AM3911 Major Assignment

Epidemic Spreading in Real Networks

Patrick S. Mahon
pmahon3@uwo.ca

April 24, 2020

Contents

Abstract

In "Epidemic Thresholds in Real Networks" Wang et al. analytically derived a metric, the *epidemic threshold*, summarizing the action of *viral* elements persisting on networks of *nodes* connected with *edges*, i.e. *graph based networks* [6]. If the threshold is exceeded, a positive number of infected individuals persists on the graph indefinitely and an *epidemic* is achieved. Otherwise the number of infected individuals tends to zero as time progresses and the outbreak dies out. Their epidemic threshold is novel in that it generally and arbitrarily describes infectious dynamics with respect to network structure, i.e. it is *topologically independent*. Wang et al. perform discrete stochastic numerical simulations on networks which mimic *real network* interactions, like those exhibited by social and biological networks, to confirm their analytic conclusion. This paper explicates the main conclusions of Wang et al.'s analytic result and presents recreations of two different simulations performed in their paper; simulations of infectious spread in *Barabási-Albert power law graphs* (BA power law graphs) and *Erdös-Rényi graphs* (ER graphs).

1 Background

Wang et al. present their results at the intersection of graph theory and infectious disease modelling. The following defines and explains the terms and concepts used in this assignment.

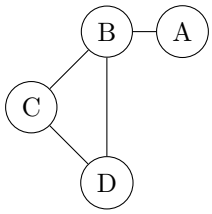
1.1 Graph Theory

1.1.1 What is a graph?

Graph theory is study of mathematical objects called graphs. A *graph*, $\mathcal{G}$, consists of a set of *vertices* or *nodes*, $\mathcal{V}$, connected to one another by a set of *edges*, $\mathcal{E}$,

$$\mathcal{G} = (\mathcal{V}, \mathcal{E})$$

There are many different ways to represent a graph. A simple drawing is the most straightforward:



The edges can be directed or undirected as well as weighted or unweighted. For our purposes we only need to consider undirected unweighted graphs.

1.1.2 The adjacency matrix

Graphs can also be represented in a special kind of matrix called an *adjacency matrix*. The adjacency matrix for the graph drawn above is,

$$\begin{array}{c} \\ \text{A} \\ \text{B} \\ \text{C} \\ \text{D} \end{array} \begin{array}{c} \begin{array}{cccc} \text{A} & \text{B} & \text{C} & \text{D} \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

The entries in an adjacency matrix take binary values. If the ij-th entry of an adjacency matrix equals one, that means there is an edge connecting nodes i and j, if it is zero then nodes i and j do not have an edge connecting them.

In the graph above we can see that node B is connected to three other nodes. In this case we would say the *degree* of node B is three since it has three edges emanating from it.

Lastly we should note that adjacency matrices are always symmetric because they contain two copies of each node organized as the rows and columns of the matrix.

1.1.3 Random graphs

The distribution of edges in a graph need not be deterministic. *Random graphs* are graphs whose vertices are randomly connected to one another. We can construct a random graph to have certain properties that follow a specific probability distribution. For example we might want the degree of any given node to be binomially distributed according to some probability p,

$$P(\deg(v) = k) \sim \binom{n-1}{k} p^k (1-p)^{n-1-k}$$

Every possible edge in a graph like this has equal probability p of existing. These graphs are *Erdös-Rényi graphs* (ER graphs) and are one the two types of random graphs considered by Wang et al., the other being *Barabási-Albert power law graphs* (BA graphs) whose node degrees follow a power law distribution [4],

$$P(\deg(v) = k) \sim k^{-\gamma}$$

In these graphs there are many nodes with very few connections and very few nodes with many connections. BA graphs are often called *real graphs* in the sense that many real world networks exhibit properties of BA graphs. Wang et al. consider simulated BA graphs with $\gamma = 3$.

1.1.4 Generating random graphs

Random graphs are generated using procedures that produce the desired statistical properties. Generation of ER graphs is relatively straightforward. Given the probability p for an edge to exist we can generate an ER graph like this,

Algorithm 1 ER Graph Generation

```
1: procedure GenerateERGraph(p, size)
2:     for i = 0, i < size do
3:         for node j = i + 1, j < size do
4:             if randNum ∈ [0, 1] < p then
5:                 Add edge between i and j
6:             end if
7:         end for
8:     end for
9: end procedure
```

