Math. Model. Nat. Phenom.
Vol. 6, No. 6, 2011, pp. 61-75
DOI: 10.1051/mmnp/20116604

Using R to Build and Assess Network Models in Biology

G. Hartvigsen *

Department of Biology, SUNY Geneseo, Geneseo, NY 14454 USA

Abstract. In this paper we build and analyze networks using the statistical and programming environment R and the igraph package. We investigate random, small-world, and scale-free networks and test a standard problem of connectivity on a random graph. We then develop a method to study how vaccination can alter the structure of a disease transmission network. We also discuss a variety of other uses for networks in biology.

Key words: networks, graph theory, R, igraph, modeling disease dynamics
AMS subject classification: 92-01

1. Introduction

Many, perhaps all, biological systems are governed by interactions among individual agents, from interacting genes to interacting organisms that constitute ecosystems. A useful construct for understanding these interactions comes from graph theory, or the division of mathematics that deals with vertices and edges. In biological systems virtually any individual object can be thought of as a vertex and the interactions it has with other like objects can be thought of as edges that connect these vertices. Once we frame our system of interest in terms of such a network we can build models of our systems and quantify various aspects of the structure of such systems.

This area of mathematics was born in a paper by Euler in 1741 referred to as the "Konigsberg Bridge problem" [6]. In that problem Euler solved the problem of whether a pedestrian could traverse all seven bridges in the town only once and return to their starting point. Biologists are using graph theory to investigate problems ranging from the structure and function of gene and protein networks, evolutionary relationships using phylogenetic trees, to the interactions of mul-

*Corresponding author. E-mail: hartvig@geneseo.edu

tiple species in food webs. In this paper we will build and analyze networks and investigate their structure using an epidemiological framework. We are at the early stages of uniting the fields of biology and graph theory and, with the advent of fast computers, new software tools, and the rapidly growing amounts of data, the future is likely to rapidly reveal new insights into complex biological systems. The approach we take and tools we develop and use can easily be used to study most other biological systems. For additional reading on graph theory readers are encouraged to see this introductory graph theory book by Chartrand [2] and the excellent new text by Newman [10].

Historically mathematical biology has been dominated by calculus-based solutions. Early work by mathematicians interested in population dynamics built upon continuous growth models (*e.g.*, exponential and logistic growth) and systems of differential equations found in competition and predator-prey models; staples still introduced in today's ecology textbooks. At about the same time that these models were being developed the important field of mathematical epidemiology emerged, primarily with the publication of the 1927 paper by Kermack and McKendrick [8]. The study of disease dynamics has been particularly amenable to mathematical treatments which involve the interactions between hosts and their disease-causing agents. In the Kermack and McKendrick [8] paper the authors defined what has come to be known as the family of *S.I.R.* models; so named for the three interdependent populations of **S**usceptible, **I**nfectious, and **R**ecovered individuals.

More recently, realistically-structured, discrete population models have been used to ask questions that are more difficult, if not impossible, to address with techniques such as differential equations. The use of discrete networks, or graphs, allows us to directly assess disease dynamics relatively easily, if we have access to the proper tool set. The goal of this paper is to gently introduce the reader to an environment that allows for the construction and analysis of a wide variety of networks. We will use the freely available statistical and programming environment `R` to build a variety of graphs and analyze their structures. In our journey we will work to verify a result from the beginning work on graph theory and show how that result can provide insight into how we might assess the susceptibility of a population to the spread of a disease agent.

Getting Up and Running with `R`

There is a rapidly growing interest in the `R` statistical and programming environment (*e.g.*, see the number of books coming out over the last 10 years). `R` is a high-level programming language, so called because many simple commands do a *lot* of behind-the-scenes work. The program is powerful (many features), works on a variety of computer operating systems (e.g., Mac, Linux, and Windows), and is free. Perhaps `R`'s greatest strength is that it is extensible through the incorporation of packages, which are add-ons that provide the user with additional tools (e.g., solving systems of differential equations, tessellating a plane, or creating high-quality, fully-rendered 3-D landscape images that a user can rotate with their mouse). The reader wanting to learn more about the basics of `R` is encouraged to read the manual included with your `R` installation (*An Introduction to R*) or seek out one of the many introductory books, such as Crawley's "The `R` Book" [3], Dalgaard's "Introductory Statistics with R" [4], and Zuur et al. "A Beginner's Guide to R" [14].

