Humza Salman, mhs180007

In [1]:
```
import networkx as nx
import sys
sys.path.append('../d3networkx/')
import d3networkx as d3nx
from d3graph import D3Graph, D3DiGraph
from numpy import *
from numpy.linalg import eig,norm
import matplotlib.pyplot as plt
plt.ioff()
from time import time
from copy import deepcopy
import asyncio
import colorsys
```

Section 6.13 Diffusion

The following functions assist in coloring nodes based on the value of the state at each node. You will only need to use `color_by_value` - the rest of the functions are helper functions that are used behind the scenes.

In [2]:
```
def RGBToHTMLColor(rgb_tuple):
    """ convert an (R, G, B) tuple to #RRGGBB """
    hexcolor = '#%02x%02x%02x' % rgb_tuple
    # that's it! '%02x' means zero-padded, 2-digit hex values
    return hexcolor

def HTMLColorToRGB(colorstring):
    """ convert #RRGGBB to an (R, G, B) tuple """
    colorstring = colorstring.strip()
    if colorstring[0] == '#': colorstring = colorstring[1:]
    if len(colorstring) != 6:
        raise ValueError("input #%s is not in #RRGGBB format" % colorstring)
    r, g, b = colorstring[:2], colorstring[2:4], colorstring[4:]
    r, g, b = [int(n, 16) for n in (r, g, b)]
    return (r, g, b)

def color_interp(color1,color2,v,m=0,M=1):
    c1 = array(HTMLColorToRGB(color1))
    c2 = array(HTMLColorToRGB(color2))
    if v > M:
        c = tuple(c2)
    elif v <= m:
        c = tuple(c1)
    else:
        c = tuple( c1 + (c2-c1)/(M-m)*(v-m) ) # linear interpolation of color
        #c = tuple( rint( c1 + (c2-c1)*(1 - exp(-2*(v-m)/(M-m))) ) ) # logistic interp
        c = (int(c[0]),int(c[1]),int(c[2]))
    return RGBToHTMLColor(c)

def color_by_value(d3,G,x,color1='#FFFFFF',color2='#F57878'):  #color1='#77BEF5'
```

```
    interactive = d3.interactive
    d3.set_interactive(False)
    m = 0
    M = 1#0.5
    for n in G.nodes():
        d3.stylize_node(n, d3nx.node_style(size=5,stroke='#494949',fill=color_interp(
    d3.update()
    d3.set_interactive(interactive)
```

Load the network

In [8]:
```
DG = D3Graph( nx.read_weighted_edgelist('dolphins.edgelist',create_using=nx.Graph) )
TG = D3Graph( nx.read_weighted_edgelist('train.edgelist',create_using=nx.Graph) )
MG = D3Graph( nx.read_weighted_edgelist('macaque.edgelist',create_using=nx.Graph) )
```

In [9]:
```
d3 = await d3nx.create_d3nx_visualizer(interactive=False,
                                       node_dstyle=d3nx.node_style(size=5,fill='#FFFF
                                       edge_dstyle=d3nx.edge_style(stroke_width=1.25))
```

```
websocket server started...visualizer connected...networkx connected...
```

In [10]:
```
def reset_visualizer(d3, G):
    d3.clear()
    d3.set_graph(G)
    d3.update()
    d3.set_interactive(True)
```

Diffusion

In [11]:
```
async def diffusion(G, dt=0.02, T=6, C=1, initial_value=1, visualize=False, wait_to_vi

    if visualize:
        reset_visualizer(d3, G)

    time = linspace(0,T,int(T/dt)) # the array of time points spaced by dt

    L = array(nx.laplacian_matrix(G).toarray().T)
    N = G.number_of_nodes()

    # compute equilibrium state
    w, v = eig(L)
    low_index = where(w == min(w))[0][0]
    eq = w[low_index] * v[:, low_index]

    x = zeros(N) # initialize N size vector of 0s
    x[0] = initial_value # initialize first value

    if visualize:
        color_by_value(d3,G,x)

    diff = [0] * len(time) # keep track of cosine difference

    for i,t in enumerate(time):
        # at each time step update the value of x!
        x += (C * -L @ x) * dt # L = D - A in networkx, so dx/dt -cLx = 0 => dx = cLx
```

```
        diff[i] = cos(norm(eq)) - cos(norm(x)) # take norm of difference between equil

        if visualize:
            color_by_value(d3,G,x)  # update the visualizer

        if wait_to_visualize:
            await asyncio.sleep(0.1) # wait a little bit so the visualizer has time to

    return time, diff
```

Simulating diffusion...