Generating a BA graph is a little trickier. The algorithm begins by generating a connected graph of m_0 nodes. The remaining unconnected nodes are then randomly connected to m_0 nodes in the graph favouring nodes of higher degree. The probability of connecting to any given node j is given by the ratio of the degree of j to the total number of edges in the graph,

$$p = \frac{\deg(j)}{\text{numEdges}}$$

The following algorithm generates a BA graph with a degree distribution of

$$P(\deg(v) = k) \sim k^{-3}$$

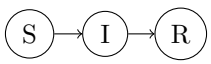
and mean node degree of m_0,

Algorithm 2 BA Graph Generation

```
1:  procedure GenerateBAGraph(m_0, size)
2:      numEdges ← 0
3:      for (i = 0, i < m_0) do                    ▷ Generate initial connected graph
4:          for (j = i + 1, j < m_0) do
5:              Add edge between i and j
6:              numEdges++
7:          end for
8:      end for
9:
10:     for (i = m_0, i < size) do                 ▷ Preferentially add remaining nodes
11:         while (deg(i) < m_0) do
12:             j ← randomNode ∈ [0, i − 1)
13:             while ( (∃ edge between i ) do
14:                 j ← randomNode ∈ [0, i − 1)
15:             end while
16:             p ← deg(j)/numEdges
17:             if randNum ∈ [0, 1] < p then
18:                 Add edge between i and j
19:                 numEdges++
20:             end if
21:         end while
22:     end for
23: end procedure
```

1.2 Compartmental Epidemiological Modelling

There are several different types of mathematical epidemiological models. The most popular of these are *compartmental* in that they categorize members of a population according to their state with respect to the virus. These models are typically described by an acronym denoting the states of infection an individual progresses through. For example, in a *SIR* model an individual is initially *susceptible* to the infection. Once in contact with another infected individual there is a probability they will be *infected*, after which there is a chance they can become *recovered* and no longer susceptible to the virus.

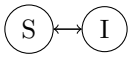


Measles, chicken pox, and rubella are SIR diseases; long lasting immunity is conferred upon recovery. Other possible states of infection include *exposed*, *carrier*, or *maternally-derived immunity*. We can combine these states to make different viral models. For example, in a *MSIR* model individuals are born with maternally derived immunity, become susceptible, then infected, and finally recovered. "Recovered" is sometimes referred to as "removed" when there is a probability that an infected individual can die.

The models can be stochastic or deterministic as well as discrete or continuous and typically fall into one of two categories: either deterministic and continuous, or stochastic and discrete. The deterministic and continuous variety are typically couched as a set of differential equations, whereas stochastic and discrete models are often graph based.

1.2.1 Graph based SIS models

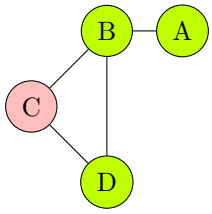
Wang et al. consider graph *SIS* models. In an SIS model an individual is initially susceptible, can become infected, and can recover to the susceptible state once again.



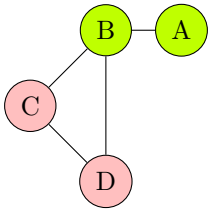
These models roughly describe the dynamics of diseases like the common cold or seasonal influenza. Here the authors are addressing SIS models in the context of computer viruses where network structure is clearly defined and relatively easy to observe.

In these models a population is represented by a graph: each node represents an individual member of the population and its edges represent contact to other individuals. For example, the graph shown in section 1.1.1 contains four individuals: individual B in contact with every other individual in the network, individuals C and D with each other and B, and individual A with B.

Each individual could either be in the infected or susceptible state. Let's say at time t_0 individual C is infected and everyone else is susceptible,



Individuals B and D are in contact with C and consequently exposed to infection whereas individual A is not exposed. Wang et al. employ a model where an infected individual has a probability β of infecting any susceptible neighbour for a given time step. In this case the probability D is infected is just β because it only shares one edge with an infected node, namely C. The same can be said for B. Let's say only D is infected during this time step. At the end of the step we have,



For the next time step only node B is exposed. In this case the probability B becomes infected has increased to $2\beta - \beta^2$. [1] Finally each infected node has a probability δ of recovering during each time step. For each time step the probability an infected node recovers and the probability that node infects a susceptible node are independent and do not affect one another; an infected node can both infect other nodes and recover in the same time step. These two probabilities, β and δ, govern the local interactions between susceptible and infected nodes.

1.2.2 A Markov chain of an infected population

At a system wide scale the state of the population at a specific time depends only on the configuration of the system in the previous time step. As a result this model is a *Markov chain*. Given that each node has only two possible states, the number of possible configurations of a network of n nodes is 2^n. In other words the Markov chain for a network of this type has 2^n possible states where the transition between each state of the Markov chain is governed by the interactions outlined above.

