Using `sirgraph`

Barry Rowlingson

April 14, 2014

1 Graph data

First install and then attach the package.

```
library(sirgraph)
```

A graph object suitable for `sirgraph` is an `igraph` object with some extra attributes on the graph and the nodes. A sample can be created with the `makedata` function.

```
g = makedata()
g

## IGRAPH U--- 34 78 -- Zachary
## + attr: name (g/c), start (g/n), time (g/n), vaccinated (v/x), sex
##   (v/c), age (v/n), state (v/c)
```

The easiest way to get the node attributes is via the `get.data.frame` function.

```
head(get.data.frame(g, "vert"))

##   vaccinated sex age state
## 1      FALSE   M  19     S
## 2       TRUE   F  19     S
## 3      FALSE   F  19     S
## 4      FALSE   M  19     S
## 5      FALSE   M  19     S
## 6       TRUE   F  21     S
```

As well as extra node attributes, the graph itself has some extra attributes.

```
g$time

## [1] "2014-01-01"
```

```
g$start

## [1] "2014-01-01"

g$stepsize

## NULL
```

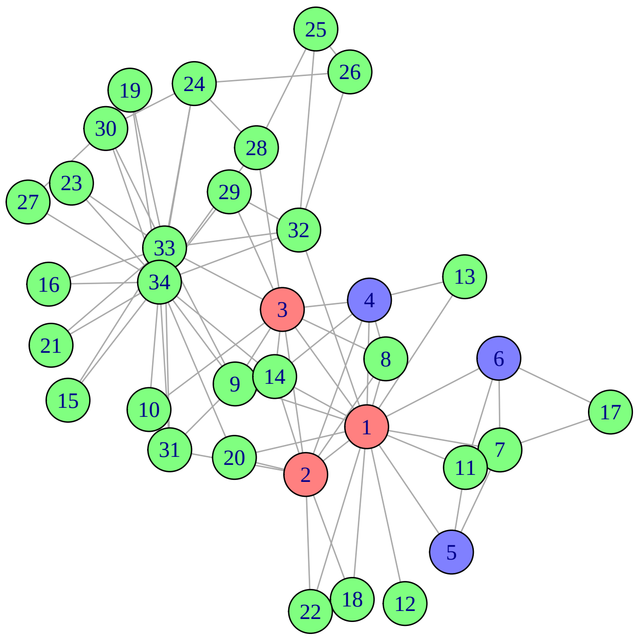
2 Plotting

First we'll set some infected and recovered nodes on the graph.

```
V(g)$state[1:3] = "I"
V(g)$state[4:6] = "R"
```

a simple plotting function can then be used:

```
plotSIR(g)
```



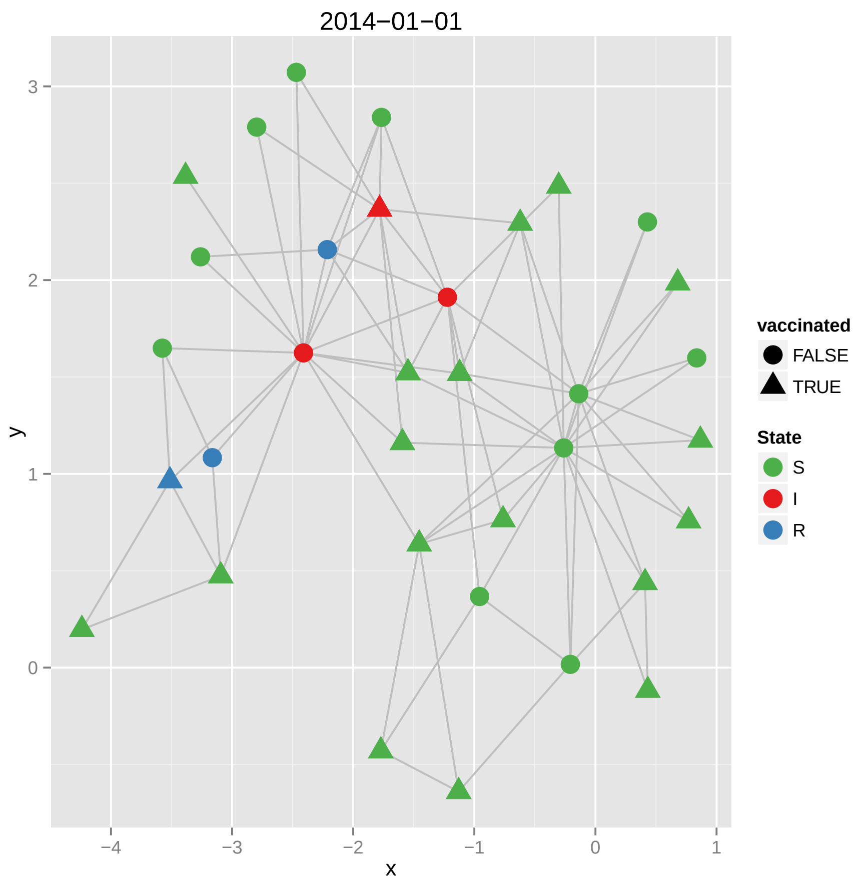
To use the ggplot2 graphics functions we first have to assign coordinates to the graph using the glayout function:

```
g = glayout(g)
```

