Network Science Project 2 (Version 2)

A line of code in part 1.1) has been modified, see comment below

Name: Joshua Tegegne

CID: 01705625

Please enter your name and 8-digit college ID in the cell above

Part 1

1) Analyze the function *Graph1* in the cell below. Place your discussion in the cell below the function where indicated.

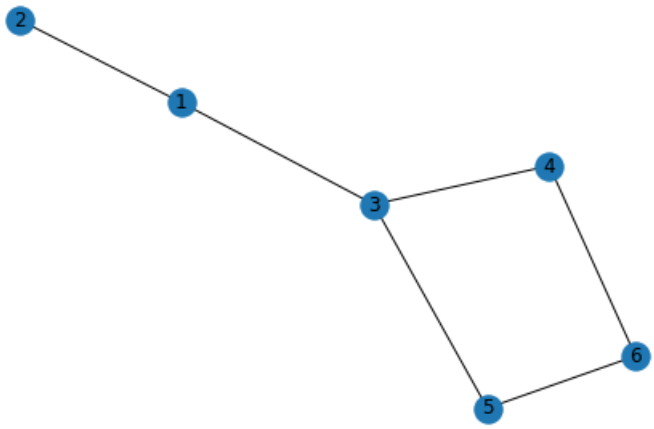
In [134]:

```
def Graph1(G,x):
    """
    Input:
    G: simple, undirected NetworkX graph with nodes numbered from 1 to N
    x: an integer, 1<=x<=N

    """


    L1 = [0 for i in G.nodes()]
    L2 = [-1 for i in G.nodes()]
    #L1 is a list where the i-1'th value is 1 if there is a path from node i to the source node x,
and is 0 if no such
    #path exists. L2 is a list where the i-1'th is the distance between node i and x, and is -1 if
there is no path.
    # This is why we initialise L1 at 0 and L2 at -1 for every element, then once paths are found
the values may change.
    L = [list(G.nodes()),L1,L2,L1.copy(),L1.copy()]
    # This list makes it easier to recall L1, L2, and so on, i.e list L1 = L[1]
    L[1][x-1] = 1
    L[2][x-1] = 0
    L[3][x-1] = 1
    L[4][x-1] = G.degree(x)

    M = [x]
    #list of nodes to iterate through
    ind = 0
    while ind<len(M):
        n = M[ind]
        ind = ind+1
        #In each loop we add neighbours of the node n, add them to list M, then in the next loop w
e iterate
        # through the neighbours of the node next in the list
        #Once ind = len(M), we will have iterated through all nodes reachable from the source node
and the while loop ends
        for i in G.adj[n]:
            #iterating through neighbours of node n
            if L[1][i-1]==0:
                L[1][i-1]=1
                L[2][i-1]=L[2][n-1]+1
                L[3][i-1] = L[3][n-1]
                L[4][i-1] = L[4][n-1] + G.degree(i)*L[3][n-1]
                M.append(i)
                #If the neighbour of node n we are on hasn't previously been discovered, we set L1
i-1] = 1 (we have
                # discovered  the node we are on) and set its distance to being 1 + the distance
of  node n
                # L[3] represents the number of shortest paths from a node to the source, so if a n
eighbour is discovered
                # for the first time it is set to being the same as node n

                #L4[i] is the sum of the degrees of all the nodes across all shortest paths from no
de i-1 to the source node
```

```
                #If a node is being discovered for the first time L4[i] then the only shortest path
s we know of that are from node i-1
                # to x would be an extension of every shortest path from node i to x. So the value
of L4 for node i compared to node n
                # should be bigger by G.degree(i)*L[3][n-1] because node i is added is on each of t
he new shortest paths and there
                # are L3[n-1] shortest paths, so we multiply L3[n-1] by the degree of node i
            elif L[2][i-1]==L[2][n-1]+1:
                L[3][i-1] = L[3][i-1]+L[3][n-1]
                L[4][i-1] = L[4][i-1] + L[4][n-1]+G.degree(i)*L[3][n-1]
                # If the neighbour of node n has already been discovered and its distance from the
source node is 1
                # greater than node n, then new paths from the source node to the neighbour have be
en discovered so this
                # increases by the number of shortest paths that node n has
                # Similarly for L4 we increase by the same factor as in the if loop
                # as we have found new shortest paths from node i to x

    return L

#example
Y = nx.Graph([(1, 2), (1, 3), (3, 4), (3, 5), (4, 6), (5, 6)])
nx.draw(Y, with_labels = "TRUE")
Graph1(Y,2)
```

