Disease spreading on small-world networks

Jeroen Hofman
10194754

October 27, 2011

Abstract

In this report we investigate disease spreading on small-world networks. First we investigate properties of small-world networks and examine two types of disease spreading models, SIR and SIRS. We find that the disease spreading rate on a SIR model scales with the average path length of the network. Furthermore we examine state synchronization of the SIRS model after reaching the equilibrium state. The state synchronization increases with increasing rewire probability p and an order parameter can be defined to find the critical point at which the unsynchronized states become synchronized. The critical point is $p \approx 0.15$. The phase transition seen by means of the order parameter can be explained by a rapidly decreasing clustering coefficient and a large spreading in the distribution of local clustering coefficients in the critical area.

Contents

1 Theory

1.1 Construction of small-world networks

In 1998 Watts and Strogatz invented an algorithm which generates graphs which are neither fully regular nor fully random [1]. One starts with a fully regular ring graph consisting of N nodes, where each node has K neighbors, $K/2$ on each side, with $K >> \log N$ [4]. One then defines a special parameter $0 < p < 1$, called the rewiring probability. Next one considers every edge of the regular graph and rewires the edge with probability p (i.e. keeping one side fixed and connecting the other side to another node). For small values of p this creates a graph which has retained the structure of the original regular graph, but with additional shortcuts. For values of p close to 1 almost all edges are rewired and the graph becomes nearly random. See figure 1 for a graphical interpretation of the algorithm.

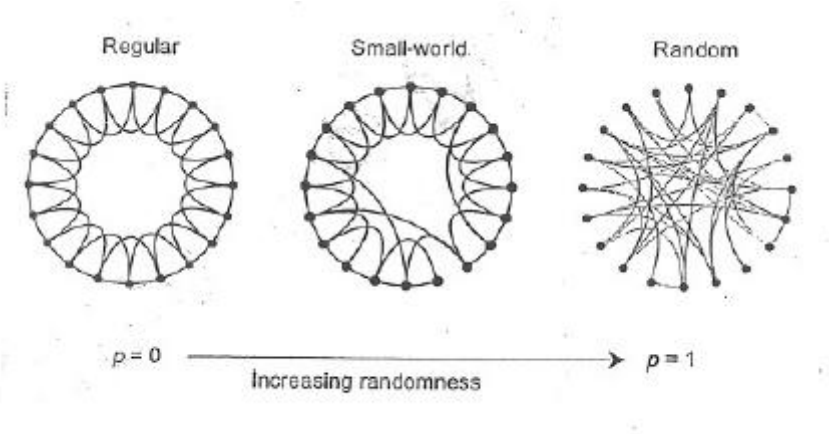

Figure 1: Graphical interpretation of the above algorithm. Starting from a regular network (left) and rewiring connections gives a small-world network (middle). If the rewiring probability is 1 a graph similar to a random graph is obtained (right).

To analyse the structure of a graph two parameters are used; the average path length $\mathrm{L}(p, K)$ describing the average node-to-node distance between any two nodes on the graph and the clustering coefficient $\mathrm{C}(p, K)$, which is the sum of the local clustering coefficients $\mathrm{c}_i(p, K)$:

$$C(p, K) = \frac{1}{N} \sum_{i=1}^{N} \mathrm{c}_i(p, k) \tag{1}$$

where

$$\mathrm{c}_i(p, K) = \frac{\#\text{ connections between neighbors of i}}{\max\ \#\text{ connections between neighbors of i}} \tag{2}$$

where the maximum number of connections between k neighbors is equal to $\frac{k(k-1)}{2}$. These parameters can be calculated for specific values of p with a fixed K, in figure 2 $\mathrm{L}(p)$ and $\mathrm{C}(p)$ are plotted for different values of $0 < p < 1$ with $K = 5$ and $N = 1000$, averaged over 100 realizations per p value.