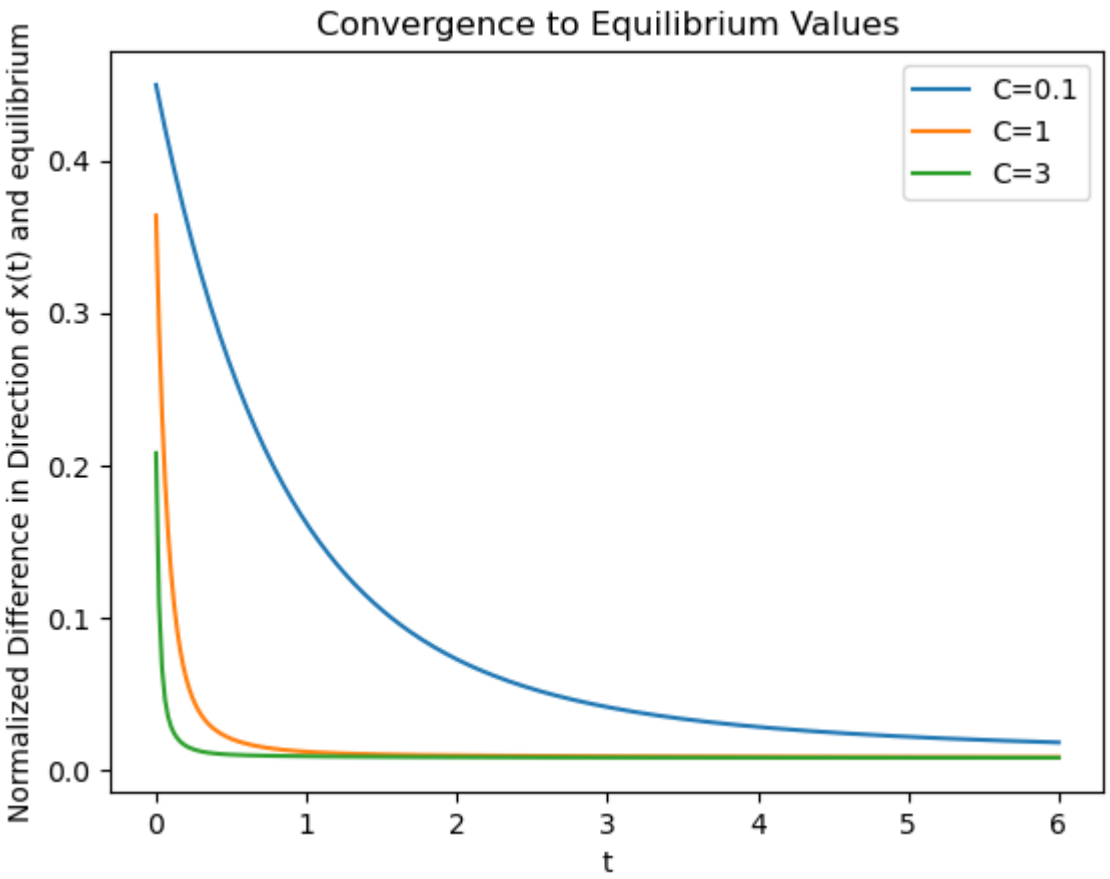
In [15]:
```
async def run_diffusion(G, Clow, Cmid, Chigh, init_value=1):
    time_Clow, diff_Clow = await diffusion(G, C=Clow, initial_value=init_value)
    time_Cmid, diff_Cmid = await diffusion(G, C=Cmid, initial_value=init_value)
    time_Chigh, diff_Chigh = await diffusion(G, C=Chigh, initial_value=init_value)

    plt.figure()
    plt.plot(time_Clow, diff_Clow, label=f'C={Clow}')
    plt.plot(time_Cmid, diff_Cmid, label=f'C={Cmid}')
    plt.plot(time_Chigh, diff_Chigh, label=f'C={Chigh}')
    plt.xlabel('t')
    plt.ylabel('Normalized Difference in Direction of x(t) and equilibrium') # change
    plt.title('Convergence to Equilibrium Values')
    plt.legend()
    plt.show()
```