By default this uses the Kamada-kawai layout, but any of the layout functions from igraph can be used.

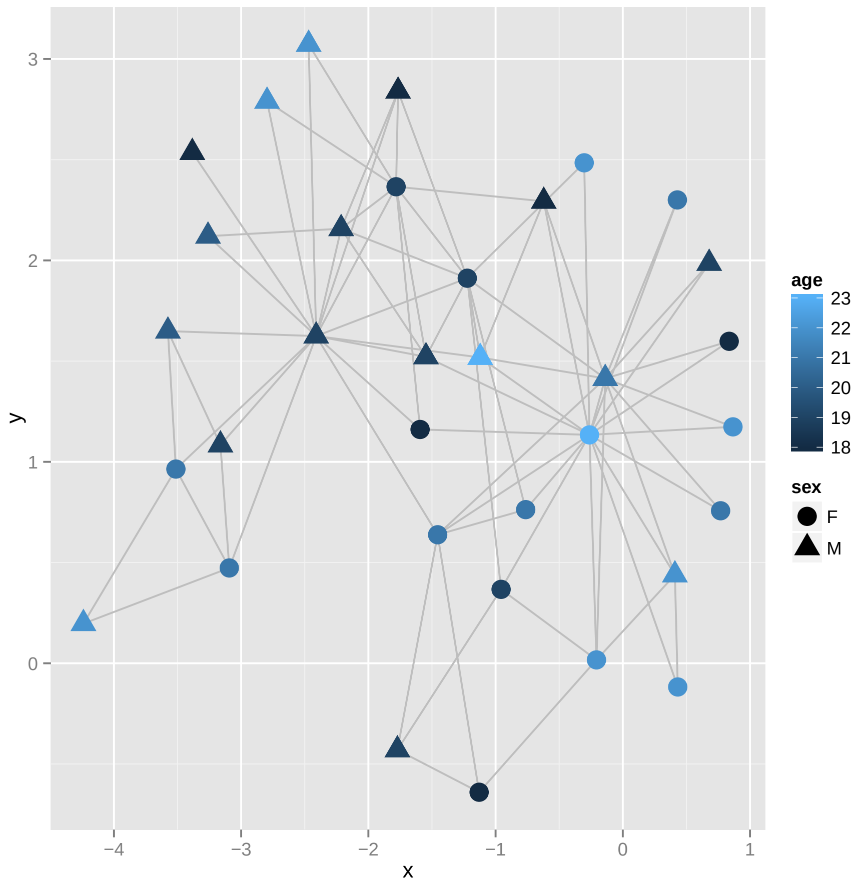
Now we can plot it:

```
gplotgraph(g)
```



This function uses the lower-level `geom` functions to plot state and vaccination status. You can use these lower-level functions to build other plots, for example to plot the network coloured by age and shaped by sex:

```
ggplot() + geom_edge(aes(x = x, y = y), col = "grey", data = g) + geom_node(aes(x = x,
    y = y, col = age, shape = sex), size = 5, data = g)
```



Full control of the plot is then available using the `ggplot2` functions, for example to change the colour scheme or shapes.