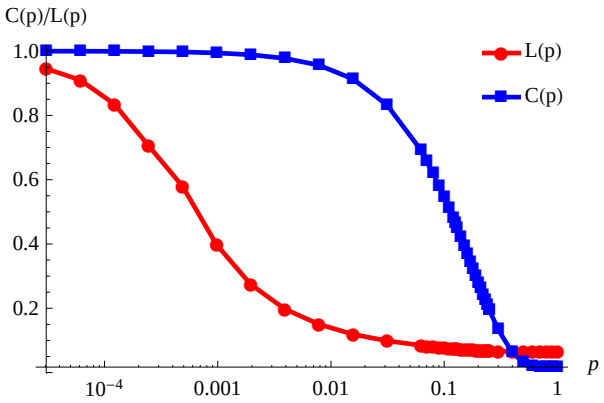

Figure 2: The average path length L(p) and the clustering coefficient C(p) as a function of p.

From figure 2 it is seen that the average path length decreases very rapidly with increasing p, whereas the clustering coefficient only starts decreasing with high values of p. Regular graphs (i.e. $p \approx 0$) have a high clustering coefficient because the graph is essentially one big cluster, and a high average path length because the graph contains no shortcuts. Random graphs ($p \approx 1$) have a low clustering coefficient, because the random edges prevent formation of clusters, and a low average path length because there is an abundance of shortcuts. The area in between, where C(p) is high but L(p) is low is the area in which networks are called small-world networks. Small-world networks resemble many real networks, e.g. social networks, neural networks or distributing networks. A study made in reference [1] indeed shows that many real life networks fall in the category of small-world networks.

1.2 Disease spreading models

In this paper we study two different types of disease spreading on networks, called SIR and SIRS, where the latter is an extension of the former [2]. Starting with a network, we can assign three possible states to every node, susceptible (S), infected (I) and recovered (R), coupled to a disease cycle time counter, which is different for every node. For the SIR network the following relations are defined for the time counter τ_i and state π_i of node i for universal time t:

$$\pi_i(t) = \text{S if } \tau_i(t) = 0 \tag{3}$$
$$\pi_i(t) = \text{I if } 1 \leq \tau_i(t) < t_{\text{inf}} \tag{4}$$
$$\pi_i(t) = \text{R if } \tau_i(t) = t_{\text{inf}} \tag{5}$$

where t_{inf} is the amount of time steps an infected node stays infected. The evolution of the system obeys the following rules:

$$\tau_i(t+1) = 0 \text{ if } \tau_i(t) = 0 \text{ and no infection occurs} \tag{6}$$
$$\tau_i(t+1) = 1 \text{ if } \tau_i(t) = 0 \text{ and node i becomes infected} \tag{7}$$
$$\tau_i(t+1) = \tau_i(t) + 1 \text{ if } 0 < \tau_i(t) < t_{\text{inf}} \tag{8}$$
$$\tau_i(t+1) = \tau_i(t) \text{ if } \tau_i(t) = t_{\text{inf}} \tag{9}$$

We can modify the above schemes to allow a node in the R state to jump back to the S state after a number of steps t_{rec}, thus creating a loop for every node of $t_{\text{inf}} + t_{\text{rec}}$ steps:

$$\pi_i(t) = \text{S if } \tau_i(t) = 0 \tag{10}$$
$$\pi_i(t) = \text{I if } 1 \leq \tau_i(t) < t_{\text{inf}} \tag{11}$$
$$\pi_i(t) = \text{R if } t_{\text{inf}} \leq \tau_i(t) < (t_{\text{inf}} + t_{\text{rec}}) \tag{12}$$

and corresponding rules:

$$\tau_i(t+1) = 0 \text{ if } \tau_i(t) = 0 \text{ and no infection occurs} \tag{13}$$
$$\tau_i(t+1) = 1 \text{ if } \tau_i(t) = 0 \text{ and node i becomes infected} \tag{14}$$
$$\tau_i(t+1) = \tau_i(t) + 1 \text{ if } 0 < \tau_i(t) < (t_{\text{inf}} + t_{\text{rec}}) \tag{15}$$
$$\tau_i(t+1) = 0 \text{ if } \tau_i(t) = (t_{\text{inf}} + t_{\text{rec}}) \tag{16}$$

From the above rules it follows that the SIR model reaches a steady state when all the nodes are either in state S or state R, because there will be no infected nodes after a certain time. For SIRS however, the process can go on indefinitely and, according to the parameters of the model, can exhibit various types of behavior as we will see in the next section.