You must first install the R program on your computer. To install R go to the R Project for Statistical Computing website (http://www.r-project.org) and simply follow the directions under "Getting Started."

Once you have installed and opened R we can begin. In the console you can issue commands directly at the command line (after the > character). To get started we can use R as an incredibly powerful calculator. I find that for many students this becomes the first time they actually use their computers to compute. You can, for example, take the square root of the natural log of 7 with this command:

```
> sqrt(log(7))
```

R works like most other programming languages. What happens in the above command is that the number 7 is passed to the log() function (natural logarithm). The answer is then returned. But in the example above that answer is then sent to the sqrt() function and then that answer comes back. R has no where else to put the answer but to the screen. So you should then see the following:

```
[1] 1.394959
```

The [1] means that the return value for the command has only a single value, which is followed by the answer. Sometimes you will get an array with lots of values for which R will keep a count. Alternatively, we could have stored the answer in a variable for later use, like this:

```
> my.answer = sqrt(log(7))
```

Then R would not have written the answer to the screen. If you wanted to see the answer you could type the following at the command line and hit <enter>:

```
> my.answer
```

Working at the command line is productive when using R as a calculator but for most projects you should record your commands in a script file. This makes returning to a project which usually will have a dozen or more commands very easy in R. With the console window active click on the File menu and open a new script file (Windows) or document (Mac) and save this file so you can find it again. The commands you write (or copy from this paper) should be placed in your script file. To run lines of code from a script file you need to *send* them to the console (that window with the > character). To do this you highlight the code and then press the following key combination: <ctrl> r in Windows or <cmd> <enter> on a Mac. You should try the above command (sqrt(log(7)) and see if you can get it to work from your script file.

The last piece of general instruction for working in R answers the question: "How do I get help?" There are many ways. If you know the function name but cannot remember how exactly to use it then, at the command line prompt, type:

```
> ?function
```

If, for instance, you are unsure how to use the function `sqrt()` you can type the name of the function at the console, followed by `<enter>`:

```
> ?sqrt
```

You should then be presented a help page within `R` or the help page will be opened in your web-browser. If you don't know the exact name of the function but know a key word you can search more broadly with this command:

```
> ??mean
```

This returns a list of quite a few routines, one of which you will find to be `base::mean` and see that this is the "Arithmetic Mean." This tells us that in the base package, which is always loaded, there is a function called `mean()`. You can calculate the arithmetic mean of the five numbers from $1 \rightarrow 5$ by issuing this command:

```
> mean(c(1,2,3,4,5))
```

The function `c()` combines the five numbers, separated with commas, into an array. That array of numbers is then sent to the `mean()` function which adds them up and devides by the length of the array (5). The answer is returned to the console:

```
> [1] 3
```

Control keywords are not functions and help for these is found by placing the word in quotes. To learn more about "for" loops, for instance, you can type this command:

```
> ?"for"
```

There also are many online sources with helpful information. If you want to start with the basics type `help.start()` followed by `<enter>` at the command line. Your browser should open a webpage found on your computer that provides useful links, including manuals and additional supplementary materials. Often when I have a problem I simply use an online search engine. For instance, if I am not sure how to do something, perhaps conduct a linear regression analysis, I would search "r linear regression." I just did this and the first hit was a 9-page manual titled "Using R for Linear Regression." Help is never far when you are using `R`!

There are many hundreds of packages that serve to extend `R`. To use these you only need to install them once on your computer and then, at the start of each `R` session, you will need to load the package in order to have its functions available for use. There are many packages that provide networking tools. The packages have some overlap and readers are encouraged to consider which

of these packages is best for their own needs (e.g., `igraph`, `sna`, `diagram`, and `dynamic graphs`, to name a few). For our purpose we will use the `igraph` package which offers a wide range of routines for building and analyzing networks. You can install `igraph` when you are connected to the internet and type this command:

```
> install.packages("igraph")
```

You will be asked to choose a mirror server from which to download the package. You should choose one that is geographically close to you. Note that you do need internet access to install packages this way. You can download packages as "zip" files when online and then install these on any number of offline computers.

