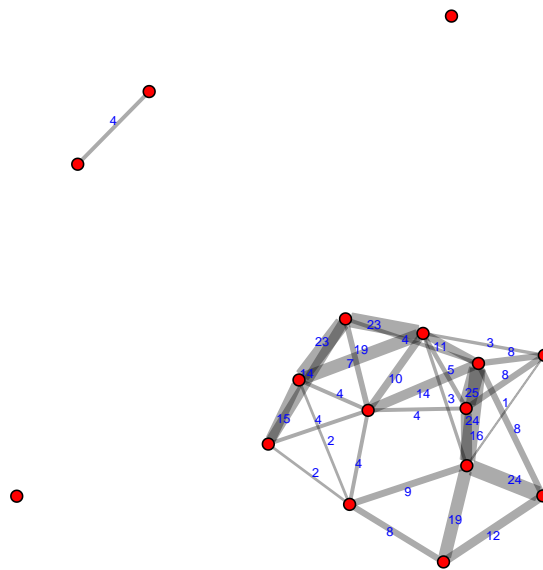
Lets do the plot again, still ignoring edge weights. But this time we are going to extract the computed coords and feed them into the plot (this is just for constructing the examples, don’t need to do it for real movies).

```
short.stergm.sim%%'slice.par'<-list(start=0,end=25,interval=25,
                                     aggregate.dur=25,rule='latest')
compute.animation(short.stergm.sim,animation.mode='MDSJ')
```

```
[1] "Calculating layout for network slice from time  0 to 25"
[1] "MDSJ starting stress: 138.6506618544938"
```

```
[2] "MDSJ ending stress: 10.902016815634658"
[1] "Calculating layout for network slice from time 25 to 50"
```

```
# 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)
```



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)
```

```

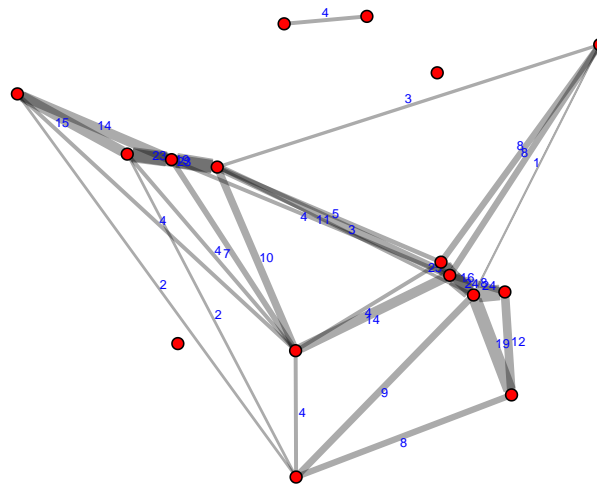
[1] "Calculating layout for network slice from time 0 to 25"
[1] "MDSJ starting stress: 176.33016382798766"
[2] "MDSJ ending stress: 13.674239842316556"
[1] "Calculating layout for network slice from time 25 to 50"

```

```

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)

```



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)

```

```

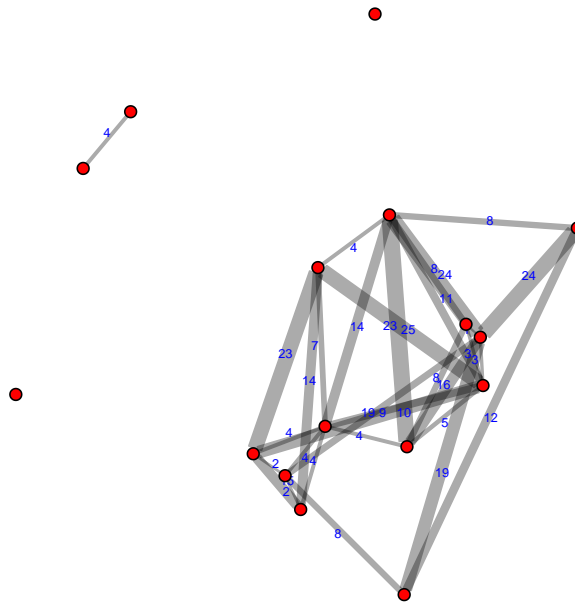
[1] "Calculating layout for network slice from time 0 to 25"
[1] "MDSJ starting stress: 1700.1229751494848"
[2] "MDSJ ending stress: 12.713427555384246"
[1] "Calculating layout for network slice from time 25 to 50"