2 Analysis

2.1 Spreading rates

The SIR model described in the previous section can be used to analyze the behavior of the spreading of diseases in the initial stage of infection, where we assume that the disease can spread uninhibited. In particular two properties can be defined, $r_{1/2}$ and r_{total}, which describe the number of steps before the disease has infected half the number of nodes or all the nodes respectively. Figure 3 shown below shows $r_{1/2}$ and r_{total} as a function of p for $N = 1000$, $K = 5$, $t_{\text{inf}} = 4$ and $p_{\text{inf}} = 1.0$ (the probability of infection), averaged over 100 realizations per p value. The number of steps needed for both halve and full penetration decreases rapidly for increasing p following the characteristic shape of the L(p) curve. Hence the disease spreading scales with the average path length and so for the small-world networks, which are in a region of low average path length, disease spreading will be nearly as fast as for random networks. Secondly the difference between full and half penetration is small for fixed p, which is expected for an infection probability of 1. This is because when half of the nodes is infected, the rest of the network will be infected within a few time steps, because neighbors of infected nodes will become infected with probability 1.

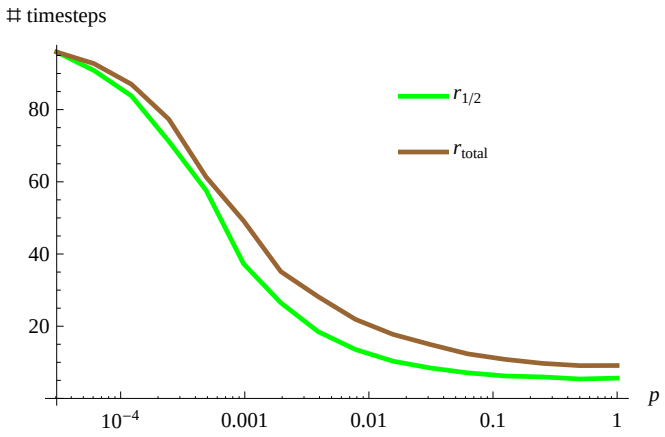

Figure 3: The spreading time for the disease when it has infected half of the graph ($r_{1/2}$) and when it has infected the full graph (r_{total}) as a function of p.

2.2 Equilibrium states

After the initial disease spreading the disease can reach an equilibrium state, which can be modelled using the SIRS model (the SIR model also reaches an equilibrium state, but this is the uninteresting case where all nodes have recovered). In the SIRS model first the disease spreading goes exponentially like the case we analysed in the previous section, but as soon as nodes start transforming from the recovered state to the susceptible state again it allows for the occurence of some oscillatory behavior. Figure 4

shows the behavior of the number of susceptible, infected and recovered nodes when the system has reached equilibrium as a function of time, for $p = 2^{-12}, p = 2^{-5}$ and $p = 1$. The parameters used are $N = 1000$, $K = 5$, $p_{\text{inf}} = 0.1$, $t_{\text{inf}} = 8$ and $t_{\text{rec}} = 8$.

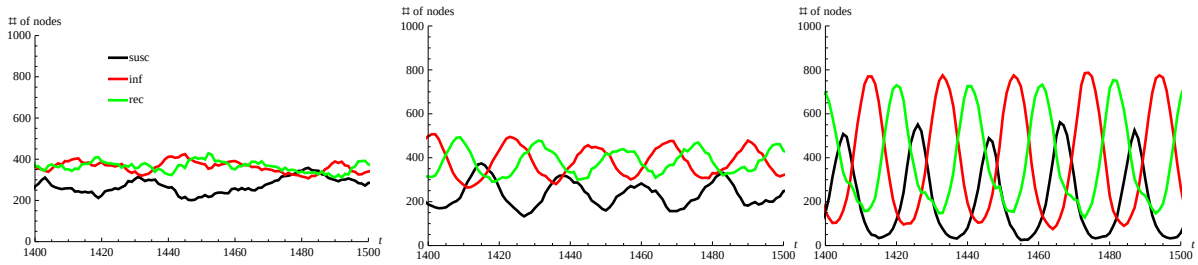

Figure 4: Evolution in time of the number of susceptible, infected and recovered nodes for $p = 2^{-12}$, $p = 2^{-5}$ and $p = 1$.