Building Networks in `R` Using `igraph`

The `igraph` package provides many functions that build a variety of popular networks and allow you to relatively easily manipulate and analyze their structure. After developing a few of these networks we will investigate how removing vertices, functionally similar to vaccinating a host, alters the structure of a network.

To work with an add-on package it must be installed *and* loaded. Assuming you successfully installed `igraph` load it with this command:

```
> library(igraph)
```

We now have at our disposal all the functions available in the `igraph` package. We can construct a variety of common, built-in networks and we also can build any graph we want by simply creating vertices and adding appropriate edges. We also can read in an existing graph if it is in one of several standard graph formats (*e.g.*, an adjacency matrix, an edgelist, or a file in the Pajek format). This latter technique is more advanced and requires that the graph data file be located in the active working directory.

We will first begin with the construction of a random graph, followed by the popular Watts-Strogatz "small–world" network [13] and then the Barabási scale-free network [1] (preferential attachment). For each we will assess their structure and discuss the benefits of each type for applications in epidemiology.

The Random Graph

A random graph is constructed by taking a set of vertices and connecting each pair with a probability P. Paul Erdös and Albert Rényi [5] worked on this graph and the primary author of that paper has become the root of a very famous network, referred to as the "Erdös Number Project". Earlier, however, Solomonoff and Rapaport [12] showed that as edges are randomly added to a graph a threshold is crossed where a large component emerges. As more edges are added to a graph

an increasing number of vertices become connected. Groups of connected vertices that cannot be reached by other vertices are called "components". As more edges are added the components become connected and more vertices join the largest component, called the "giant component". As $P \rightarrow 1.0$ the graph will contain fewer components and fewer isolated vertices. A "complete graph" occurs when $P = 1.0$ (where every possible edge exists).

Solomonff and Rapaport [12] found that when the number of edges equals the number of vertices in a random graph then approximately 80% of the vertices will be contained in the giant component. This is an important problem to consider because the giant component can represent the maximum number of people who could get sick if just one person becomes infected and the probability of spread is 1.0 (worst case scenario!). Keep in mind this important interpretation of the giant component in epidemiology. We can verify this result with just a few lines of code.

Let us investigate how the size of the giant component changes as we add more and more edges to a graph. We will, therefore, create a series of networks with a fixed number of vertices and an increasing number of edges ($0 \leq N_{edges} \leq 2 \cdot N_{vertices}$) and determine the size of the giant component of each network. If we graph the proportion of the vertices in the giant component against $N_{edges}/N_{vertices}$ we should expect to see this relationship start out low and increase, passing through the point (1, 0.8). We can use the following command to construct a single random graph with just 25 vertices and the probability of an edge $P = 0.1$.

```
> g = random.graph.game(25, p.or.m = 0.1, type = "gnp")
```

We can then plot our graph with the following simple command

```
> plot(g)
```

We can add a few arguments to our `plot()` function to improve the visualization:

```
> plot(g, layout = layout.fruchterman.reingold, vertex.size = 5,
    vertex.label.dist = 1)
```

The result of running these commands in `R` for me are displayed in Figure 1. Note that your random graph should appear different, even in the number of edges (but not the number of vertices). Note also that the result from running the function "random.graph.game" is stored in the variable g. This will allow us to do some analyses and plot our graph without building a new graph each time.

Below is the complete code required to test the Solomonoff and Rapaport [12] result. For this test we create a large number of graphs, each with $10,000$ vertices with ever increasing numbers of random edges and determine the size of the giant component. You should copy these lines of code to your script file and then run them. Note that any text that follows the "#" symbol is ignored by `R`. The lines will take several seconds to run but, if entered correctly, will end with a nice graph. And be sure that you have run the `library(igraph)` command before you run these lines.