In [26]:
```
print('HIGHLIGHTED QUESTION --  make a comment about this in your report.')
print('The diffusion constant, C, impacts the rate of the spread. The lower the value
```

```
HIGHLIGHTED QUESTION --  make a comment about this in your report.
The diffusion constant, C, impacts the rate of the spread. The lower the value of the
constant the slower the spread starts. The bigger the value of the constant, the fast
er we diffuse through the network.
```

In [27]:
```
print('Dolphin Network')
print('HIGHLIGHTED QUESTION -- make a plot of the distance between the state x and the
await run_diffusion(DG, 0.1, 1, 3, init_value=1)
print('As we can see here, as we increase the diffusion constant we converge to the eq
```

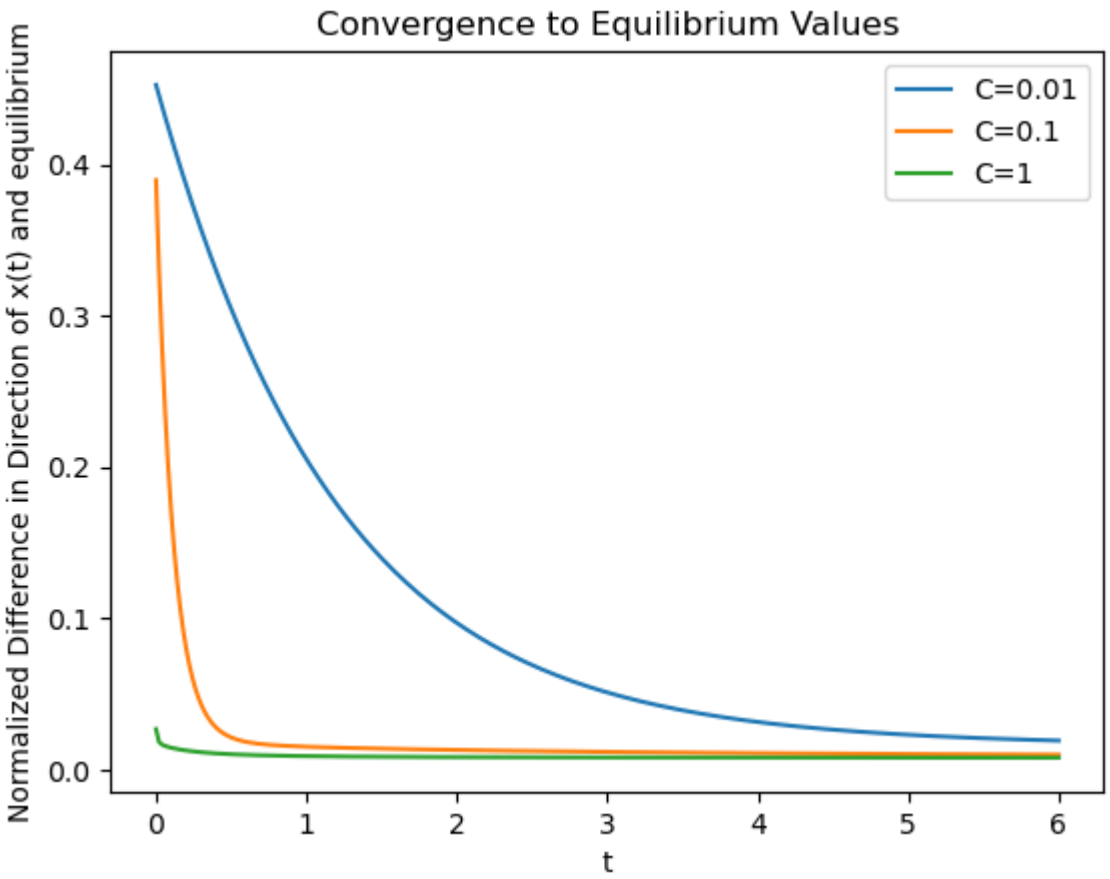
```
Dolphin Network
HIGHLIGHTED QUESTION -- make a plot of the distance between the state x and the equil
ibrium you've calculated over time
```



As we can see here, as we increase the diffusion constant we converge to the equilbir
ium faster.