The higher the value of p the more there seems to be an oscillatory behavior (we will make this more precise below). Furthermore the period of the oscillations seen in the two figures on the right is bigger than $t_{\text{inf}} + t_{\text{rec}}$, namely 22.2 and 20.4 respectively, which is understandable since the system cannot oscillate faster than 1 cycle time takes and the higher p is the more the average path length decreases, which will increase the rate at which the states can be synchronized over the whole network. The figure below also shows this, giving the oscillation period as function of p obtained by fitting a cosine to the data. Indeed the oscillation period decreases for increasing p.

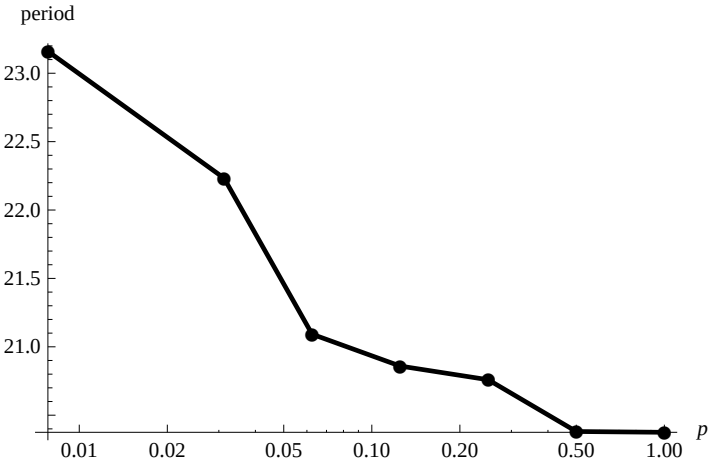

Figure 5: Some values of the oscillation period as a function of p, determined by fitting a cosine to the data.

Though for all values of p there is an equilibrium state the system only starts oscillating for high values of p, which can be explained by small average path lengths which ensure fast synchronization of nodes over the network.
The synchronization of states can be made more precise by using an order parameter [5], defined by:

$$\sigma(t) = \left| \frac{1}{N} \sum_{i=1}^{N} e^{i\phi_i(t)} \right| \quad \text{with } \phi_i(t) = 2\pi \frac{\tau_i(t) - 1}{t_{\text{inf}} + t_{\text{rec}}} \tag{17}$$

where the case $\tau_i(t) = 0$ is not included, because this time state is overrepresented. For low values of p the states are unsynchronized and so $\sigma(t)$ will be close to 0, for high values of p however the states are more synchronized and so $\sigma(t)$ will not be close to zero anymore. Figure 6 below shows σ for different values of p, averaged over time (2000 time steps) and 50 realizations per p, using the same parameters as before.

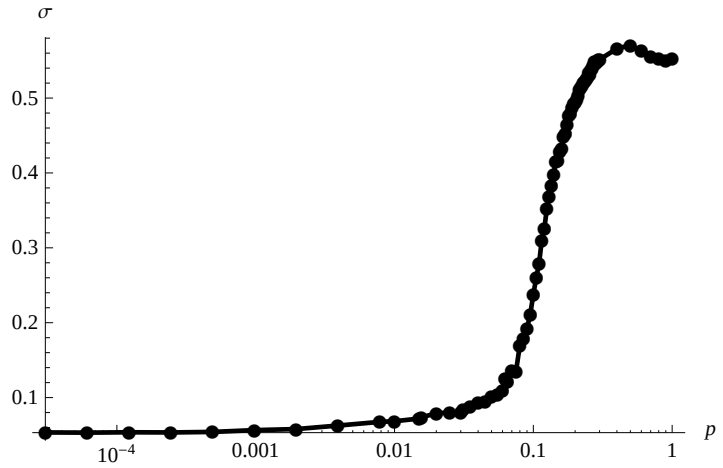

Figure 6: σ, the order parameter as a function of p.