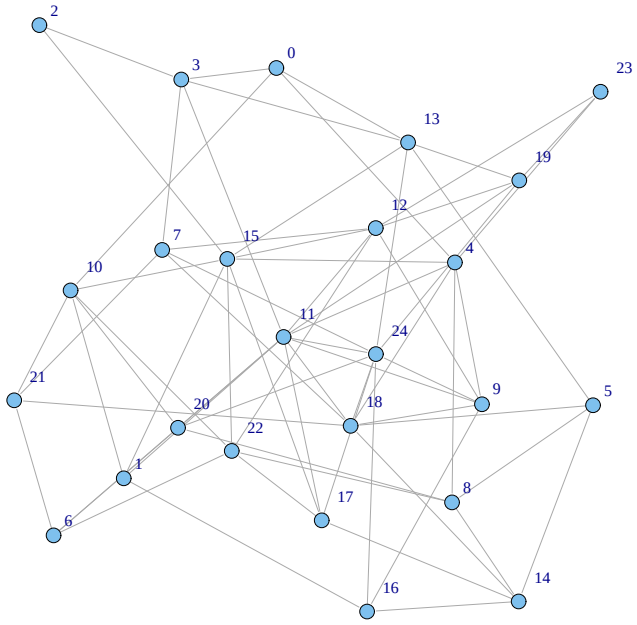

Figure 1: A random graph with 25 vertices and the probability of an edge = 0.1. Note that the first vertex has an $ID = 0$.

```
size.giant.comp = function(g) {max(clusters(g)$csize)}
Nverts = 10000
Nedges = seq(Nverts/10, Nverts*2, by = Nverts/100)
P = numeric(length(Nedges)) # Prop. verts in giant component.
for (i in 1:length(Nedges)) {
    g = erdos.renyi.game(Nverts, Nedges[i], type = "gnm")
    P[i] = size.giant.comp(g)/Nverts
 }
plot(Nedges/Nverts, P,
    ylab = "Proportion of Vertices in Giant Component",
    xlab = "Nedges/Nvertices", xlim = c(0,2), ylim = c(0,1))
    cex.lab = 1.5)
abline(h=0.8);abline(v=1)   # add reference lines
```

This code should look terrifying at first. If we break it down, however, it is doing some remarkable things. You might notice the two lines in the `for` loop really do the bulk of the work). The first line creates a function called `size.giant.comp` that receives a network object g. Behind the scenes, in `R` with the `igraph` package loaded, the `clusters()$csize` function call returns the number of vertices in each component while the function (`max()`) returns the biggest of those components. Our very own function `size.giant.comp` is now ready to use any time we need it during the current session (until you close the `R` program).

The second line defines how many vertices we want in all of our networks. The more we ask for the slower the program will run but the better our test will be. The next line creates an array for the number of edges we are going to test. If, after running these lines of code, you type

```
> Nedges
```

at the command line you will see what the variable looks like. The array is a sequence, starting from the first argument in the function "seq", to the second argument, in increments of the third "by" argument. There should be 191 different numbers of edges that will be tested, regardless of the number of vertices we request.

The "for" loop is often tricky to understand at first. The computer will complete the task in the curly braces however many times we ask it. Here we want it be done for every value for the number of edges we defined in the variable `Nedges`. The loop includes a counter *i* that increases from 1 to however many edges we requested (`Nedges`).

Now, within the `for` loop, the work of the model begins. The first time we enter the `for` loop we ask `R` to create a random graph (*sensu* Erdös and Rényi [5]) with the number of vertices (`Nverts`) and the number of edges (`Nedges[1]`), which is 1000 edges. The second time the `for` loop runs i is incremented to 2 so the progam uses `Nedges[2]`, which is 1100 edges. After each graph is produced the loop calculates the proportion of vertices in the giant component. This value is stored in the i^{th} element of array `P`, which was declared to be a numeric array able to hold the number of values equal to the length of the `Nedges` array.

Once the `for` loop has completed its work and the proportion of vertices in the giant component calculated for each network, the remaining lines generate the plot. The `plot` function takes two important arguments, `x` and `y`. The code following these arguments improves the looks of the graph by giving the axes names and defining the `x-axis` and `y-axis` limits. The first `abline` function places a horizontal reference line at $y = 0.8$ while the second places a vertical line at $x = 1.0$. Your graph should look similar to Figure 2.