3 Discrete Time Infection Modelling

Running an SIR model on a graph network requires the following:

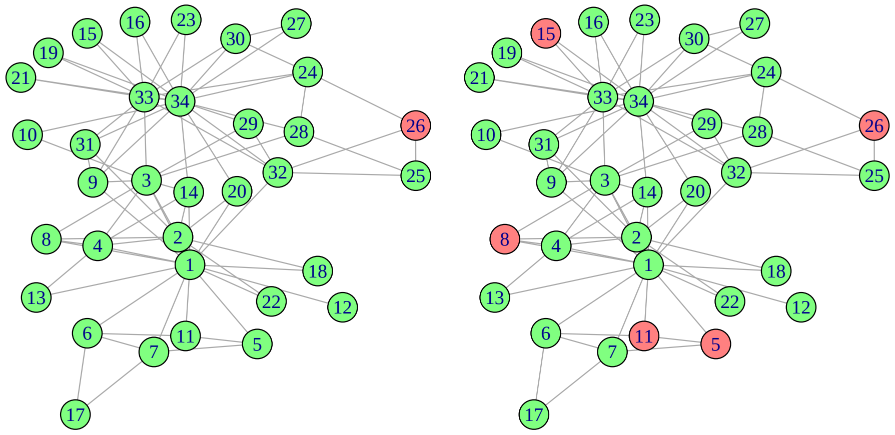
- A graph
- An initial state of each node (S, I, or R).
- A method for the spread of infection (S to I).
- A method for the removal of infection (I to R).
- A way of deciding when to stop the simulation

3.1 Starter Functions

A "starter" function is one which sets the initial infection state of a graph. Possible starter functions could be to set a fixed number of nodes as infectious, or to set infectious status with a fixed probability, or to set infectious status according to attributes of the nodes. A starter function always takes a graph as parameter, and returns the graph with a modified `state` attribute.

The `infectN` function is a starter function *generator*, in that it returns starter functions from a family of functions based on N, the number of initial infectious nodes. So `infectN(1)` is a starter function that infects one node, `infectN(5)` is a starter function that infects five nodes.

```
g = makedata()
g = glayout(g)
g = infectN(1)(g)
par(mfrow = c(1, 2))
par(mar = c(0, 0, 0, 0))
plotSIR(g)
g = infectN(5)(g)
plotSIR(g)
```



Exercise: write a starter function generator, `infectP`, perhaps, that starts the infection with each node having a probability, `p` of being infected.

3.2 Spreader Functions

A "spreader" function is responsible for stepping the infection along by a single time step. Currently it has the job of infecting new nodes and removing infected nodes (but this may be worth changing at some point).

As with a starter function, it takes an SIR graph as input and returns the updated graph. Function generators are again useful for creating spreader functions from parameterised families. We illustrate that now.

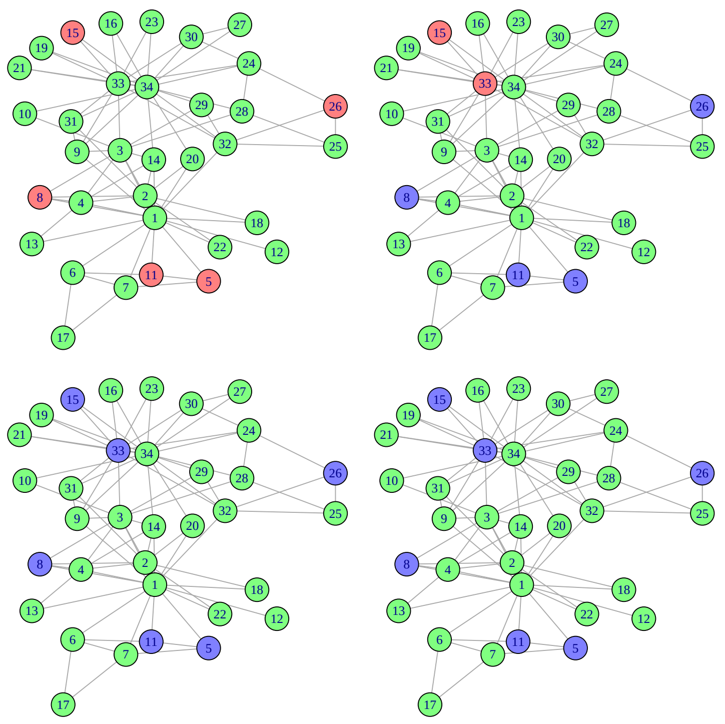
The `spreadP2` spreader function generator produces spreader functions based on two probabilities. The probability of a susceptible node being infected by an infectious node is `pSI` in any time step, and the probability of an infected node recovering is `pIR` in any time step. Hence a spreader function that gives each infectious possibility and recovery possibility an even chance is created thus:

```
sEvens = spreadP2(pSI = 0.5, pIR = 0.5)
sEvens

## SIR spread function
## Fixed spread probabilities, pSI=0.5 pIR=0.5
```