The figure shows a steep increase of σ around $p = 0.15$, indicating that the system goes from an unsynchronized state to a synchronized one around this value. If we compare this with the clustering coefficient in figure 7 it is seen that the increase of σ is in the same region as where the clustering coefficient goes from high values to low values. We can try to understand this phase transition better by also analyzing the standard deviation of the local clustering coefficient $c_i(p)$, which gives a measure of the width of the distribution of local clustering coefficients, and which peaks around the same value as where the derivative of $\sigma(p)$ peaks, see figure 7 below. The spreading of the local clustering coefficient is large around $p = 0.15$, indicating that there are both highly clustered regions and lowly clustered regions around this value of p. For low values of p the standard deviation is small and the clustering coefficient is high, so there are many big clusters, for high values of p de standard devation is again small but the clustering coefficient is low, so there are no big clusters anymore. Hence the phase transition can be linked to the clustering coefficient, with an intermediate phase in the small-world domain where the clustering coefficient decreases as a function of p and the ordering parameter increases as a function of p, coupled to a large distribution in local clustering coefficients.

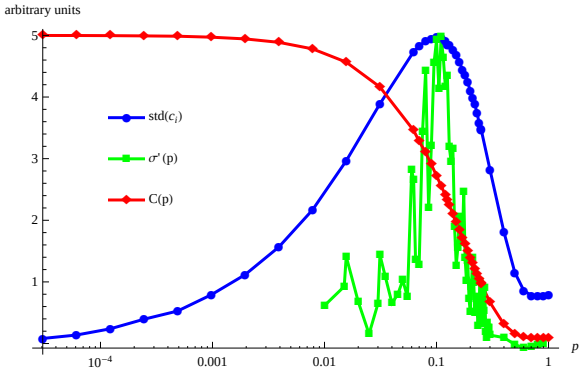

Figure 7: The standard deviation of the local clustering coefficient, the derivative of σ and the clustering coefficient C(p) all shown together.

3 Conclusion and discussion

We investigated behavior of small-world networks, constructed by an algorithm designed by Watts and Strogatz in the late 90's. Small-world networks typically have a large clustering coefficient but a small average path length. Small-world networks can be used to model disease spreading, which we did using the SIR and the SIRS model. With the SIR model the disease spreading was investigated, which turns

out to scale with the average path length. With the SIRS model we studied the occurence of state synchronization as a function of p. We defined an order parameter and found the phase transition from unsynchronized to synchronized states to be at $p \approx 0.15$, which coincides with the region of rapidly decreasing clustering coefficient and high values of the standard deviation of the local clustering coefficient. Hence this transition region seems to be corresponding to local high clustering and local low clustering regions across the network.

The validity of these kind of models is still an ongoing debate. Especially when the nodes are interpreted as humans the question arises whether static nodes (with static edges) represent a realistic model, since in real social networks connections change rapidly and constantly [3]. It depends on the infection rate of the disease whether or not the static model is realistic. For low infection rates, people will not get infected after one isolated contact but will more probably be infected by their regular contacts, hence a model where each edge can be given a weight (according to the amount of social interaction between two persons) would be a realistic in such a case. For highly infectious diseases the single contacts (for instance in public transport) become very important. Another important aspect is that the parameters that are used in the model might not be constant, because the disease may evolve or the nodes might start behaving differently. Especially when investigating disease spreading on social networks the behaviour of humans should be taken into account, because people might start taking precautions after the disease has spread to a significant part of the population for example. Furthermore the analytical background for these types of networks which are neither regular nor random is still very poor and much theoretical work has yet to be done. There are examples of very simple onedimensional directed percolation models (which is basically a simplification of a network one dimensional line) obeying very simple rules which are not yet analytically solvable, so there is still a long way to go in the theoretical field of this subject.