The result from Solomonoff and Rapaport [12] appears to be verified. The data points appear to pass through the point $(1, 0.8)$ where our reference lines cross.

A random network is an unusual structure in biological systems. It is hard to imagine a system that would actually be connected randomly. However, this structure is quite useful in that it might function as an appropriate null hypothesis network. The Hardy-Weinberg Principle, for instance,

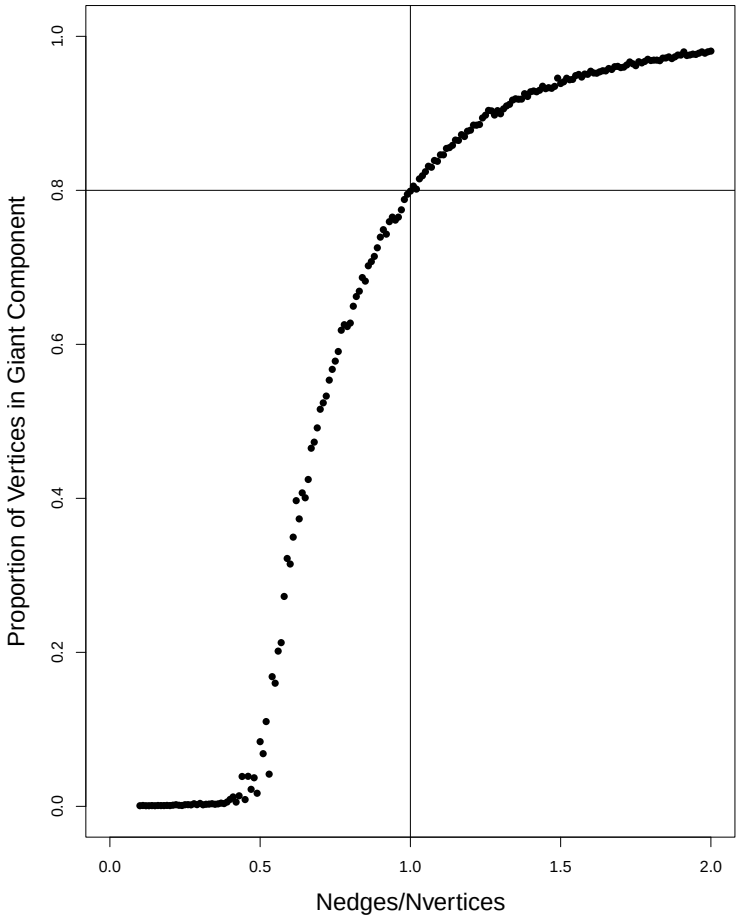

Figure 2: The results from running the Solomonoff and Rapaport [12] test.

assumes random mating. It might be conceivable to consider matings among individuals in a population using this type of random network and comparing the outcome of a model against a network that had a more realistic structure.

The Watts-Strogatz "Small-World" and the Albert and Barabási "Scale-Free" Graphs

There are many graph structures that have received a great deal of attention and are used in biology. Two important structures include the so-called "small-world" network generated using a rewiring algorithm on a circular graph by Watts and Strogatz [13] and a "scale-free" network created by Albert and Barabási [1]. Each of these is built in a similar manner in `R` to the Erdös–Rényi random graphs described above.

These two graph structures have been used for a variety of purposes in biology and can yield quite different results. This demonstrates that the structure of the underlying network of a biological system can greatly influence the dynamics of the system (see below). In short, the Watts-Strogatz graph allows one to investigate, on a continuous scale, the transition from a network with high clustering coefficient (the neighbors of individual vertices are connected) and high average path length (vertices are many connections away from each other, on average) to structures with low clustering coefficients and low average path lengths (similar to random graphs).

In contrast, the "scale-free" network is appropriate for systems that grow by accumulation of vertices that attach preferentially to other vertices based on their 'degree', or the relative number of connections that vertices have in the network.