2 The Epidemic Threshold

The conditions under which a viral outbreak becomes an epidemic is codified by the *epidemic threshold*. Wang et al. define the epidemic threshold, τ, to be the number such that,

$\frac{\beta}{\delta} < \tau \rightarrow$ infection dies out over time

$\frac{\beta}{\delta} > \tau \rightarrow$ infection survives and becomes an epidemic

Previously derived epidemic thresholds were limited to specific graph topologies. No "unifying model for *arbitrary*, *real* graphs" had been proposed [2]. The threshold they derive is,

$$\tau = \frac{1}{\lambda_{1,A}} \tag{1}$$

[1] In general the probability a susceptible node x is infected is given by a special case of the inclusion-exclusion principle for probabilities,

$$\mathbb{P}(\text{node } x \text{ becomes infected}) = \sum_{k=1}^{n} (-1)^{k-1} \left(\begin{array}{c} n \\ k \end{array} \right) \beta^n$$

where n is the number of infected neighbours of x.

where $\lambda_{1,A}$ is the principle eigenvalue of the adjacency matrix $\mathbf{A}$ of the network [2, 6]. If,

$$\frac{\beta}{\delta} < \tau = \frac{1}{\lambda_{1,A}}$$

then the probability any given node is infected goes to zero as time progresses. The authors use this property to define another metric called the score, s, of a network,

$$s = \frac{\beta}{\delta} \cdot \lambda_{1,A} \tag{2}$$

For an arbitrary undirected graph, if $s > 1$ the outbreak will become a pandemic, if $s < 1$ " the epidemic will die out over time irrespective of the size of the initial outbreak" [2].

3 Simulations

3.1 Wang et al.'s Simulations

Wang et al. considered six graphs, two of which were ER and BA graphs. The ER graph consisted of 256 nodes and 982 edges. The edge probability was not specified. A 3000 node, 5980 edge, BA graph was generated using the Barabási-Albert process [1], giving a power-law graph with $\gamma = 3$. The initial state of all nodes was set to infected and epidemics were simulated for various score values. The number of infected nodes was recorded after 500,1000, and 2000 time steps. The number of infected nodes was plotted against the score (Figure 1).

3.2 Performed Simulations

For this assignment four different ER graphs and BA graphs of 1000 nodes each were generated. Simulations were carried out in parallel on four cores. The edge probability for the ER graphs was set to 0.01 and the mean degree for the BA graphs was 2.0. The virus birth rate, β, was fixed at 0.05 and virus recovery rates, δ, were spaced out in increments of 0.01 on $[0.1, 0.9]$. In the initial state all nodes were infected. Simulations were run for 500 and 1000 time steps. Three simulations were run for each set of parameters and for each set, the average (of the three simulations) of the average ratio of infected nodes over the course of the simulation were recorded plotted against the score (Figures 2,3,4,and 5).

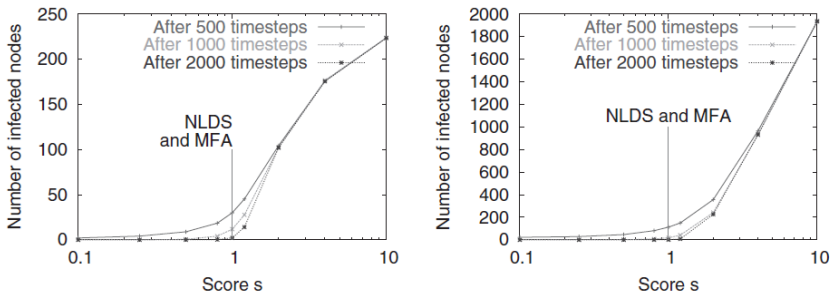
4 Results and Conclusion

The results for the graphs are fairly clear; above an epidemic threshold score of one there is a sharp jump in the number of infected nodes in the network. The recreated simulations in this assignment agree with the results in Wang et al. The authors found these results to be more accurate than previous results and note that there are a number of applications for their epidemic threshold.

Most notably the spread of a viral outbreak can be mitigated by; (i) optimizing immunization schemes by selecting those nodes that maximally reduce the principle eigenvalue of the network $\lambda_{1,A}$; (ii) "Throttling" networks by minimizing the virus birth rate, β, (in computer networks this could be achieved by a reduction in network connection speeds, in human populations by behaviours including hand washing, sneezing into one's elbow, etc.), and lastly; (iii) altering network structure to reduce $\lambda_{1,A}$ (i.e. organizing computer networks appropriately and implementing "social distancing" in human populations).

A Appendices

A.1 Figures



(a) *RANDOM* (b) *POWER LAW*

Figure 1: Infected nodes for random (a.k.a ER) and power law (a.k.a BA) graphs in an SIS model at various scores. NLDS refers to the threshold laid out in Wang et al. and MLA refers to a previous analysis by other authors [2].