We can now run this spreader a few times and see the change.

```
par(mfrow = c(2, 2))
par(mar = c(0, 0, 0, 0))
plotSIR(g)
g = sEvens(g)
plotSIR(g)
g = sEvens(g)
plotSIR(g)
g = sEvens(g)
plotSIR(g)
```

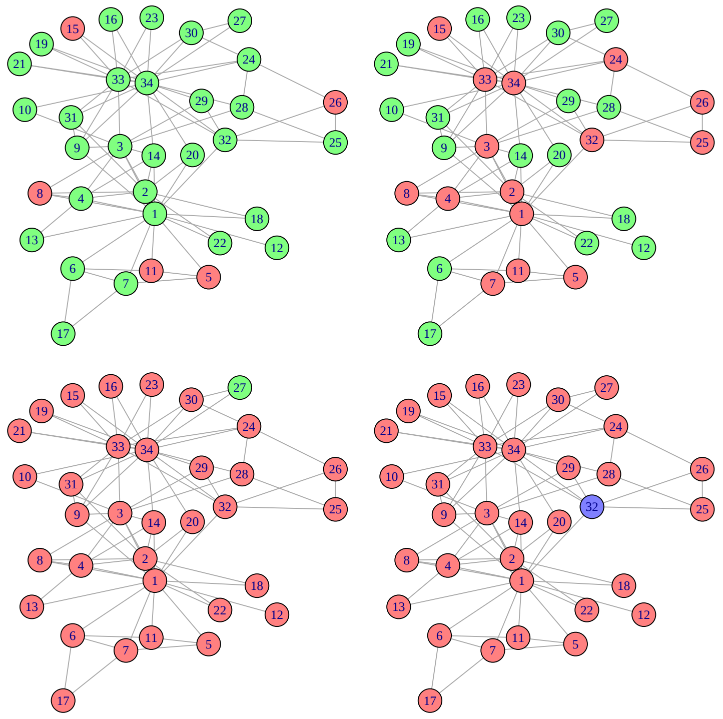


Now we repeat, but with a lower recovery probability and a higher infectious probability:

```
sMore = spreadP2(pSI = 0.8, pIR = 0.1)
sMore

## SIR spread function
## Fixed spread probabilities, pSI=0.8 pIR=0.1

g = infectN(5)(g)
par(mfrow = c(2, 2))
par(mar = c(0, 0, 0, 0))
plotSIR(g)
g = sMore(g)
plotSIR(g)
g = sMore(g)
plotSIR(g)
g = sMore(g)
plotSIR(g)
```



```
g$time

## [1] "2014-01-04"
```

Notice how the current time has changed.

The `spreadP2` function is a special case of a more general spreader process where the probabilities are a function of time and the attributes of the potentially infected person. This is handled by the `spreadF` function. By writing any function of time and vertex attributes, it is possible to simulate other processes in this class. These functions must be vectorised over the vertices so that when called with a number of vertices they return the same number of probabilities.