Information on these functions is available by typing:

```
> ?watts.strogatz.game
> ?barabasi.game
```

These graphs each have their own set of arguments that need to be supplied in order to generate the appropriate graph. Both functions have flexibility in the number of vertices, edges, and the dimensionality of the networks produced. The various properties of these graphs is beyond the scope of this paper but the reader is encouraged to peruse the extensive literatures on these graphs and the many books that address such properties (*e.g.*, Newman, Barabási, and Watts [9].

Simulating Vaccination Strategies

One application that is well suited for testing the effect of changing graph structure is to study the spread of a disease agent (*e.g.*, virus) through a population. We imagine that each individual in a population is a vertex in a graph and the connections between these individuals (called "edges") represent the possible transmission route between these individuals.

To see how this might work we will test a random vaccination strategy using the Watts–Strogatz network. The goal for our work here is simply to determine the maximum size of the resulting giant component following vaccination. We will not actually spread the disease agent through the network, although this is not hard to implement. This is important because if a network is broken into smaller components (groups of connected individuals) through our vaccination effort then, at least in theory, if a single individual in our population becomes infected then the largest number of people who might get sick is equal to the size of the largest component.

Hartvigsen et al. [7] investigated a variety of vaccination strategies, including random vaccinations as a type of control. In addition, they tested vaccinating individuals based on the number of connections each individual had (vertex "degree") as well as preferentially based on either high or low clustering coefficients. Below we will build a Watts-Strogatz [13] network and vaccinate these graphs using a simple random strategy and different levels of effort (from no vaccinations to 90% of the vertices vaccinated).

The Hartvigsen et al. [7] simulation actually was written using the C programming language but all of the work could now be implemented very quickly using the `R` environment with the

`igraph` package. It is important to note that simulation development time using `R` and `igraph` can be just a fraction of what it would take to write such simulations from scratch in a language like C. However, thorough testing of hypotheses requires running many replicate simulations on relatively large networks which can be substantially slower using `R` and `igraph`.

The Watts-Strogratz [13] network relies on a rewiring parameter that looks at individuals edges of each vertex and, with a probability *prw*, disconnects an edge from a neighbor and attaches it to a randomly selected vertex in the graph (note that self-loops are allowed in this implementation). In this way the graphs are easy to compare since they all contain the same number of vertices and edges with only the structure differing among graphs.

To simulate vaccination of a host we will simply remove the host from the graph, which also removes its edges. One might be interested in selectively removing (or adding) edges to change the graphs's structure. The code for our test might be written as follows:

```
size.giant.comp = function(g) {max(clusters(g)$csize)}
Nverts = 1000
prw = 0.05  #   Prob of rewiring parameter
VE = seq(0,0.9, by = 0.05)  # Vaccination Effort
SGC = numeric(length(VE))   # array hold size of giant comps.
for (i in 1:length(VE)) {
    g = watts.strogatz.game(1, size = Nverts, nei = 2, p = prw)
    Ndel = VE[i]*Nverts
    g = delete.vertices(g, sample(V(g),Ndel))
    SGC[i] = size.giant.comp(g)
}
plot(VE,SGC, type = "b")
```

This set of instructions might appear daunting at first. If we look line-by-line, however, it is not too bad. The first line you have seen before, and is a function that returns the size of the giant component in graph *g*. Next we declare the size of the graphs we will create (`Nverts`. The *prw* variable is the probability of rewiring which, for our purpose we will fix as $prw = 0.05$. We establish our vaccination efforts (VE), which is an array of numbers from 0 to 0.9 by intervals of 0.05. We create a variable that will hold the size of the giant components for each vaccination effort we try (SGC).

Within the `for` loop we create a network *g*. We then determine the number of vertices that need to be deleted (`Ndel`), which is based on vaccination effort (VE) and the size of the network ($Nverts$). The function `V(g)` returns the IDs of all the vertices in our network. The `delete.vertices` function takes two arguments. The first is the graph we wish to operate on. The second argument is a random sample of size `Ndel` (`sample()`) of vertex IDs (`V()`), without replacement. These vertices are then sent to the `delete.vertices()` function to delete. The new network with fewer vertices replaces the original network *g*. We then determine the size of the giant component and store this result in our `SGC` array.