Out[134]:

```
[[1, 2, 3, 4, 5, 6],
 [1, 1, 1, 1, 1, 1],
 [1, 0, 2, 3, 3, 4],
 [1, 1, 1, 1, 1, 2],
 [3, 1, 6, 8, 8, 20]]
```



Graph1 Discussion

We already know breadth first search is of O(L+N), where L is number of edges and N is the number of nodes. The only way the Graph1 code would be longer is because it has to perform more operations inside the if loop. However, assigning variables more and appending to a list if of O(1) still so the time taken to run Graph1 would be of O(L+N). This code is efficient because of how we iterate over the nodes, we check through neighbors which allows us to change the values of L step by step. Graph1 is efficient, because as we aren't repeating the same calculations- once we find there is a new shortest path to a node we use this to change our lists accordingly.

In [5]:

```
#The number of for loops is of O(KN), where K is the maximum degree , because we at worst, iterate
through
#the neighbours of every node. Each for loop performs at most 1 comparison, 4 assignments, 1 appen
d, so the code time
#is O(KN)

N = len(G.nodes)
import numpy as np

Dlink = {}
for i,e in enumerate(G.edges()):
```

```
    if e[0] in Dlink:
        Dlink[e[0]][e[1]]=i+1
    else:
        Dlink[e[0]]={e[1]:i+1}

    if e[1] in Dlink:
        Dlink[e[1]][e[0]]=i+1
    else:
        Dlink[e[1]]={e[0]:i+1}
```

2) Complete *Graph2* below to compute source edge centrality. Place your discussion in the cell below the function where indicated.

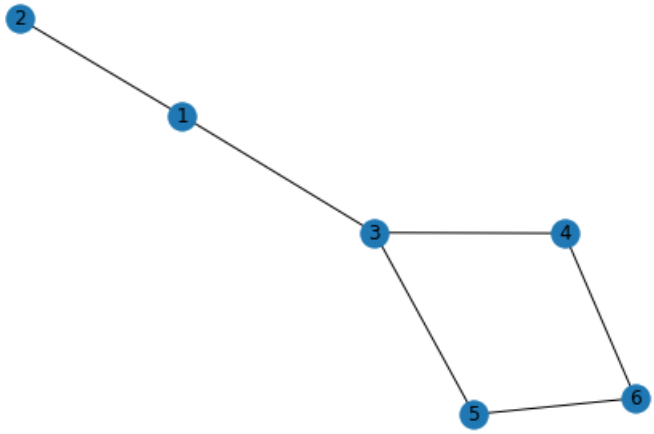
In [137]:

```
def Graph2(G,a,gamma0=1):
    """
    Input:
    G: simple, connected, undirected NetworkX graph with N nodes numbered from 1 to N and L links
    a: source node
    gamma0: parameter used in edge centrality calculation

    Output: Lf: a L-element list. The ith element of the list contains the source edge centrality
    for the i+1th edge in G.edges()
    """



    # Create dictionary Dlink mapping node-pairs to link number
    # Dlink[i][j] will give a number between 1 and L corresponding to
    # the label for link i-j
    Dlink = {}
    for i,e in enumerate(G.edges()):
        if e[0] in Dlink:
            Dlink[e[0]][e[1]]=i+1
        else:
            Dlink[e[0]]={e[1]:i+1}

        if e[1] in Dlink:
            Dlink[e[1]][e[0]]=i+1
        else:
            Dlink[e[1]]={e[0]:i+1}
    #--------------------------------
    L = len(G.edges)
    N = len(G.nodes)
    Lf = [] #modify as needed

    edgelist = [list(e) for e in G.edges]
    # list of edges, i'th element is L-1'th edge

    LLa = Graph1(G,a)

    for i in range(1, L+1):
        # computing source edge centrality with i'th link
        central_list = []
        edge = edgelist[i-1]
        for b in range(1, N+1):
            #for b from 1 to n, we will append the term inside the sum to cental_list, then take t
he sum of the list at the end
            LLb = Graph1(G,b)

            k = 1
            if edge[1] == b:
                k = gamma0
            #sets gamma to y_Bb if beta = b, otherwise it stays as 1

            if LLa[2][edge[0]-1] < LLa[2][edge[1]-1]:
                #if first node of given edge is closer to the source, we can work out the the numbe
r of
                #shortest paths from node a to b that pass through our edge, by multiplying the nur
ber of shortest paths from node a to
                # node edge[0] by the number of shortest paths from node edge[1]
                if LLa[2][b-1] == 1 + LLa[2][edge[0]-1] + LLb[2][edge[1]-1]:
                    #However, these are only valid shortest paths if they can be joined together to
give us the same distance as the
                    #shortest path from node a to node b
                    central_list.append(k*LLa[3][edge[0]-1]*LLb[3][edge[1]-1]/LLa[3][b-1])
```

```
            else:
                if LLa[2][b-1] == 1 + LLa[2][edge[1]-1] + LLb[2][edge[0]-1]:
                    #Similar to before except shortest paths are from node a to node edge[1] to no
e edge[0] to node b
                    central_list.append(k*LLa[3][edge[1]-1]*LLb[3][edge[0]-1]/LLa[3][b-1])

        Lf.append(sum(central_list))

    return Lf, edgelist
    # output Lf and our list of edges so i'th centrality corresponds to i'th element in edgelist


#example
nx.draw(Y, with_labels = "TRUE")
Graph2(Y,3,0.5)
```