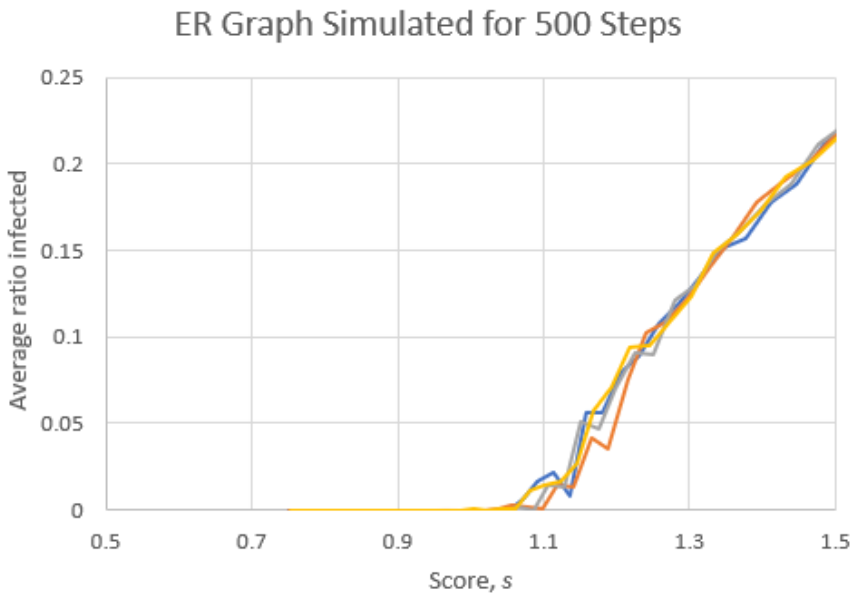


Figure 2: Ratio of infected nodes to total graph size in an SIS model for four ER graphs of size 1000 over 500 time steps.

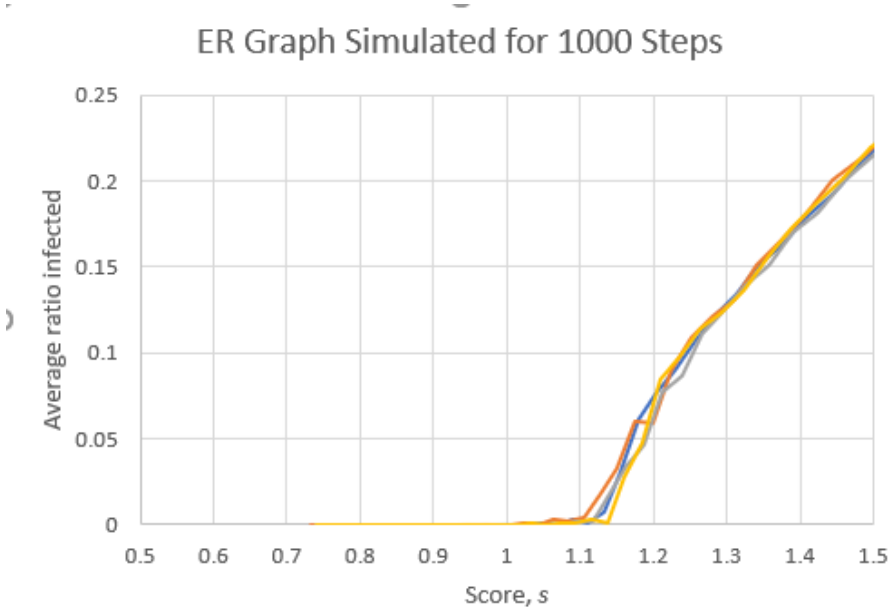


Figure 3: Ratio of infected nodes to total graph size in a SIS model for four ER graphs of size 1000 over 1000 time steps.

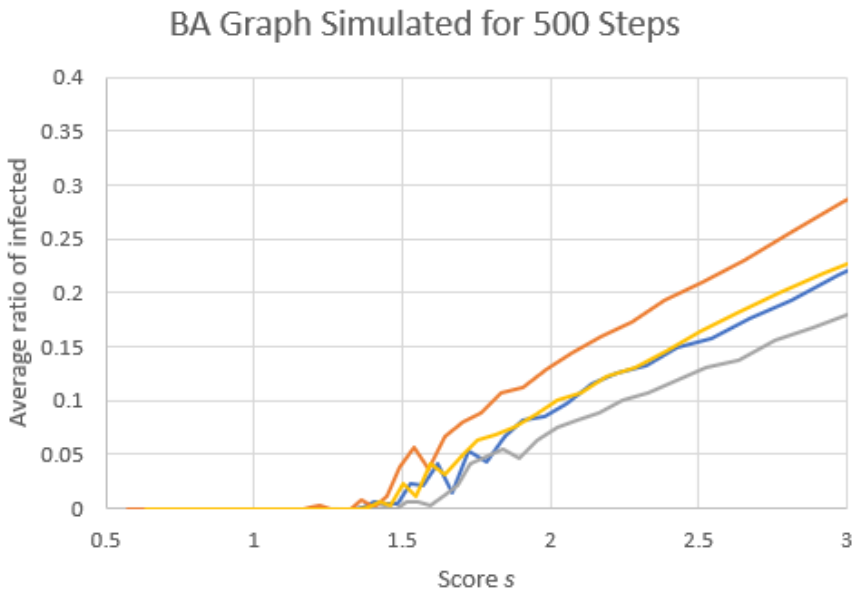


Figure 4: Ratio of infected nodes to total graph size in a SIS model for four BA graphs of size 1000 over 500 time steps.