If you copy and paste this code into a script file and run it you should see something similar

to the graph in Figure 3. Notice that it appears the relationship between the size of the giant component and vaccination effort is not linear. The size of the giant component drops to near zero when we remove (or vaccinate) about 60% of the vertices. This suggests that even with a random vaccination strategy we do not have to vaccinate everyone in a network to greatly reduce the size of an epidemic. Obviously there are great opportunities to work to determine better vaccination strategies than random and to investigate this on a wide variety of supplied network structures.

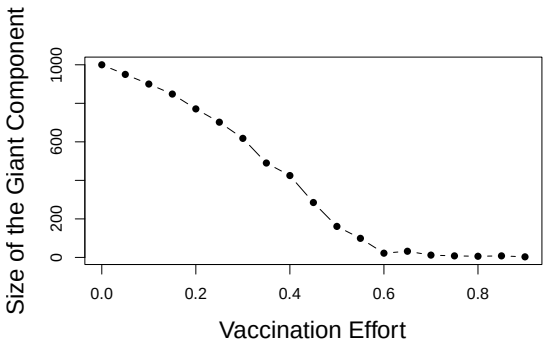

Figure 3: The size of the giant component decreases with increasing vaccination effort on a Watts-Strogatz [13] network with $P = 0.05$.

Additional Supported Functions

In this paper we have seen only a fraction of the analytical tools available in `igraph`. These include a variety of valuable routines for assessing such basic analyses as determining the graph's degree distribution. Not surprisingly, these routines are simple in `R`. The degree distribution is easily compared between the Watts–Strogatz network and the Barabasi scale–free network, for instance. This comparison can be done as follows:

```
> g = watts.strogatz.game(1,size=100000,nei=4,p=0.25)
> h = barabasi.game(100000) # a million nodes takes a few seconds
> a = hist(degree(g))
> b = hist(degree(h))

> par(mfrow = c(1,2))
> plot(1:length(a$counts),a$counts+1,
+   xlab = "Degree", ylab = "Frequency", cex.lab = 1.5,
+      main = "Watts-Strogatz", log = "xy", type = "b")
> plot(1:length(b$counts),b$counts+1,
+   xlab = "Degree",ylab = "Frequency", cex.lab = 1.5,
+   main = "Scale-Free", log = "xy", type = "b")
> par(mfrow = c(1,1))
```

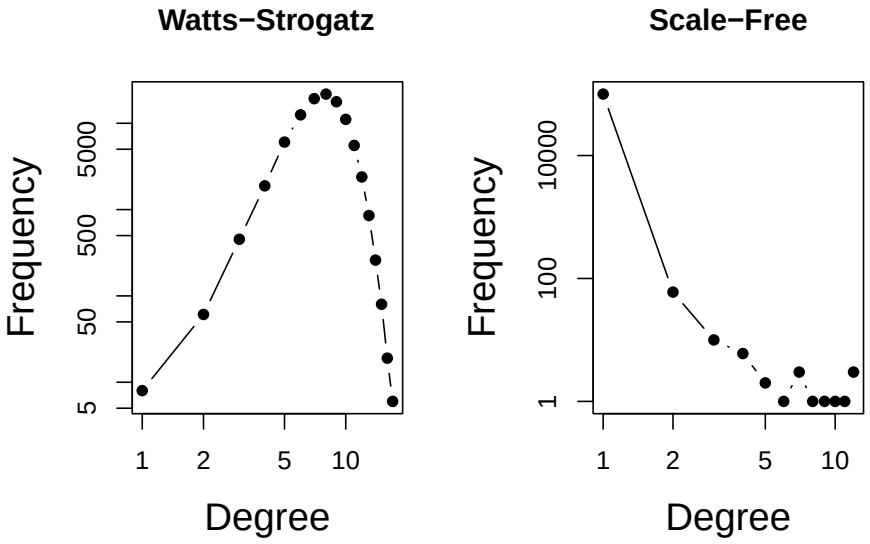

Figure 4: The degree distribution for the Watts–Strogatz network (left) and Barabási network (right).