In [29]:
```
print('Train Network')
print('HIGHLIGHTED QUESTION -- make a plot of the distance between the state x and the
await run_diffusion(TG, 0.01, 0.1, 1, init_value=1)
print('For the Train Network we see that we had to choose smaller values of C so that
```

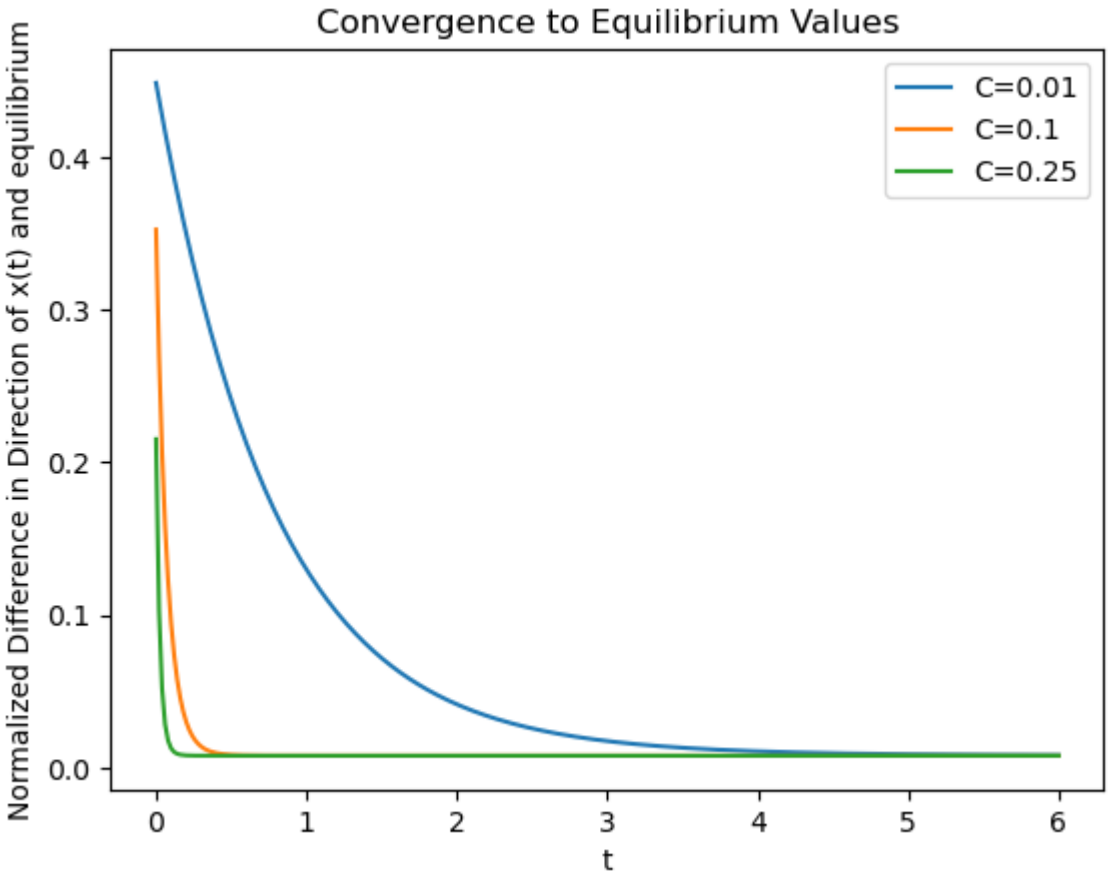
```
Train Network
HIGHLIGHTED QUESTION -- make a plot of the distance between the state x and the equil
ibrium you've calculated over time
```



For the Train Network we see that we had to choose smaller values of C so that we did not converge too fast. Choosing a value of C=1 makes us converge almost instantaneously.