```

```

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)

```



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
```

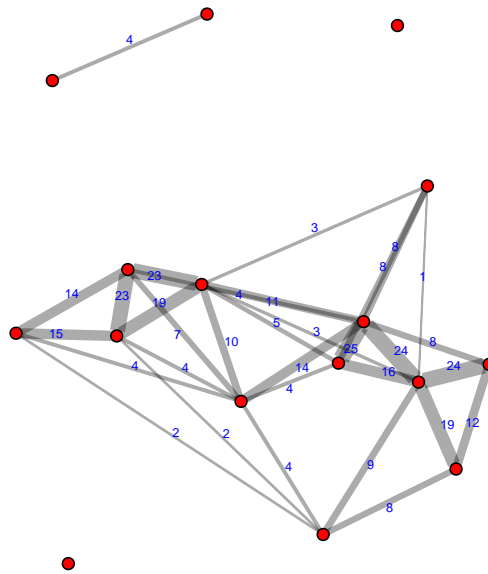
```
range(log(simAgg%e%'activity.duration'+1))
```

```
[1] 0.6931472 3.2580965
```

```
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)
```

```
[1] "Calculating layout for network slice from time 0 to 25"  
[1] "MDSJ starting stress: 5882.179561475249"  
[2] "MDSJ ending stress: 11.841858788837008"  
[1] "Calculating layout for network slice from time 25 to 50"
```

```
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)
```



Which gives us something better looking—big edges are a bit shorter, but not so much to squinch up the layout.

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

3.5 Adjusting isolates and components separation with `layout.distance`

We've briefly mentioned the `default.dist` parameter which can be tweaked to increase or decrease the spacing between isolates and disconnected components. The default value for `default.dist` is `sqrt(network.size(net))`, see `?layout.dist` for more information.

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)
```

```

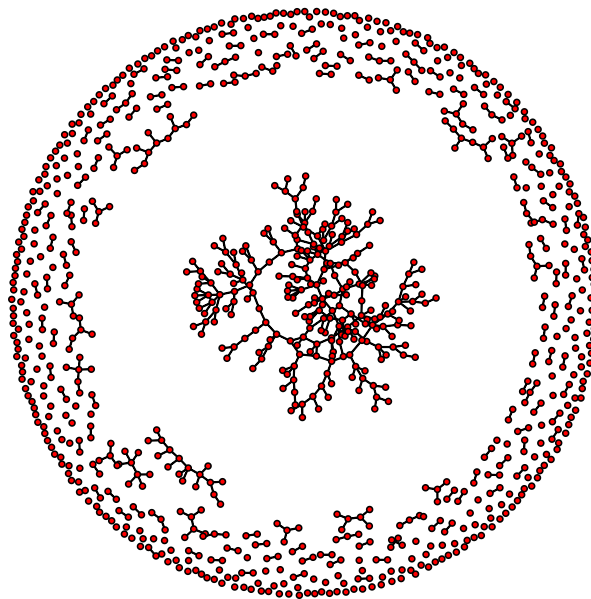
network.size(msmAt50)

[1] 1000

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

[1] "MDSJ starting stress: 960863.7568256531"
[2] "MDSJ ending stress: 151725.5346750167"