4 Methods and learning process

In this section I describe the tools I used and the knowledge I have obtained via this project. As can be seen in the attached source code I have programmed my code entirely in PYTHON, since that was one of my learning goals. In PYTHON I used the package IGRAPH, which provides functions to make graphs and also provides functions to plot graphs and to animate graphs. Most of the figures in this report however are made in Mathematica, which was a tool I used before and now I have refreshed my knowledge concerning this program, using it mostly for my graphical output. Almost all of the editing and coding (including writing this report) is done in EMACS, which I started using at the beginning of the project.
As described in the report I used stochastic models to describe disease spreading. It is also possible to write down a system of coupled ODE's and solve this system using some particular solvers, like Runge-Kutta methods. However, in the early stages of the project I choose a topic about enzyme kinetics, which involved solving a system of coupled ODE's, which turned out to be very simple to program and not add much to my programming knowledge. I found this topic to be very interesting and insightful and it was fun to learn PYTHON and some stochastic modelling. I am looking forward to the next obligatory course of the programme, Stochastic Simulation, which will hopefully give a more broad overview of the field of stochastics.
I choose this particular topic because I have done some very simple simulations on directed percolation disease models before and I found that to be very interesting. My main interests at the moment lies in the area of complex dynamical systems, where this subject can also be classified under, so this project was an excellent way in getting more familiar with these kind of systems.
My starting point for the literature was the paper published by Watts and Strogatz about the dynamics of small-world networks. Than I looked at papers using the small-world algorithm to construct such networks and do some analysis on them. Next to the papers cited in the bibliography I used various Wikipedia pages on topics related to small-world networks. There are many papers on this topic and most of them are still fairly recent, showing that this is an active research area.

References

[1] Watts, Duncan J.; Strogatz, Steven H.., *Collective dynamics of 'small-world' networks*, Nature, 6/4/98, Vol. 393 Issue 6684, p440

[2] Kuperman, M.; Abramson, G.,*Small World Effect in an Epidemiological Model*, Phys. Rev. Letters, 3/2001, Vol. 86, p2909-2912, [arXiv:nlin/0010012]

[3] Li, Sheng; Meng, Meng; Ma, Hongru,*Epidemic Spreading in Dynamic Small World Networks*, [arXiv:nlin/0411017]

[4] Bollabas, B., *Random Graphs*, Academic, London, 1985.

[5] Y. Kuramoto, *Chemical Oscillations, Waves, and Turbu- lence*, Springer, Berlin, 1984.

5 Appendix: code

In this section the code used for the project is given:
The following piece of code computes basic properties using the Watts-Strogatz algorithm, namely the clustering coefficient C(i), the average path length L(p) and the standard deviation of the local clustering coefficient:

```
from igraph import *
import random
from decimal import Decimal
import numpy
from numpy import std

#Open file for writing
f = open('/home/jhofman/Desktop/Project/test.txt','a')

#Function for calculating global clustering coefficient
def connectivity(g):

    #Gives a (symmetric) matrix of connections between nodes
    matrix = g.get_adjacency()
    conn = 0
    for i in range(len(g.vs)):
        p = 0

        #List the neighbors of node i
        list = g.neighbors(i)
        if len(list)>1:

            #Maximum possible number of connections
            max_p = len(list)*(len(list)-1.)/2.

            #Number of connections among neighbors using the adjacency
                matrix
            for j in range(len(list)):
                for k in range(j+1,len(list)):
                    if matrix[list[j]][list[k]]==1:
                        p +=1

            #add ratio to average connectivity
            conn += p/max_p

        else: conn += 1
```

```
    return conn/len(g.vs)

#Parameters
n = 100 #number of realizations
c0,l0 = 0,0
stdd = 0

#loop over realizations
for m in range(n):

    #Initialize graph with rewiring probability 0 and compute transitivity
        (=connectivity) and average path length
    g0 = Graph.Watts_Strogatz(1,1000,5,0)
    c0 += connectivity(g0)
    l0 += g0.average_path_length()
    trans = GraphBase.transitivity_local_undirected(g0,g0.vs)

    #standarddeviation of transitivity
    stdd += numpy.std(trans)

#Print and write averages over realizations
print "p = ",0," Relative average path length = ",l0/l0," Relative
    connectivity =",c0/c0," std = ",stdd/float(n)
f.write('p = '+str(0)+' l(p) = '+str(l0/l0)+' c(p) = '+str(c0/c0)+' '+str(
    Decimal(stdd/float(n)))+'\n')

#Loop over values in p-lists
plist = [2**i for i in range(-15,1)]
p2list = range(0.07,0.25,0.01)
p2list.append(range(0.3,1,0.1))
p2list.extend(plist)
for p in p2list:

    stdd = 0
    c,l = 0,0
    m = 0

    #loop ove realizations
    while m < n:

        #Initialize graph and calculate properties
        g = Graph.Watts_Strogatz(1,1000,5,p)
        trans = GraphBase.transitivity_local_undirected(g,g.vs)

        #Make sure graph is connected
        if not math.isnan(numpy.std(trans)):
            c += connectivity(g)
            l += g.average_path_length()
            stdd += numpy.std(trans)
            m += 1

    #Average properties over realizations and normalize over p = 0
        properties
    print "p = ",Decimal(p)," Relative average path length = ",l/float(l0),
        " Relative connectivity =",c/float(c0)," std = ",stdd/float(n)
    f.write('p = '+str(Decimal(p))+' l(p) = '+str(l/float(l0))+' c(p) = '+
```

```
        str(c/float(c0))+' '+str(Decimal(stdd/float(n)))+'\n')

f.close()
```