In [38]:
```
print('Macaque Network')
print('HIGHLIGHTED QUESTION -- make a plot of the distance between the state x and the
await run_diffusion(MG, 0.01, 0.1, 0.25, init_value=1)
print('In the Macaque Network we see that we converge relatively fast with smaller C v
```

```
Macaque Network
HIGHLIGHTED QUESTION -- make a plot of the distance between the state x and the equil
ibrium you've calculated over time
```



In the Macaque Network we see that we converge relatively fast with smaller C values. This can tell us that the macaque network is more strongly connected compared to the dolphin and train network because we are still diffusing pretty through the network.

SI Model

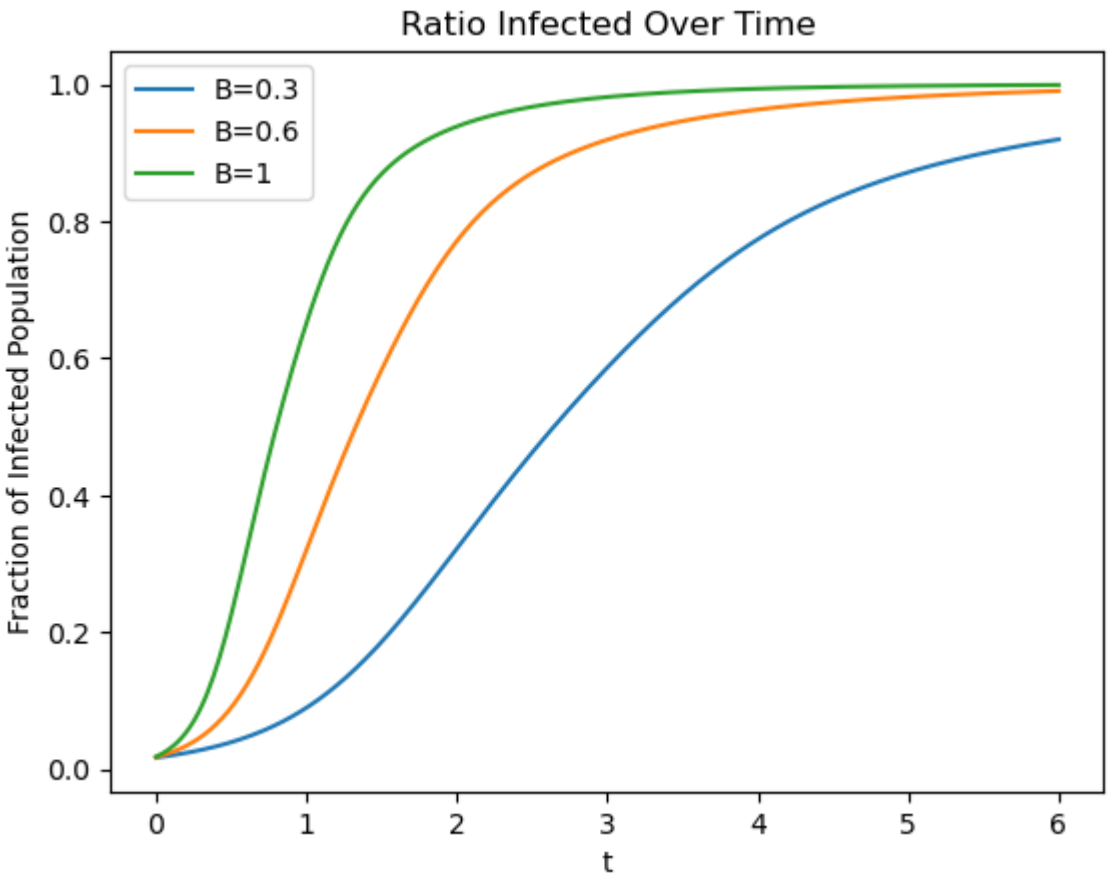
Simulating SI model...