```



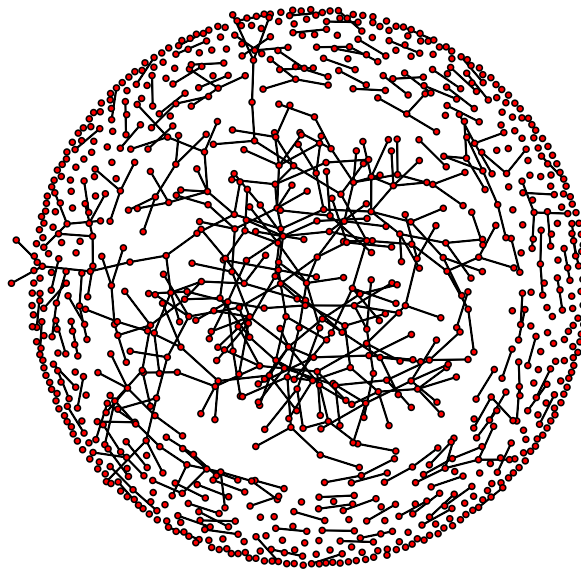
In this case, the default distance must have been set to about 31 (square root of 10000). This results in the giant component being well separated from the smaller components and isolates. Although this certainly focuses visual attention on the big component, it squishes up the rest of the network. We can set it smaller.

```

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

```

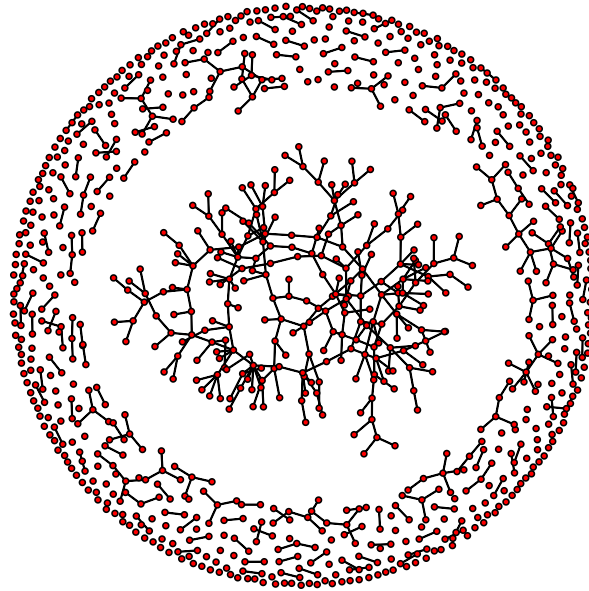
```
[1] "MDSJ starting stress: 896257.3990526946"  
[2] "MDSJ ending stress: 174094.7850881342"
```



But in the example above, the `default.dist` value was too small to effectively separate the components, resulting in a lot of unnecessary edge crossing.

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

```
[1] "MDSJ starting stress: 937097.082046701"  
[2] "MDSJ ending stress: 160423.64514629357"
```



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. (this is going to be slow..). 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.

```
#TODO: disabled to save time while writing doc
#compute.animation(msm.sim,animation.mode='MDSJ',default.dist=18)
#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 Transmission trees and assembled layouts

We will construct 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.

The code below defines a function to run the simulation, sets initial seeds (starts the rumor) and then runs the simulation ¹⁰. If you don’t care about the details, just execute the entire block of code.

```
# 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 (kowner in knowers){
      conversations<-get.edgeIDs.active(net,v=kowner,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",kowner,
                                     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)
          }
        }
      }
    }
  }
}
```

¹⁰The EpiModel package includes much better utilities for simulating transmission networks with various realistic properties

```

    }
    return(net)
}

```

We next need to load in the data with the edge dynamics, and 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)