The resulting networks from this code are found in Figure 4. We see that the two separate networks are created and assigned different variable names (`g` and `h`). The `hist()` function, which takes all the degrees for all vertices and creates a histogram, or "degree distribution", can be run as follows:

```
> hist(degree(g))
```

but the return values are not stored. Many graphing functions in `R` actually return information, such as the hinges from a boxplot. Here we are interested in the actual frequencies of the degrees so that we can plot the degree distributions. The `par` function sets up the graphics window so that two plots are graphed side–by–side (the `mfrow = c(1,2)` argument means one row and two columns). Following the `par()` call are the two `plot()` function calls.

The resulting degree distribution graphs are quite different (Figure 4). The Watts-Strogatz network yields a hump-shaped degree distribution while the Barabäsi network yields a scale–free degree distribution, also known as a power–law distribution.

The `igraph` package includes many additional functions for building a wide range of graphs, reading and writing graphs (`read.graph()` and `write.graph`) and for assessing the structure of networks, such as those for calculating a network's clustering coefficient (`transitivity()`), finding all "maximal cliques" (`cliques()`), calculating the "betweenness" values for vertices (`betweenness()`), finding "communities" (`walktrap.community()`), and finding "shortest paths" between vertices (`shortest.paths()`). The reader will find additional information

in the igraph manual (see
http://cran.r-project.org/web/packages/igraph/index.html).

Conclusions

We have used several powerful tools available through the R programming and statistical environment, along with the igraph package, to build networks, assess their structure, and simulate a vaccination strategy. With the use of this programming environment biologists are able to ask and answer important questions over a range of problems. We have seen some of this through the assessment of the Watts-Strogatz small–world network, Barabäsi's scale–free network, and the Erdös–Rényi random graph.

Acknowledgements

I would like to thank John Jungck for inviting this paper and the National Institute for Mathematical and Biological Synthesis for hosting the 2010 "Graph Theory and Biological Networks Tutorial." I also would like to thank two anonymous reviewers and Elsa Schaefer for their insightful comments on an earlier draft. In addition, I am pleased to acknowledge the developers and contributors to the R project [11] and to acknowledge Gabor Csardi, the author and maintainer of the igraph package.

References

[1] R. Albert, A.-L. Barabási. *Statistical mechanics of complex networks*. Reviews of Modern Physics, 74 (2002), 47–97.

[2] G. Chartrand. *Introductory graph theory*. Dover Publications, 1985.

[3] M. Crawley. *The R book*. John Wiley & Sons, 2007.

[4] P. Dalgaard. *Introductory Statistics with R, 2/e*. Springer, 2008.

[5] P. Erdös, A. Rényi. *On Random Graphs. I.* Publicationes Mathematicae, 6 (1959), 290-297.

[6] L. Euler. *Solutio problematis ad geometriam situs pertinentis*. Commentarii academiae scientiarum Petropolitanae, 8, (1741), 128-140.

[7] G. Hartvigsen, J.M. Dresch, A.L. Zielinski, A.J. Macula, C.C. Leary. *Network structure and vaccination strategy and effort interact to affect the dynamics of influenza epidemics*. Journal of Theoretical Biology, 246, Vol. 2, (2007), 205–213.

[8] W.O. Kermack, A.G. McKendrick. *A Contribution to the mathematical theory of epidemics.* Proceedings of the Royal Society of London A, 115, (1927), 700–721.

[9] M.E. Newman, A.-L. Barabási, D.J. Watts. *The Structure and Dynamics of Networks.* Princeton University Press, 2006.

[10] M.E. Newman. *Networks: an Introduction*. Oxford University Press, 2010.

[11] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing (2010), Vienna, Austria. URL http://www.R-project.org/.

[12] R. Solomonoff, A. Rapoport. *Connectivity of random nets.* Bulletin of Mathematical Biophysics, 13 (1951), 107.

[13] D.J. Watts, S.H. Strogatz. *Collective dynamics of "small-world" networks.* Nature, 393 (1998), 440–442.

[14] A.F. Zuur, E.N. Ieno, E.H.W.G. Meesters. *A Beginner's Guide to R*, Springer, 2009.