For example, this function returns 0.9 on any day apart from Sunday, when it returns 0.5 – it is independent of the vertex attributes.

```
restDay <- function(t, v) {
    nv = length(v)
    if (wday(t) == 1) {
        return(rep(0.5, nv))
    } else {
        return(rep(0.9, nv))
    }
}
restDay("2014-04-06", 0)

## [1] 0.5

restDay("2014-04-07", 0)

## [1] 0.9
```

Further, this function returns 0.9 if the vertex is an unvaccinated vertex, and 0.2 if it is vaccinated. This function ignores the time, and note it is vectorised over the vertices by using the `ifelse` function:

```
isVaccinated <- function(t, v) {
    ifelse(v$vaccinated, 0.2, 0.9)
}
# see the first few
isVaccinated("2014-01-01", V(g))[1:8]

## [1] 0.9 0.2 0.9 0.9 0.9 0.2 0.2 0.9
```

Now we can use either of these to build a spreader function using `spreadF`, along with a constant function for the recovery probability. We'll use a tree network to see how vaccinated nodes isolate branches of the tree.

```
fConst = function(t, v) {
    rep(0.3, length(v))
}
spreadVaccine = spreadF(isVaccinated, fConst)
spreadVaccine

## SIR spread function
## Spread probabilities as functions
```

3.3 Stopper Functions

A “stopper” function takes an SIR graph and returns TRUE if it decides you should stop running a simulation. Possible stopper functions include stopping after a given time, or stopping if there are no infectious cases. Those two stopper functions are currently implemented.

Stopper functions aren’t much use on their own, but are designed to be passed into the main looping function `stepSim`.

4 Running Simulations

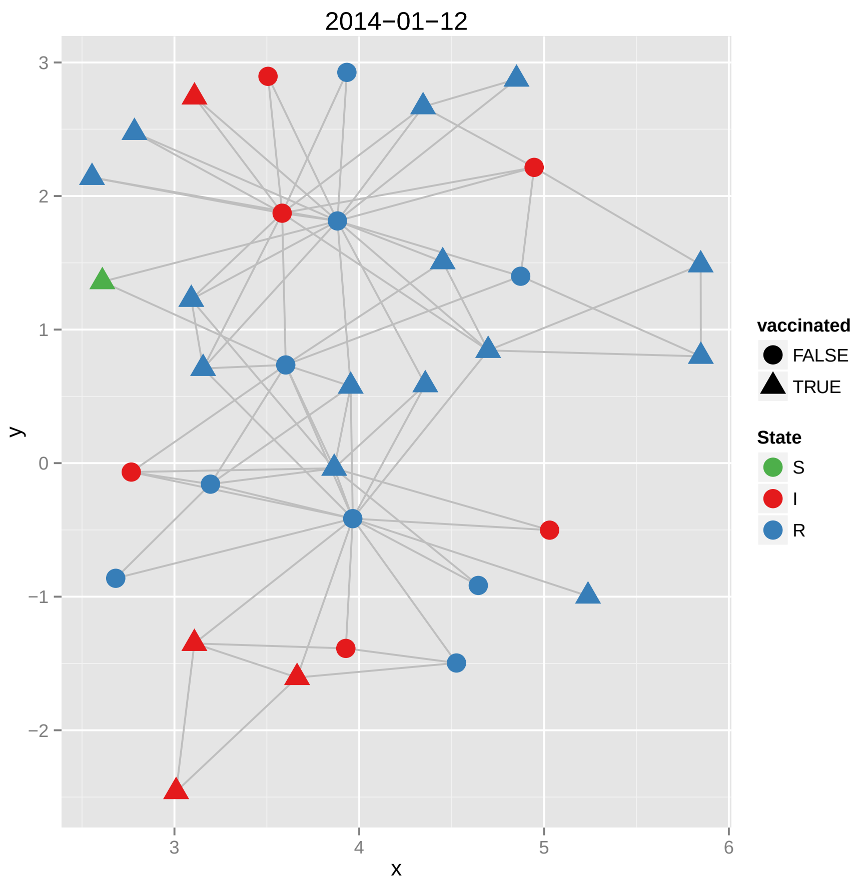
The main simulation looping function, `stepSim` takes a graph, a a spreader function, and a stopper function, and runs the spreader until the stopper indicates it has finished. The graph is returned in its final state with infection and removal times as node attributes.