In [39]:
```
async def SI(G, dt=0.02, T=6, beta=1, initial_value=1, visualize=False, wait_to_visual

    if visualize:
        reset_visualizer(d3, G)
        await asyncio.sleep(0.5)

    time = linspace(0,T,int(T/dt)) # the array of time points spaced by dt

    A = array(nx.adjacency_matrix(G).todense().T)
    N = G.number_of_nodes()

    x = zeros(N) # initialize N size vector of 0s

    x[0] = 1 # initialize first value
    S = 1 - x

    if visualize:
        color_by_value(d3,G,x)

    x_vals = []
    s_vals = []
    frac_infected = []
```

```
    frac_s = []

    for i,t in enumerate(time):
        # at each time step update the value of x!

        ds = -beta * (A @ x) * (S) # susceptible dynamics
        dx = beta * (A @ x) * (S)  # infected dynamics
        x += dx * dt
        S += ds * dt

        x_vals.append(copy(x))
        s_vals.append(copy(S))
        frac_infected.append(sum(x) / N)
        frac_s.append(sum(S) / N)

        if visualize:
            color_by_value(d3,G,x)  # update the visualizer

        if wait_to_visualize:
            await asyncio.sleep(0.1) # wait a little bit so the visualizer has time to

    return time, x_vals, s_vals, frac_infected, frac_s
```

In [43]:
```
time_Blow, _, _, frac_infected_DG_Blow, _ = await SI(DG, beta = 0.3)
time_Bmid, _, _, frac_infected_DG_Bmid, _ = await SI(DG, beta = 0.6)
time_Bhigh, _, _, frac_infected_DG_Bhigh, _ = await SI(DG, beta = 1)

print('Dolphin Network')
plt.figure()
plt.plot(time_Blow, frac_infected_DG_Blow, label='B=0.3')
plt.plot(time_Bmid, frac_infected_DG_Bmid, label='B=0.6')
plt.plot(time_Bhigh, frac_infected_DG_Bhigh, label='B=1')
plt.xlabel('t')
plt.ylabel('Fraction of Infected Population')
plt.title('Ratio Infected Over Time')
plt.legend()
plt.show()
print('We see that in the dolphin network the infected population increases much faste
```

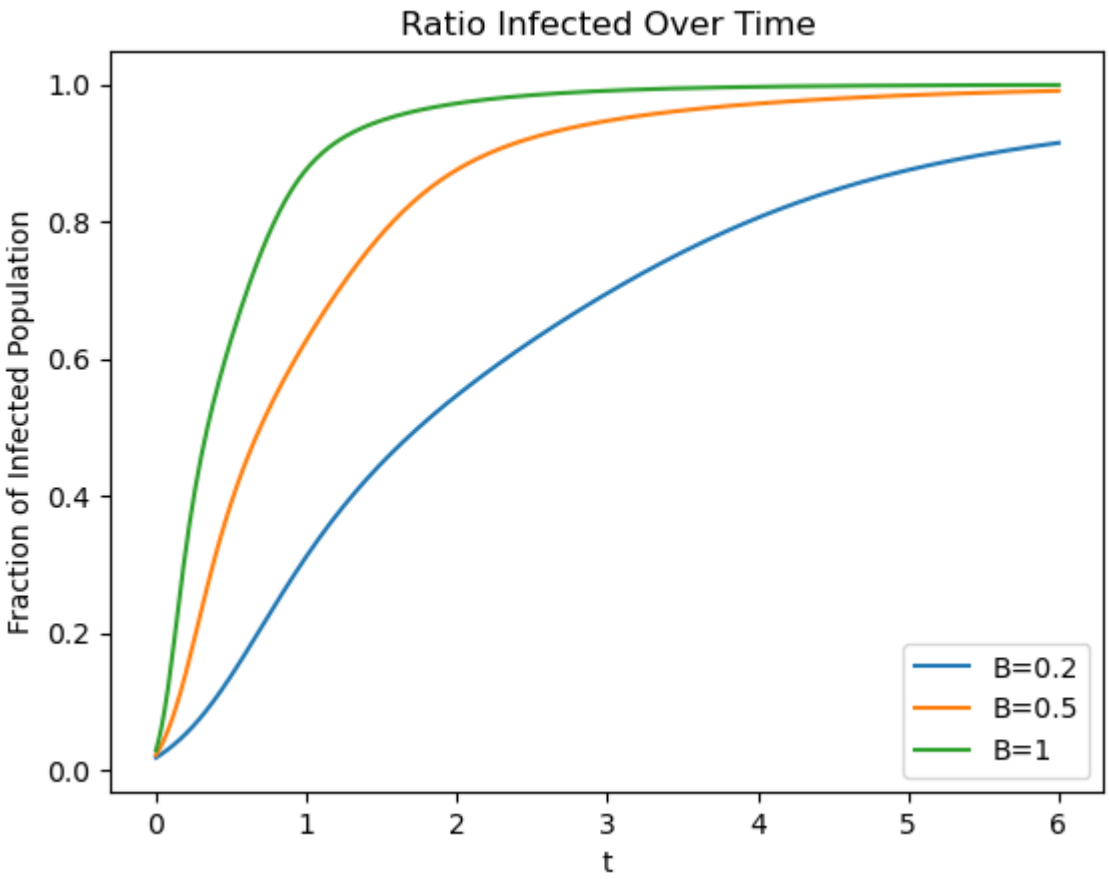
```
Dolphin Network
C:\Users\Humza\AppData\Local\Temp\ipykernel_16864\649843076.py:9: FutureWarning: adja
cency_matrix will return a scipy.sparse array instead of a matrix in Networkx 3.0.
  A = array(nx.adjacency_matrix(G).todense().T)
```