The following piece of code implements the SIR model on a network generated with the Watts-Strogatz algorithm and computes the spreading times $r_{1/2}$ and r_{total} and also counts state (i.e. S, I or R):

```
from igraph import *
import random
from decimal import Decimal

#Open files for writing
f1 = file('/home/jhofman/Desktop/Project/rhalve.txt','a')
f2 = file('/home/jhofman/Desktop/Project/rtotal.txt','a')

#Function for setting state and counting states
def count():
    for i in range(0,N):

        #Append to infected list and add time step
        if g.vs[i]["time"] in range(t_inf,t_rec):
            infected.append(i)
            g.vs[i]["time"] += 1

        #Append to recovered list and add time step
        elif g.vs[i]["time"] == t_rec:
            recover.append(i)

    #Return S,I and R counts and the infected+recovered portion
    return 1000-(len(infected)+len(recover)),len(infected), len(recover),(
        len(infected)+len(recover))/1000.

#Function doing the infection step
def infect():
    for i in range(len(infected)):

        #Calculate neighbors of infected node
        nb = g.neighbors(infected[i])
        for j in range(len(nb)):
            if g.vs[nb[j]]["time"] == 0:
                r = random.random()

                #Infect with probability p_inf
                if r <= p_inf:
                    g.vs[nb[j]]["time"] = t_inf
    return None

#Parameters
p_inf = 1.0 #Probability of infection
t_max = 1000 #Maximum time of run
n = 100  #number of realizations
N = 1000 #number of nodes

t_susc = 0
t_inf = 1 #infection time 5-1 = 4
t_rec = 5
```

```
plist = [2**i for i in range(-15,1)] #list of p-values
for p in plist:
    r_total_sum,r_half_sum = 0.,0.

    #loop over n realizations
    for m in range(n):
        t,check,r_half = 0,0,0

        #Initialize graph and simplify (remove double loops)
        g = Graph.Watts_Strogatz(1,1000,5,p)
        g = Graph.simplify(g)

        #Set all nodes to susceptible state except random seed
        for i in range(N):
            g.vs[i]["time"] = 0
        seed = random.randint(0,N-1)
        g.vs[seed]["time"] = 1

        #Iteration step
        while r_half < 1.0:
            infected = []
            recover = []

            (susc,inf,rec,r_half) = count()
            infect()

            #record time to infect halve the network
            if r_half > 0.5 and check == 0:
                half_time = t
                r_half_sum += t
                check = 1

            #record time to infect the whole network
            if r_half == 1.:
                r_total_sum += t

            #if time reaches t_max while r_half<1 some nodes have not been
                infected, so generate new graph
            if t==t_max:
                m -= 1
                r_half_sum -= half_time
                break

            t += 1

    #write averaged half and whole infection rates to file
    r_half_sum /= n
    r_total_sum /= n
    print "t_rec = ",t_rec-1," p = ",p," r_half_sum = ",r_half_sum
    print "t_rec = ",t_rec-1," p = ",p," r_total_sum = ",r_total_sum
    f1.write(str(t_rec-1)+' '+str(Decimal(p))+' '+str(r_half_sum)+'\n')
    f2.write(str(t_rec-1)+' '+str(Decimal(p))+' '+str(r_total_sum)+'\n')

f1.close()
f2.close()
```