```

So 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

```

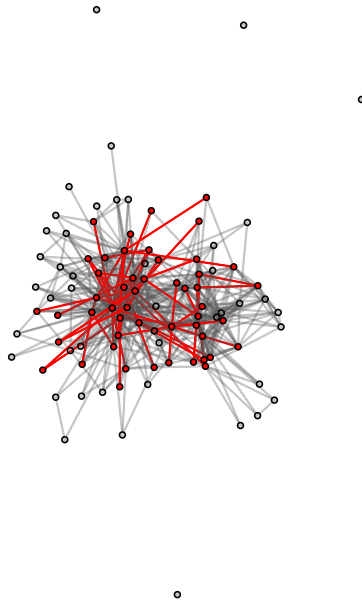
# plot the aggregate network, hiliting infected
plot(windsurfers,
     vertex.col=ifelse(
       !is.na(get.vertex.attribute.active(windsurfers,'knowsRumor',at=31)),
       'red','#55555555'
     ),
     edge.col=ifelse(
       get.edge.attribute.active(windsurfers,'passedRumor',at=31),

```

```

        'red','#55555555'
    ),
    vertex.cex=0.5
)

```



Since we know that the high rate of vertex dynamics makes it hard to see this as a movie, we can create a “flip-book” style movie, where we will keep the vertex positions fixed, and just animate the dynamics.

Question: If a network has a high-turnover vertex set, should it be considered to be a dynamic network?

```

# 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')

# render it
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 started at -31, so we are sliding along a giant bin, which is gradually accumulating more of the network edges with each step. 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.

In this view, it is still quite difficult to see the sequence of infections and the infection path. So lets try extracting that so that we can visualize it directly. The function below will create 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)
}

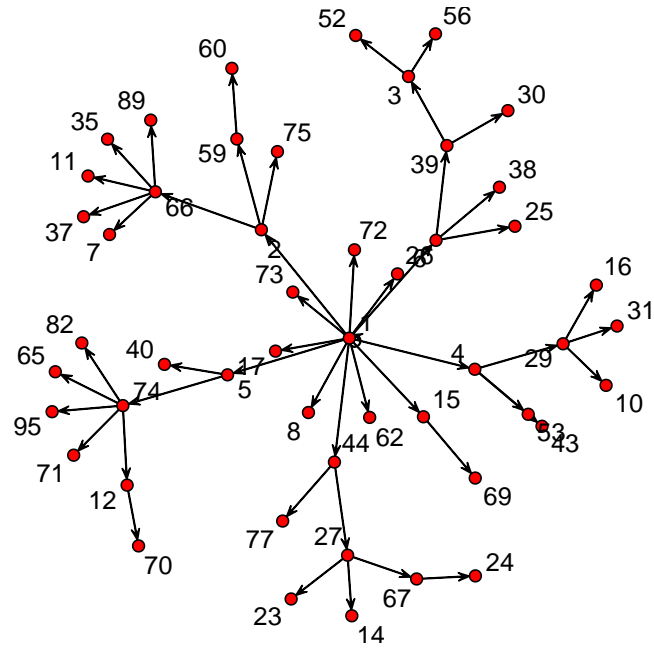
```

Now lets use the `transTree()` function to find the transmission tree, and plot it just for curiosity.

```

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

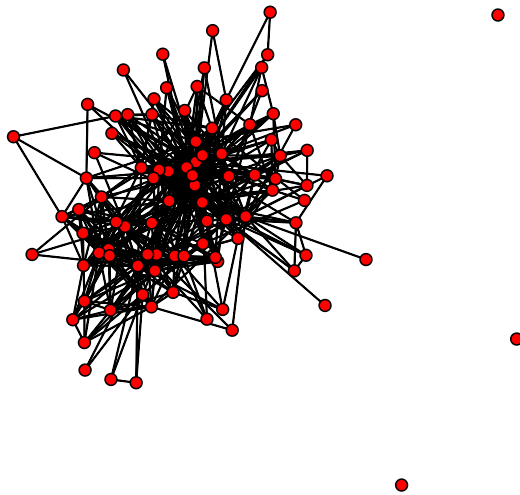
```



Now that we know how things work, we don't necessarily have to `compute.animation` to construct the sequence of coordinates. If we are careful, we can even use coordinates from one layout and apply them to another. In the next example, we will create an animated transition from the full cumulative network into a hierarchical representation of the transmission tree.