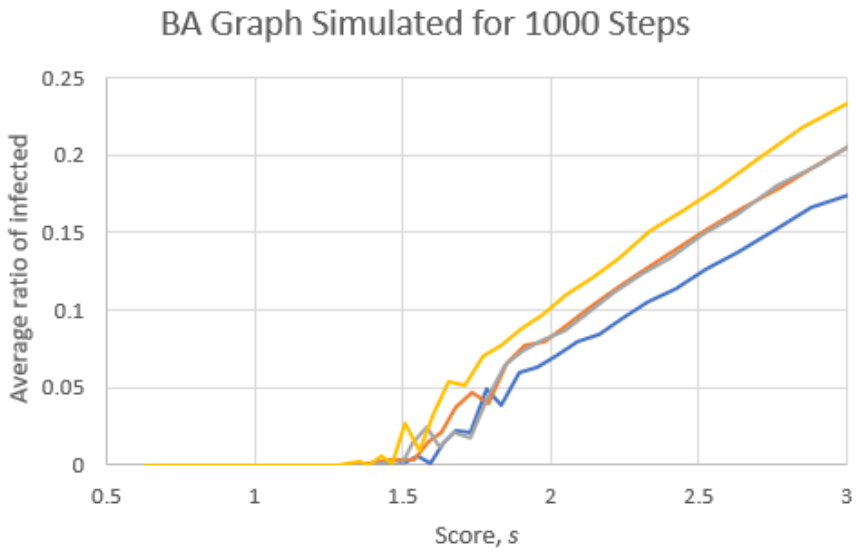


Figure 5: Ratio of infected nodes to total graph size in an SIS model for four BA graphs of size 1000 over 1000 time steps.

A.2 Code

The code was written in C. Random numbers were generated using a Mersenne-Twister pseudo-random number generator [3, 5]. Graphs were implemented using $n \times n$ adjacency matrices, with node status stored in a vector of length n. The number of infected neighbours for a given node was stored in a vector of length n, calculated as the matrix product of the adjacency matrix and the status vector. Although computationally inefficient compared to an adjacency list, the adjacency matrix is much easier to implement and the matrix computations for graphs considered here is fairly quick. Unfortunately the current implementation appears to suffer a memory leak, small graph sizes of less than two thousand nodes over a period of less than one thousand time steps are recommended

Eigenvalues for adjacency matrices were calculated using power iteration [7].

Driver.c: Top level of the program implementing parameter initialization and parallelization.

```
#include "Graph.h"
#include "MT19937.h"
#include "AuxillaryFunctions.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#define NTHREADS 4

void *mainfun(void *x){

  // Get thread id
  int tid = *((int *) x);
  // Create a results file for the thread
  char buf[0x100];
  snprintf(buf, sizeof(buf), "Thread%d.csv", tid);

  FILE* results  = fopen(buf, "w");

  sgenrand(time(0));

  int tsteps = 900;
  int burnin = 100;
  int size = 1000;

  int nreplicates = 3;

  // Graph initialization parameters; uncomment either 1. or 2. for the
requisite graph and...
  // select the corresponding adjacency matrix generating function in the
createGraph function in Graph.c.
  // 1. Edge probability for an Erdos-Renyi graph
  double p = 0.01;
  // 2. Mean degree for Barabasi-Albert graph
  //double m_0 = 2;

  // Graph initialization
  graph* g = malloc(sizeof(struct graph));
  createGraph(g, size, p, 0, 0);

  // Run simulations

  double beta = 0.05;
  double delta = 0.9;
  double score = beta / delta * g->lambda;

  while( delta >= 0.01){

    g->beta = beta;
```

```
    g->delta = delta;

    double totalavginfected = 0;
    for ( int i = 0; i < nreplicates ; i++ ){
      simEpidemic(g, burnin, tsteps);

      totalavginfected += g->avginfected;
    }

    printf("Thread %d\nLambda: %f\nBeta: %f\nDelta: %f\nScore: %f\nRatio
infected: %f\n\n", tid, g->lambda, g->beta, g->delta, score, totalavginfected
/ (double) nreplicates );
    fprintf(results, "%f, %f,\n", score, totalavginfected / (double)
nreplicates );
    delta -= 0.01;
    score = beta / delta * g->lambda;
  }

  fclose(results);

  printf("Thread %d complete.\n", tid);

  deleteGraph(g);
  return NULL;
}

int main(){

  // Parallelization code sourced from
gribblelab.org/CBootCamp/A2_Parallel_Programming_in_C.html
  pthread_t threads[NTHREADS];
  int thread_args[NTHREADS];
  int rc, i;

  /* spawn the threads */
  for (i=0; i<NTHREADS; ++i)
    {
      thread_args[i] = i;
      printf("spawning thread %d\n", i);
      rc = pthread_create(&threads[i], NULL, mainfun, (void *)
&thread_args[i]);
    }

  /* wait for threads to finish */
  for (i=0; i<NTHREADS; ++i) {
    rc = pthread_join(threads[i], NULL);
  }

  return 0;
}
```