We see that in the dolphin network the infected population increases much faster with higher beta values.

In [45]:
```
time_Blow, _, _, frac_infected_TG_Blow, _ = await SI(TG, beta = 0.2)
time_Bmid, _, _, frac_infected_TG_Bmid, _ = await SI(TG, beta = 0.5)
time_Bhigh, _, _, frac_infected_TG_Bhigh, _ = await SI(TG, beta = 1) #, visualize=True

print('Train Network')
plt.figure()
plt.plot(time_Blow, frac_infected_TG_Blow, label='B=0.2')
plt.plot(time_Bmid, frac_infected_TG_Bmid, label='B=0.5')
plt.plot(time_Bhigh, frac_infected_TG_Bhigh, label='B=1')
plt.xlabel('t')
plt.ylabel('Fraction of Infected Population')
plt.title('Ratio Infected Over Time')
plt.legend()
plt.show()
print('We see that in the train network the network gets infected faster as we increas
```

```
Train Network
C:\Users\Humza\AppData\Local\Temp\ipykernel_16864\649843076.py:9: FutureWarning: adja
cency_matrix will return a scipy.sparse array instead of a matrix in Networkx 3.0.
  A = array(nx.adjacency_matrix(G).todense().T)
```

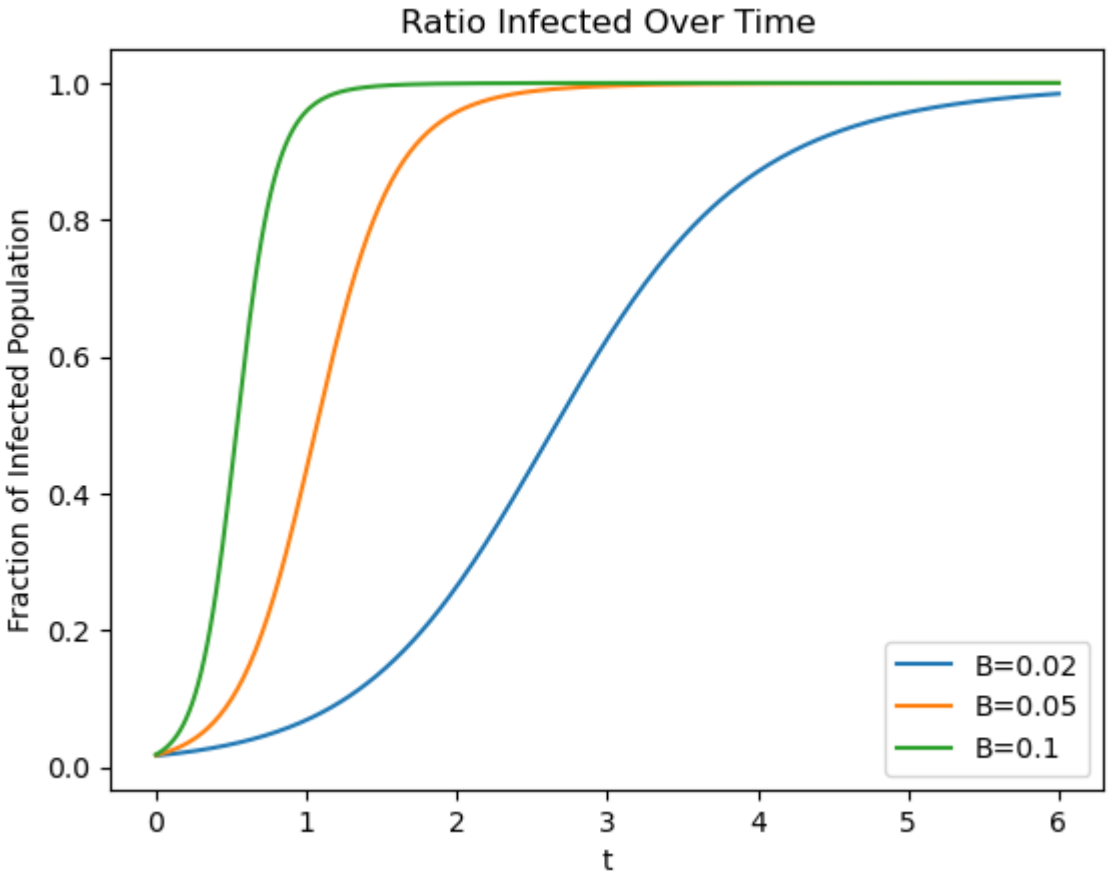


We see that in the train network the network gets infected faster as we increase our values of beta.

In [ ]:

In [49]:
```
time_Blow, _, _, frac_infected_MG_Blow, _ = await SI(MG, beta = 0.02)
time_Bmid, _, _, frac_infected_MG_Bmid, _ = await SI(MG, beta = 0.05)
time_Bhigh, xs, ss, frac_infected_MG_Bhigh, _ = await SI(MG, beta = 0.1)

print('Macaque Network')
plt.figure()
plt.plot(time_Blow, frac_infected_MG_Blow, label='B=0.02')
plt.plot(time_Bmid, frac_infected_MG_Bmid, label='B=0.05')
plt.plot(time_Bhigh, frac_infected_MG_Bhigh, label='B=0.1')
plt.xlabel('t')
plt.ylabel('Fraction of Infected Population')