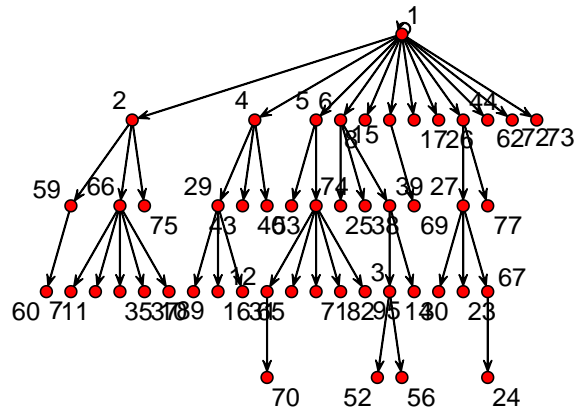
First, let's compute a layout for the cumulative across-time network

```
# 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` using the parts of other networks.

```
# FIRST FRAME: all vertices and edges active
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)
# 2ND FRAME: activate vertices that know and edges that passed it
activate.vertices(windAni,onset=1,terminus=3,
                  v=which(windAni%v%'knowsRumor'))
activate.edges(windAni,onset=1,terminus=3,
               e=which(windAni%e%'passedRumor'))
# THIRD FRAME: transition to tree
```

```

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)

4.2 A more complete tergm/stergm example

Expanding on the introductory example. Let say we've got a statistical model of a dynamic network defined as a STERGM, and we want to see what it looks like.

- Using statnet's `tergm` package to estimate the parameters for an edge formation and dissolution process which produces a network similar to the Florentine business network (`?ergm::flobusiness`) given as input.
- 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.
- See the `tergm` workshop and materials¹¹ for more background on the modeling process.

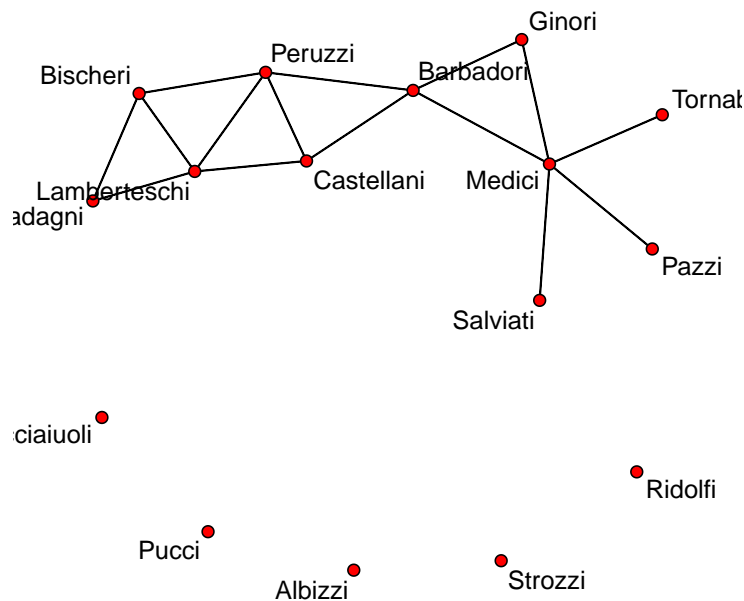
¹¹<https://statnet.csde.washington.edu/trac/wiki/Sunbelt2013#WorkshopMaterials>

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

```
library(ndtv)      # dynamic network animations
```

Load in the original Florentine business network.