Graph.c: Implements the functions dealing directly with simulation and graph maniupulation.

```
#include "Graph.h"
#include "AuxillaryFunctions.h"
#include "MT19937.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

void simEpidemic(graph* g, int burnin, int timesteps ){
  // Select initial infected population
  g->infected = 0;
  g->susceptible = g->size;

  for ( int i = 0 ; i < (int)g->size ; i++) {
    g->nodevec[i] = 1;
    g->infected += 1;
    g->susceptible -= 1;
  }
  setInfVec(g);

  // Cumalitve total infected over the course of the simulation
  int cuminf = 0;

  // Run time steps
  for ( int time = 0 ; time < timesteps ; time++ ){

    // Set vector tracking infected negihbours
    setInfVec( g );

    // newnodevec stores the changes to the current state.
    // g->nodevec stores the current state (i..e. what we are using to compute
newnodevec.
    int* newnodevec = malloc(sizeof(int)*g->size);

    // Evaluate each node
    for ( int node = 0 ; node < g->size ; node++ ){

      newnodevec[node] = g->nodevec[node];

      // Infection
      if ( g->nodevec[node] == 0 ){

    for ( int i = 0 ; i < g->infvec[node] ; i++ ){
      if ( compareDouble( g->beta, genrand()) ){
        //printf("Node Infected: %d\tProb: %f\n\n", node, nodephi);
        newnodevec[node] = 1;
        g->infected += 1;
        g->susceptible -=1;
        break;
      }
    }
      }
      // Recovery
      else if ( g->nodevec[node] == 1 ){
    if ( compareDouble( g->delta, genrand()) ){
      //printf("Node Recovered: %d\tProb: %f\n\n", node, prob);
```

```
      newnodevec[node] = 0;
      g->susceptible += 1;
      g->infected -= 1;
    }
      }
    }

    // Replace current state with updated state.
    for ( int i = 0; i < g->size ; i++ ){
      g->nodevec[i] = newnodevec[i];
    }
    free(newnodevec);

    // Add to cumalitive infected if burn in is finished

    if( time > burnin ) cuminf += g->infected;
  }

  // Set average infected
  g->avginfected = (double) cuminf / (double) (timesteps - burnin) / (double) g->size;
}

void createGraph(graph* g, int size, double p_m0, double beta, double delta){

  g->size = size;
  g->beta = beta;
  g->delta = delta;
  g->infected = 0;
  g->susceptible = size;
  g->avginfected = 0;

  g->adjmat = malloc(sizeof(int*)*size);
  for ( int i = 0 ; i < size ; i++ ){
    g->adjmat[i] = malloc(sizeof(int)*size);
  }

  // Adjacency matrix generation; either uncomment 1. or 2. for the requisite graph
type
  // 1. Generate adjacency matrix for Erdos-Renyi graph
  genERAdjMat( g->adjmat, g->size, p_m0 );

  // 2. Generate adjacency matrix for Barabasi-Albert power law graph
  // genBAAdjMat( g->adjmat, g->size, (int) p_m0 )

  g->nodevec = malloc(sizeof(int)*size);
  g->infvec = malloc(sizeof(int)*size);
  g->lambda = powerMethod(g->adjmat, 1000, g->size);
  g->tau = 1/g->lambda;
}

void addEdge( graph* g, int nodeA, int nodeB){

  g->adjmat[nodeA][nodeB] = 1;
  g->adjmat[nodeB][nodeA] = 1;

  return;
}
```

```
void removeEdge( graph* g, int nodeA, int nodeB){
  g->adjmat[nodeA][nodeB] = 0;
  g->adjmat[nodeB][nodeA] = 0;
  return;
}

void setInfVec( graph* g ){
  int size = g->size;

  for ( int i = 0; i < size; i++ ){
    int sum = 0;
    for ( int j = 0; j < size; j++){
      sum += g->adjmat[i][j] * ( g->nodevec[j] % 2);
    }
    g->infvec[i] = sum;
  }
}

void setStatus( graph* g, int node, int status ){
  g->nodevec[node] = status;
}

void removeNode( graph* g, int node ){
  int size = g->size;

  for ( int i = 0 ; i < size ; i++ ){
    g->adjmat[node][i] = 0;
    g->adjmat[i][node] = 0;
  }
}

void deleteGraph(graph* g){
  free(g->nodevec);
  free(g->infvec);
  for( int i = 0 ; i < g->size; i++ ){
    free(g->adjmat[i]);
  }
  free(g->adjmat);
  free(g);
}

void printGraph( graph* g){

  int size = g->size;
  printf("Status code:\n");
  printf("\t0 = Susceptible\n\t1 = Infected\n\t2 = immune\n\n");
  printf("======================\n");
  printf("|Node|Status|Infected Contacts|Contacts\n");
  printf("=======================================\n");

  for ( int i = 0 ; i < size; i++){

    printf("|%4d|%6d|%17d|", i, g->nodevec[i], g->infvec[i]);
    for ( int j = 0; j < size; j++ ){
      if ( g->adjmat[i][j] == 1) printf(" %2d ", j);
    }
    printf("\n");
```

```
    printf("----------------------------------------\n");
  }
}
```