The last piece of code implements the SIRS model on a network generated with the Watts-Strogatz algorithm and computes the ordering parameter and also counts states (i.e. S, I or R):

```
import random
from decimal import Decimal
from math import pi,exp,log
import cmath
from igraph import *

#Function making list of infected and recovered individuals and order
    parameter
def count_order():
    sigma = 0 #order parameter
    for i in range(N):

        #append if node is in infected state and add time step
        if g.vs[i]["time"] in range(t_inf,t_rec):
            infected.append(i)
            if t>=t_cut:
                phi = 2.*pi*(float(g.vs[i]["time"]-1.))/float(t_reset-1)
                sigma += cmath.exp(1j*phi)
            g.vs[i]["time"] += 1

        #append if node is in recovered state and add time step
        elif g.vs[i]["time"] in range(t_rec,t_reset):
            recover.append(i)
            if t>=t_cut:
                phi = 2.*pi*(float(g.vs[i]["time"]-1.))/float(t_reset-1)
                sigma += cmath.exp(1j*phi)
            g.vs[i]["time"] += 1

    #return S,I and R nodes and sigma
    if (len(infected)+len(recover))!=0:
        return N-len(infected)-len(recover),len(infected),len(recover),abs(
            sigma/(len(infected)+len(recover)))
    else:
        return N-len(infected)-len(recover),len(infected),len(recover),0

#Function infect neighbours of infected nodes
def infect():
    for i in range(len(infected)):

        #calculate neighbors of node
        nb = g.neighbors(infected[i])
        for j in range(len(nb)):
            if g.vs[nb[j]]["time"] == 0:
                r = random.random()

                #infect neighbor with probability p_inf
                if r < p_inf:
                    g.vs[nb[j]]["time"] = t_inf
    return None

#Parameters
p_inf = 0.1 #probability of infection
t_max = 3000 #maximum run time
```

```
t_cut = 1000 #cut off time for calculating sigma
N = 1000 #size of graph
K = 5 #number of connections per node for p = 0
real = 50 #number of graph realizations per p-value

#Time parameters for nodes
t_susc = 0
t_inf = 1 #infected steps 9-1 = 8
t_rec = 9 #recovered steps 17-9 = 8
t_reset =17 #total cycle time

#Open file for writing
f1 = file('/home/jhofman/Desktop/Project/sigma_test_test.txt','a')

#Construction of list for p-values
plist1 = [i*0.005 for i in range(1,20)]
plist2 = [2**i for i in range(-15,1)]
plist3 = [i/20. for i in range(0,21)]
plist2.extend(plist3)
plist1.extend(plist2)

#Loop over all p
for p in plist1:

    sigma_mean = 0

    #loop over number of realizations
    for m in range(real):

        sigma = 0

        #creates graph with Watts-Strogatz algorithm and remove loops
        g = Graph.Watts_Strogatz(1,N,K,p)
        g = Graph.simplify(g)

        #set all nodes to susceptible state except for the random seed
        for i in range(N):
            g.vs[i]["time"] = 0
        seed = random.randint(0,N-1)
        g.vs[seed]["time"] = 1

        #loop over t
        t = 0

        while t < t_max:

            infected = []
            recover = []
            (susc,inf,rec,sig) = count_order()
            sigma += sig #sum up all the order parameters for all time
                steps
            infect()

            #put some recovered nodes back in susceptible state
            for i in range(len(recover)):
```

```
                g.vs[recover[i]]["time"] = (g.vs[recover[i]]["time"]%
                    t_reset)

            t += 1

        #calculates the order parameter over time t_max-t_cut
        sigma_mean += sigma/float(t_max-t_cut)
        print "p = ",p," m = ",m," sigma = ",sigma_mean

    #save the order parameter as a mean over all realizations
    f1.write(str(Decimal(p))+' '+str(sigma_mean/float(real))+'\n')
    print "p = ",p," sigma_mean_mean = ",sigma_mean/float(real)

f1.close()
```