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



Define basic **stergm** model with formation and dissolution parameters. The **tergm** package will do lots of complicated stuff to figure out an appropriate network model.

```
theta.diss <- log(9)
stergm.fit.1 <- stergm(flobusiness,
  formation= ~edges+gwesp(0,fixed=T),
```

```
dissolution = ~offset(edges),
targets="formation",
offset.coef.diss = theta.diss,
estimate = "EGMME" )
```

(time passes, lots simulation status output hidden)

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

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

CODE ABOVE DISABLED TO SAVE TIME WHILE WORKING ON DOC, CODE BELOW FAKES ITS OUTPUT

```
stergm.sim.1 <- short.stergm.sim
```

We define some parameters in a list named `render.par` in order to specify how the movie should be rendered. The `show.stats` parameter accepts a term summary formula to be evaluated, to print the model statistics for each slice on the appropriate frame of the movie.

```
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.

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

As we are fairly familiar with the output by now, we can suppress it by adding a `verbose=FALSE` argument.

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

4.3 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, unzipping in the process
hp1 <- read.table(unz(temp_hp.zip, "hpbook1.txt"),
                  sep=" ",stringsAsFactors=FALSE)
# is it really a matrix?
dim(hp1)
```

```
[1] 64 64
```

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

Notice the `stringsAsFactors=FALSE` argument to `read.table`. This prevents the strings from being converted into factors, which then may unexpectedly appears as integers causing all kinds of headaches. 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.

```
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 discrete
 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
```

```
# 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$name
# read in and assign the attributes
attributes<-read.table(unz(temp_hp.zip, "hpattributes.txt"),
                      sep="\t",stringsAsFactors=FALSE,header=TRUE)
harry_potter_support%v%'id'<-attributes$id
harry_potter_support%v%'schoolyear'<-attributes$schoolyear
harry_potter_support%v%'gender'<-attributes$gender
harry_potter_support%v%'house'<-attributes$house
# define a net.obs.period
harry_potter_support%n%'net.obs.period'<-list(
  observations=list(c(0,6)),mode="discrete",
  time.increment=1,time.unit="book volume")
# which vertex is Harry Potter?
which(network.vertex.names(harry_potter_support)=="Harry James Potter")
```

[1] 25

This 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="#CCCCC5"
)

```

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?

Exercise: Choose one of the dynamic dataset (perhaps one from the package `networkDynamicData` 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 compression¹², determining the right settings can be a trial and error process. Handbrake <http://handbrake.fr/> is an excellent and easy to use tool for doing video compression into the web-standard H.264 codec with appropriate presets.

5.2 Transparent colors

Using a bit of transparency can help a lot with readability for many visualizations. It makes it so that overlapping edges can show through and/or be less distracting. Many of the R plot devices support transparency, but specifying the color can be a bit awkward. The most concise way is to define colors as HTML color-code hex strings with an alpha channel in the format `"#RRGGBBAA"`. For example, 50% black is `"#00000088"`, 50% green would be `"#00FF0088"`. These can be hard to remember! It may be easier include an alpha (the computer graphics term for transparency) parameter to the `rgb()` function, but you still may need to guess the correct proportions of red, green, and blue. If you want to just make one of R's existing named colors more transparent, try `grDevices::adjustcolor()`.

```

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

```

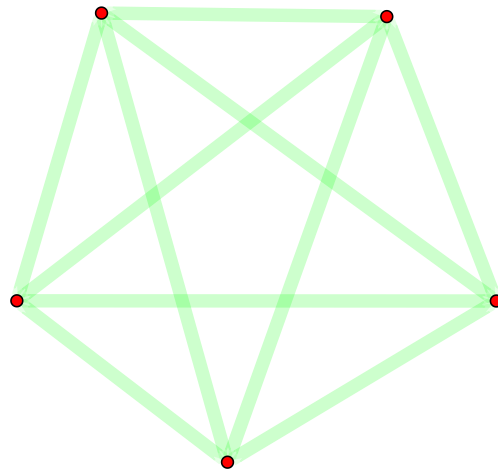
¹²The default settings for `ffmpeg` differ quite a bit depending on platform, some installations may give decent compression without tweaking the settings

```
[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))
```



5.3 How to get help

Here are some starting points if you reach a dead end:

- 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

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://statnet.org>.
- 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/>.
- Krivitsky P and Handcock M (2013). *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 3.1.3, 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.
- Padgett, John F. (1994) Marriage and Elite Structure in Renaissance Florence, 1282-1500. Paper delivered to the Social Science History Association.
- 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/>.