AuxillaryFunctions.c: Implements other functions including eigenvalue computation, double comparision and adjacency matrix generation.

```c
#include "AuxillaryFunctions.h"
#include "MT19937.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

// powerMethod computes the principle eigenvalue of a given matrix using the power method or power iteration.
double powerMethod( int** mat, int iters, int size){
  double* x_k = malloc(sizeof(double) * size );
  double* y_k = malloc(sizeof(double) * size );

  for ( int i = 0 ; i < size ; i++ ){
    y_k[i] = 1 + genrand();
  }

  double m_k = 0.0;

  for ( int j = 0 ; j < iters; j++){
    m_k = 0;
    for ( int i = 0 ; i < size ; i++ ){
      double sum = 0;
      for ( int j = 0 ; j < size ; j++ ){
    sum += mat[i][j] * y_k[j];
      }
      x_k[i] = sum;
      if ( compareDouble(sum, m_k) ) m_k = sum;
    }

    for ( int i = 0 ; i < size ; i++ ) y_k[i] = x_k[i] / m_k;
  }

  free(y_k);
  free(x_k);
  return m_k;
}

// compareDouble returns 1 if a is bigger than b and zero otherwise.
int compareDouble( double a, double b){
  double result = a - b;
  int sgn = copysign(1.0, result);

  if ( (int) sgn > 0 ){
    return 1;
  }
  else return 0;
```

```
}

// genERAdjMat generates an Erdos-Renyi graph where each edge has a
probability, p, of existing.

void genERAdjMat(int** adjmat, int size, double p ){
  for ( int i = 0 ; i < size; i++ ){
    for ( int j = i+1 ; j < size ; j++) {
    if ( compareDouble( p, genrand() ) ){
      adjmat[i][j] = 1;
      adjmat[j][i] = 1;
    }
    }
  }
}

// genBAAdjMat generates an adjacency matrix with mean node degree m0, with
node degrees distributed according to the probability P(X=x) = x^(-3)
// For algorithm details see:
//      Barabasi and Albert. (1999) Emergence of scaling in random networs.
Science 286, 509-512
//      Albert and Barabasi. (2002) Statistical mechanics of complex
networks. Reviews of Modern Physics 74, 48-85
//      Prettejohn et al. (2011) Methods for generating complex networks with
selected structural properties for simulations: a review and tutorial for
neuroscientists. Fronties in Computational Neuroscience

void genBAAdjMat(int** adjmat, int size, int m0 ){

  // Initialize the matrix
  int i,j;

  // Intialize vector to store node degrees
  int* degmat = malloc(sizeof(int)*size);

  for ( i = 0; i < size; i++ ){
    degmat[i] = 0;
  }

  //Set initial connected nodes
  int totaldeg = 0;
  for ( i = 0 ; i < m0 ; i++ ){
    for ( j = i + 1; j < m0; j++ ){
      adjmat[i][j] = 1;
      adjmat[j][i] = 1;
      degmat[i]++;
      degmat[j]++;
      totaldeg++;
    }
```

```
  }

  // Add edges to remaining nodes with preference given to nodes with higher
degree
  // For all yet to be connected nodes...
  for ( i = m0 ; i < size ; i++ ){

    // while the current nodes degree is less than the seed...
    while( degmat[i] < m0 ) {

      // select a random node j from the set of nodes already connected, not
equal to i and not adjacent to i,...
      j = (int)(i * genrand());

      while ( adjmat[i][j] == 1 || j == i ){
    j = i * genrand();
      }

      // compute the probability of adding an between i and j
      double prob = (double)degmat[j]/(double)totaldeg;

      // add the edge if a random number on [0,1] is less than the
probability.
      if ( genrand() < prob ){
    adjmat[i][j] = 1;
    adjmat[j][i] = 1;
    degmat[i]++;
    degmat[j]++;
    totaldeg++;
      }
    }
  }
  free(degmat);
  }
```