The default start time in `makedata` is 2014-01-01, so here we’ll run for 12 days:

```
g = makedata()
g = glayout(g)
g = infectN(1)(g)
g = stepSim(g, sMore, stopAfter("2014-01-12"))
g$time

## [1] "2014-01-12"

gplotgraph(g)
```

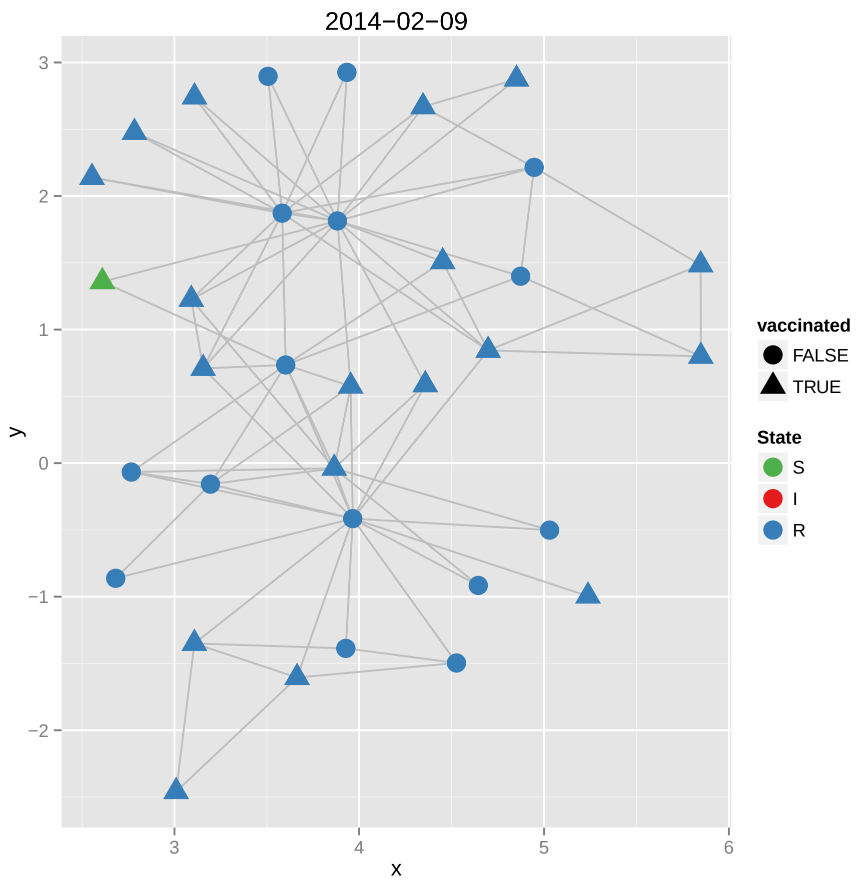


The infection is clearly still going at that point. Now we'll start again and let it run until it has finished and there are no infectious nodes.

```
g = makedata()
g = glayout(g)
g = infectN(1)(g)
g = stepSim(g, sMore, stopWhenClear)
g$time

## [1] "2014-02-09"

gplotgraph(g)
```