Out[137]:

```
([0.5, 2.0, 1.0, 1.0, 0.25, 0.25],
 [[1, 2], [1, 3], [3, 4], [3, 5], [4, 6], [5, 6]])
```



The difficult part of the code is computing m_alpha_beta(a,b), the number of shortest paths from node a to b passing through edge alpha-beta. I talked about my algorithm for computing this briefly but this is how I computed it: 1) Calculate if node alpha or beta is closest to node a (if they are equal distance it doesnt matter which path we choose) 2) If closer to node alpha, we see the length of the shortest paths from node a to alpha and them to the length of the shortest paths from beta to b. The composition of these two shortest paths (with the path from alpha to beta) will consist of a shortest path from a to b if the sum of the distances are equal to the sum of the distance between node a and b. If they dont't add up, m_alpha_beta(a,b) = 0 3) If the distances do add up, m_alpha_beta(a,b)= p_a_alpha * p_beta_b , where p_i_j is the number of shortest paths between i and j. 4) Note that if node a is closest to beta instead, process is essentially the same except we 'swap' alpha and beta.

Double for loop has LN iterations, recalling our function Graph1 twice is of O(L+N), everything else inside the loop is O(1) , so the whole code time is O(LN(L+N)). This code is good for small graphs but would take long for larger graphs.

Part 2

In [64]:

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from scipy.integrate import odeint
#you may also use itertools and scipy as needed. Do not import/use other modules without explicit
permission.

#Run this cell to create load graph to be analyzed
elist = np.loadtxt('project2.dat',dtype=int)
N = elist.max()
G = nx.Graph()
G.add_nodes_from(range(1,N+1))
G.add_edges_from(elist)
```

2.1 (a) and (b): Complete SInct and RHS below

2.1 (a) and (b): Complete SInet and RHS below

In [65]:

```
def SInet(G,i0=1,x0=0.001,beta=1.0,delta=1.0,omega=80.0,tf=0.4,Nt=1000):
    """
    Questions 2.1 and 2.2
    Simulate naive network-SI model with time-periodic transmission

    Input:
    G: Networkx graph (simple, connected, undirected with nodes numbered from 1 to N)
    i0: node which is initially infected
    x0: magnitude of initial condition
    beta,delta, omega: model parameters
    tf,Nt: Solutions are computed at Nt time steps from t=0 to t=tf (see code below)

    Output:
    tarray: Nt+1-element array of times at which solution is output
    xarray: Nt+1 x N Numpy array containing <x> across network nodes at
                each time step.
    """

    N = G.number_of_nodes()
    xarray = np.zeros((Nt+1,N))
    xarray[0,i0-1] = x0 #initial condition
    tarray = np.linspace(0,tf,Nt+1) #times at which solution should be returned
    B = nx.adjacency_matrix(G)


    #i j 'th element of adjacency matrix is 1 if nodes i and j are connected, and is 0 otherwise

    def RHS(x,t):
        """Compute RHS of model at time t
        input: x should be a size N array corresponding to <x(t)> for all nodes
        output: dxdt, also a size N array corresponding to d<x>/dt for all nodes
        """
        dxdt = beta * (1 + delta*np.cos(omega*t))*(B@x)*(1-x)
        return dxdt