MT19937.c: Matsumoto and Nishimura's Mersenne-Twister pseudorandom number generator.

```
/* A C-program for MT19937: Real number version  (1998/4/6)    */
/*   genrand() generates one pseudorandom real number (double) */
/* which is uniformly distributed on [0,1]-interval, for each  */
/* call. sgenrand(seed) set initial values to the working area */
/* of 624 words. Before genrand(), sgenrand(seed) must be      */
/* called once. (seed is any 32-bit integer except for 0).     */
/* Integer generator is obtained by modifying two lines.       */
/*   Coded by Takuji Nishimura, considering the suggestions by */
/* Topher Cooper and Marc Rieffel in July-Aug. 1997.           */

/* This library is free software; you can redistribute it and/or   */
/* modify it under the terms of the GNU Library General Public     */
/* License as published by the Free Software Foundation; either    */
/* version 2 of the License, or (at your option) any later         */
/* version.                                                        */
/* This library is distributed in the hope that it will be useful, */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of  */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.            */
/* See the GNU Library General Public License for more details.    */
/* You should have received a copy of the GNU Library General      */
/* Public License along with this library; if not, write to the    */
/* Free Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA   */
/* 02111-1307  USA                                                 */

/* Copyright (C) 1997 Makoto Matsumoto and Takuji Nishimura.       */
/* When you use this, send an email to: matumoto@math.keio.ac.jp   */
/* with an appropriate reference to your work.                     */

/* REFERENCE                                                       */
/* M. Matsumoto and T. Nishimura,                                  */
/* "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform  */
/* Pseudo-Random Number Generator",                                */
```

```
/* ACM Transactions on Modeling and Computer Simulation,           */
/* Vol. 8, No. 1, January 1998, pp 3--30.                          */

#include<stdio.h>

/* Period parameters */
#define N 624
#define M 397
#define MATRIX_A 0x9908b0df   /* constant vector a */
#define UPPER_MASK 0x80000000 /* most significant w-r bits */
#define LOWER_MASK 0x7fffffff /* least significant r bits */

/* Tempering parameters */
#define TEMPERING_MASK_B 0x9d2c5680
#define TEMPERING_MASK_C 0xefc60000
#define TEMPERING_SHIFT_U(y)  (y >> 11)
#define TEMPERING_SHIFT_S(y)  (y << 7)
#define TEMPERING_SHIFT_T(y)  (y << 15)
#define TEMPERING_SHIFT_L(y)  (y >> 18)

static unsigned long mt[N]; /* the array for the state vector  */
static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */

/* initializing the array with a NONZERO seed */
void
sgenrand(seed)
    unsigned long seed;
{
    /* setting initial seeds to mt[N] using         */
    /* the generator Line 25 of Table 1 in          */
    /* [KNUTH 1981, The Art of Computer Programming */
    /*    Vol. 2 (2nd Ed.), pp102]                  */
    mt[0]= seed & 0xffffffff;
```

```
    for (mti=1; mti<N; mti++)
        mt[mti] = (69069 * mt[mti-1]) & 0xffffffff;
}

double /* generating reals */
/* unsigned long */ /* for integer generation */
genrand()
{
    unsigned long y;
    static unsigned long mag01[2]={0x0, MATRIX_A};
    /* mag01[x] = x * MATRIX_A  for x=0,1 */

    if (mti >= N) { /* generate N words at one time */
        int kk;

        if (mti == N+1)   /* if sgenrand() has not been called, */
            sgenrand(4357); /* a default initial seed is used   */

        for (kk=0;kk<N-M;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1];
        }
        for (;kk<N-1;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1];
        }
        y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
        mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1];

        mti = 0;
    }

    y = mt[mti++];
```

```
    y ^= TEMPERING_SHIFT_U(y);
    y ^= TEMPERING_SHIFT_S(y) & TEMPERING_MASK_B;
    y ^= TEMPERING_SHIFT_T(y) & TEMPERING_MASK_C;
    y ^= TEMPERING_SHIFT_L(y);

    return ( (double)y * 2.3283064370807974e-10 ); /* reals */
    /* return y; */ /* for integer generation */
}

/* this main() outputs first 1000 generated numbers  */
/*main()
 *{
 *   int j;

 *   sgenrand(4357); /* any nonzero integer can be used as a seed */
/*   for (j=0; j<1000; j++) {
 *       printf("%10.8f ", genrand());
 *      if (j%8==7) printf("\n");
 *   }
 *   printf("\n");
 }*/
```

References

[1] Albert Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 1999.

[2] Deepayan Chakrabarti, Yang Wang, Chenxi Wang, Jurij Leskovec, and Christos Faloutsos. Epidemic thresholds in real networks. *ACM Transactions on Information and System Security*, 2008.

[3] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modelling and Computer Simulation*, 1998.

[4] Mark Newman, S H Strogatz, and D J Watts. Random graphs with arbitrary degree distributions and their applications. *Physical Review*, 2001.

[5] T. Nishimuram. A c-program for mt19937: Real number version (1998/4/6). `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/VERSIONS/C-LANG/980409/mt19937-1.c`. Accessed: 2020-04-24.

[6] Yang Wang, Deepayan Chakrabart, Chenxi Wang, and Christos Faloutsos. Epidemic spreading in real networks: An eigenvalue viewpoint. *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, 2003.

[7] Wikipedia. Power iteration. `https://en.wikipedia.org/wiki/Power_iteration`. Accessed: 2020-04-24.