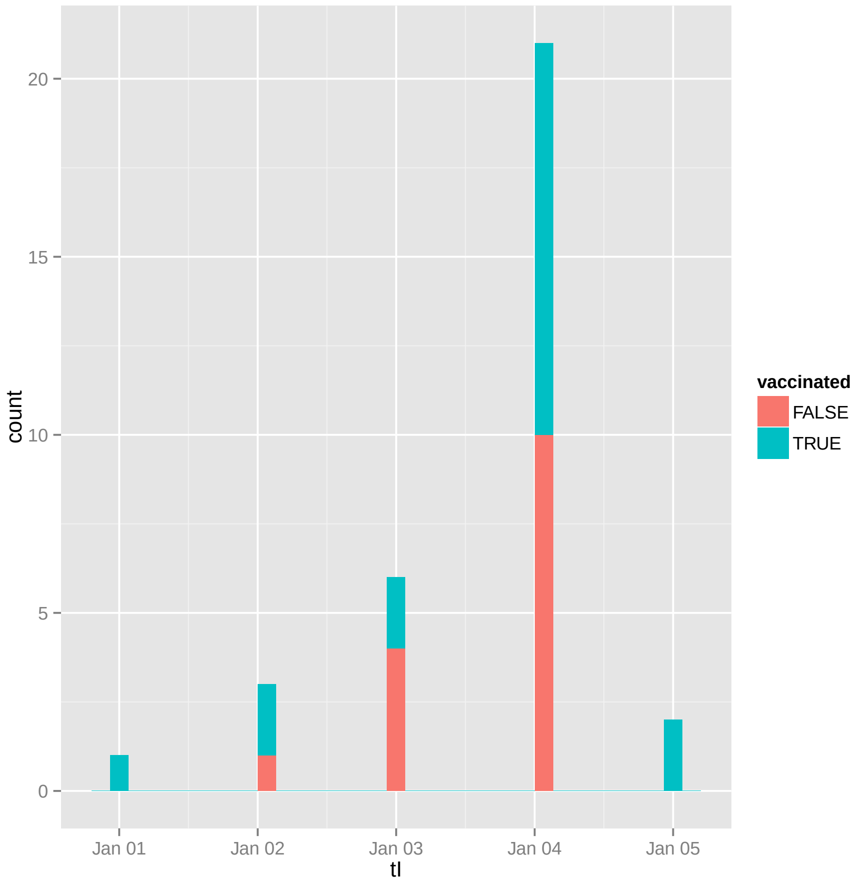
The timing of the state changes can be obtained by getting the vertex data frame. Note that dates aren't kept as dates and so need converting from the number of days from the start of 1970. Then we can, for example, plot a histogram of the infection times, broken down by vaccination status (which in this case is irrelevant):

```
d = get.data.frame(g, "vertices")
d$tI = as.Date(d$tI, "1970-01-01")
d$tR = as.Date(d$tR, "1970-01-01")
head(d)

##   vaccinated sex age state     x       y         tI         tR
## 1      FALSE   M  19     R 3.965 -0.4153 2014-01-03 2014-01-08
## 2       TRUE   F  19     R 3.865 -0.0386 2014-01-04 2014-01-12
## 3      FALSE   F  19     R 3.602  0.7354 2014-01-04 2014-01-06
## 4      FALSE   M  19     R 3.195 -0.1572 2014-01-04 2014-01-10
## 5      FALSE   M  19     R 4.526 -1.4961 2014-01-04 2014-01-12
## 6       TRUE   F  21     R 3.108 -1.3501 2014-01-04 2014-01-13
```