#    x = xarray[0,:]
 #   for b in range(1,Nt+1):
#        x = RHS(x,tarray[b])

#        xarray[b,:] = x
    xarray = odeint(RHS, xarray[0], tarray)
        # start with inital conditions, on each iteration calculate dxdt and update corresponding
row in xarray




    return tarray,xarray
```

2.1 (c) We calculate the adjacency matrix first so to look at how long the code will take to run, we need to look at the for loops. The first for loop performs an append, and the second has an assignment, so it is as quick as we can make it. To increase accuracy we can increase the number of time steps across our interval.

2.2

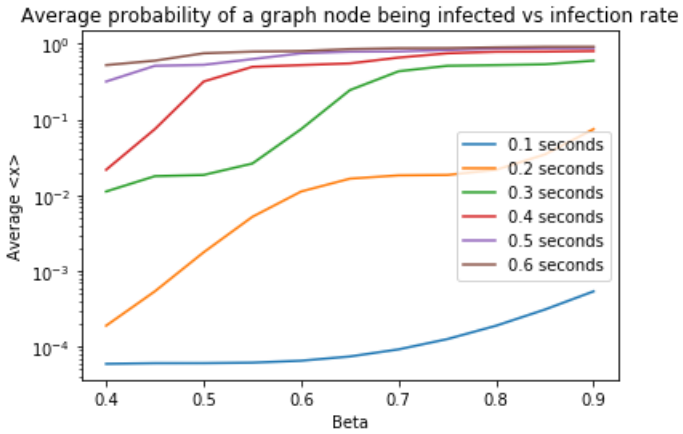
In [128]:

```
k = np.array(nx.degree(G),dtype=int)[:,1]
i0 = np.argmax(k)+1

x0 = 0.001
delta = 1
beta_l = np.linspace(0.4,0.9,11)
average_s1 = np.array([])
average_s2 = np.array([])
average_s3 = np.array([])
average_s4 = np.array([])
average_s5 = np.array([])
average_s6 = np.array([])

#initialise our arrays
```

```
#Initialise our arrays


for i in beta_l:
    data = SInet(G, i0, x0, i, delta, 80*i, 0.5,61)[-1]
    average_s1 = np.append(average_s1,np.mean(data[10,:]))
    average_s2 = np.append(average_s2,np.mean(data[20,:]))
    average_s3 = np.append(average_s3,np.mean(data[30,:]))
    average_s4 = np.append(average_s4,np.mean(data[40,:]))
    average_s5 = np.append(average_s5,np.mean(data[50,:]))
    average_s6 = np.append(average_s6,np.mean(data[60,:]))
```

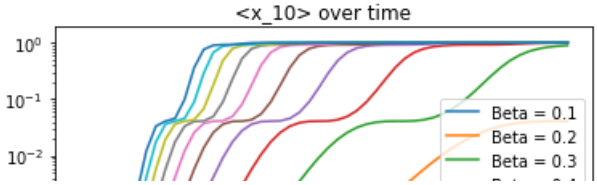
In [130]:

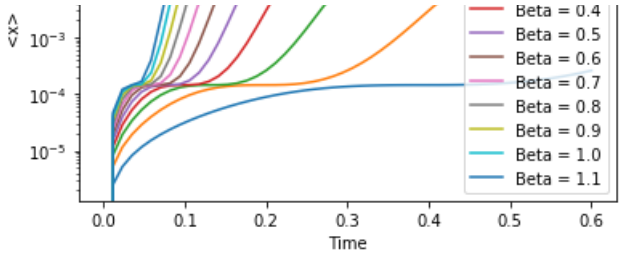
```
ax = plt.subplot(111)
ax.plot(beta_l,average_s1, label = '0.1 seconds')
ax.plot(beta_l,average_s2, label = '0.2 seconds')
ax.plot(beta_l,average_s3, label = '0.3 seconds')
ax.plot(beta_l,average_s4, label = '0.4 seconds')
ax.plot(beta_l,average_s5, label = '0.5 seconds')
ax.plot(beta_l,average_s6, label = '0.6 seconds')

ax.set_yticklabels(["{:.2e}".format(t) for t in ax.get_yticks()])
ax.set_xlabel("Beta")
ax.set_ylabel("Average <x>")
ax.set_yscale("log")
ax.set_title("Average probability of a graph node being infected vs infection rate")
plt.legend(loc='best')
plt.show()
```



In [115]:

```
ax = plt.subplot(111)
beta_l = np.linspace(0.1,1.1,11)
betas2 = ["Beta =" + " " + str(round(i,2)) for i in beta_l]

for i in range(len(beta_l)):
    data_ = SInet(G, i0, x0, beta_l[i], delta, 80*beta_l[i], 0.6,51)
    ax.plot(data_[0], data_[1][:,9], label = betas2[i])

ax.set_yticklabels(["{:.2e}".format(t) for t in ax.get_yticks()])
ax.set_xlabel("Time")
ax.set_ylabel("<x>")
ax.set_yscale("log")
ax.set_title("<x_10> over time")
plt.legend(loc='best')
plt.show()
```





For the first graph, it seems like as beta increases, the average probability of a node being infected increases. There seems to be a consistent non-linear pattern to this. This probability also increases as time increases as expected.

For the second graph, we plot the evolution of for node 10 over time for different values of beta. We can see that at a given time, if beta is higher then the value of is always higher. For any value of beta the pattern of evolution seems to be the same, it repeatedly increases quickly then slows down in growth. For lower values of beta the growth slows down. For higher values of beta reaches 1 fastest and eventually all curves reach this point.

From this data we can say that the higher beta is, the faster the spread of the disease over time. Note that if is higher for higher values of beta, then at a given time, on average, more nodes will be infected.

2.3

In [138]:

```
comlist = list(nx.algorithms.community.greedy_modularity_communities(G))
model = SInet(G, i0, x0, 0.5, delta, 40, 0.03,4)[-1]
print ("There are",len(comlist),"communities")

print ("Node",i0,"has highest degree, and belongs to community 2")
```

```
There are 4 communities
Node 679 has highest degree, and belongs to community 2
```

In [139]:

```
com = [np.array([]) for n in range(12)]
for i in comlist[0]:
    com[0] = np.append(com[0], model[1,i-1])
    com[1] = np.append(com[1], model[2,i-1])
    com[2] = np.append(com[2], model[3,i-1])
for i in comlist[1]:
    com[3] = np.append(com[3], model[1,i-1])
    com[4] = np.append(com[4], model[2,i-1])
    com[5] = np.append(com[5], model[3,i-1])
for i in comlist[2]:
    com[6] = np.append(com[6], model[1,i-1])
    com[7] = np.append(com[7], model[2,i-1])
    com[8] = np.append(com[8], model[3,i-1])
for i in comlist[3]:
    com[9] = np.append(com[9], model[1,i-1])
    com[10] = np.append(com[10], model[2,i-1])
    com[11] = np.append(com[11], model[3,i-1])
#com has 12 lists, 4 for each time we plot at. These 4 lists contain <x> for each community
```

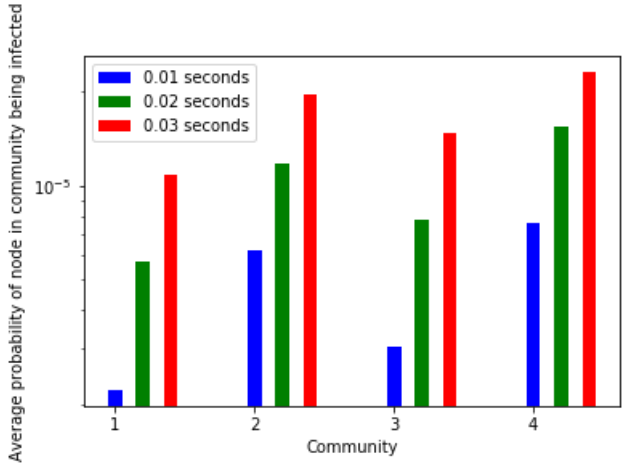
In [140]:

```
meanlist = [np.mean(i) for i in com]
meanlist = np.array(meanlist)
```

In [141]:

```
ax = plt.subplot(111)
ax.bar([1,2,3,4], meanlist[[0,3,6,9]], width=0.1, color='b', align='center', label = "0.01 seconds"
)
ax.bar([1.2,2.2,3.2,4.2], meanlist[[1,4,7,10]], width=0.1, color='g', align='center', label="0.02 s
econds")
ax.bar([1.4,2.4,3.4,4.4], meanlist[[2,5,8,11]], width=0.1, color='r', align='center',label="0.03 se
conds")
ax.set_yscale("log")
```

```
ax.set_xticks([1,2,3,4])
ax.set_xlabel("Community")
ax.set_ylabel("Average probability of node in community being infected")
plt.legend(loc='best')
plt.show()
```



First I find the partition which maximises the modularity, this gives us 4 communities. The plot shows how the average value of across nodes in a given community evolves over the first 0.03 seconds. For any community, the average increases over time. Except at the community 4 at t = 0.01, at the plotted times community 2, then 3,1, 4 has the highest average . Node 679 has the highest degree in the graph and is where the outbreak begins, it is in the 2nd community which explains why it always has the highest average .

In [ ]:

```
```