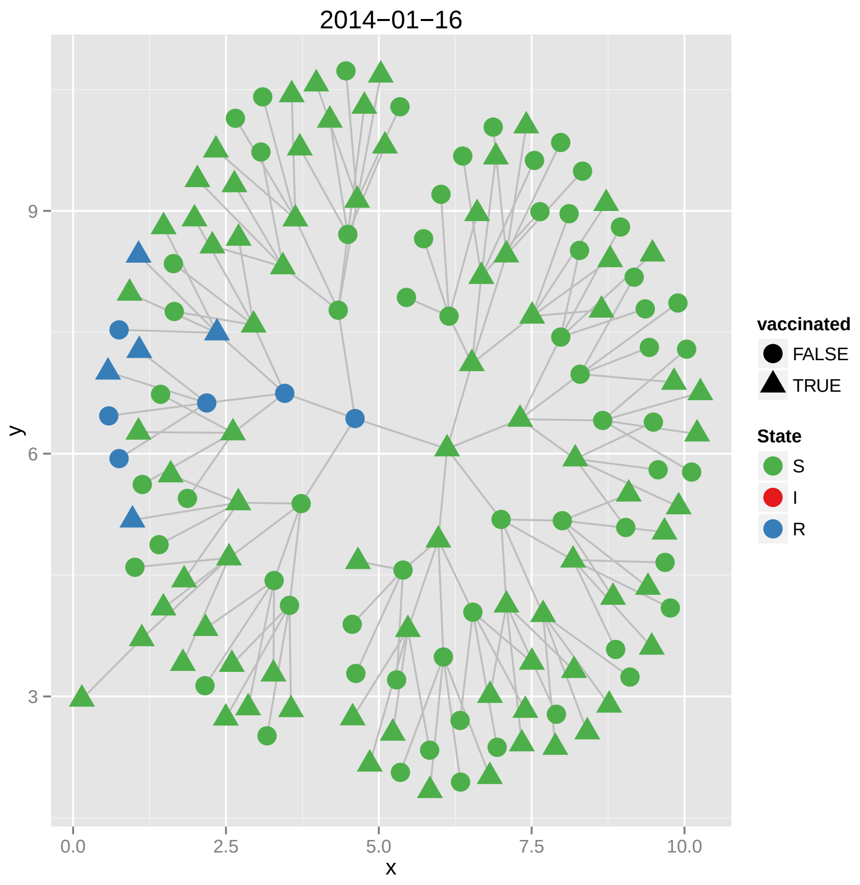
```
ggplot(d, aes(x = tI, fill = vaccinated)) + geom_histogram()
```

```
## stat_bin:  binwidth defaulted to range/30.  Use 'binwidth = x' to
adjust this.
## Warning:  position_stack requires constant width:  output may be
incorrect
```



We now try running the simulation using the vaccine-dependent infection model defined as `spreadVaccine` previously. We'll use a tree structure for our network to show the effect of vaccinated nodes protecting branches.

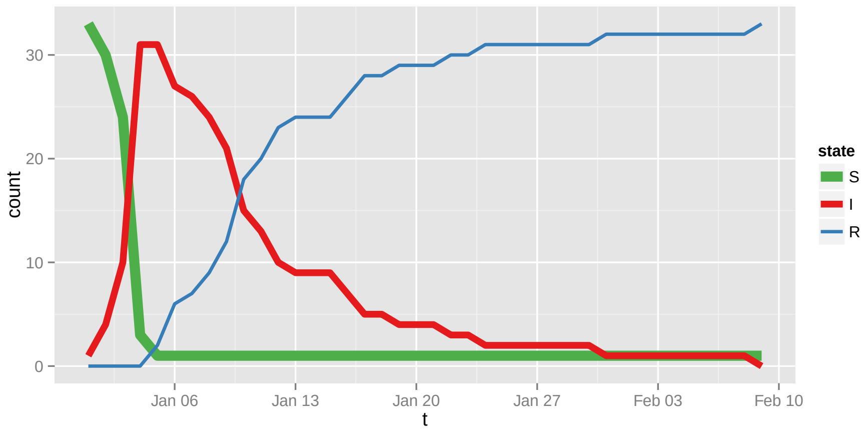
```
gF = infectN(2)(glayout(makedata(g = graph.tree(150, children = 4))))
gF = stepSim(gF, spreadVaccine, stopWhenClear)
gplotgraph(gF)
```



5 Plotting in Time

The resulting epidemic progress can be plotted:

```
timePlot(g, s = 1)
```



Or you can use the `animation` package to produce a web page:

```
saveHTML(
    {
      print(gplotgraph(g));
      g = stepSim(g,
                  sMore,
                  stopWhenClear,
                  after=function(g){
                      print(gplotgraph(g))
                    })
    },
    outdir="./spread/"
    )
```

Note how the graph must be printed at the start and via the `after` function in the stepping loop. This makes sure we see the initial state of the network and the final state.

6 Design

This code has been written with a fairly strong functional program methodology, hence the sprinkling of function generators here and there. The design of the `stepSim` function has been influenced by the idea of *dependency injection*, so that the function code represents the minimal amount of code needed for any generic simulation, and specific behaviour is passed in via spreader and stopper functions.

This design also means that spreader and stopper functions can be easily run and hence tested outside any looping code.

7 To-do

Some ideas:

- make lots of things into classes so the print nicely
- separate the infectious and removal aspects of the spreader function