plt.title('Ratio Infected Over Time')
plt.legend()
plt.show()
print('I chose particularly low values of beta for the Macaque Network because it is s
```

```
Macaque Network
C:\Users\Humza\AppData\Local\Temp\ipykernel_16864\649843076.py:9: FutureWarning: adja
cency_matrix will return a scipy.sparse array instead of a matrix in Networkx 3.0.
  A = array(nx.adjacency_matrix(G).todense().T)
```



I chose particularly low values of beta for the Macaque Network because it is so stro
ngly connected that if we choose values similar to the Train or Dolphin network, then
thhe ratio of infected people blows up to infinity.

In [51]:
```
print('HIGHLIGHTED QUESTION -- adjust it to see how it effects the evolution.')
print('The beta value controls the rate of spread of the disease. A beta value of 1 ir
```

```
HIGHLIGHTED QUESTION -- adjust it to see how it effects the evolution.
The beta value controls the rate of spread of the disease. A beta value of 1 indicate
s that the disease will always spread and a value of 0 indicates the disease will nev
er spread. So, the lower the value the less likely the disease is transmitted and vic
e versa. In our case, we see that with enough time the disease will always spread as
long as Beta > 0, however the rate at which it spreads is proportional to the value o
f Beta. The spread of the disease can also depend on how strongly connected the netwo
rk is. One quick note is that for the Macaque graph we had to use small Beta values d
ue to the dense nature of the graph causing overflow errors.
```

In [ ]:
```
```

SIR Model

Simulating SIR model...

In [52]:
```
async def SIR(G, dt=0.02, T=6, beta=1, gamma = 1, initial_value=1, visualize=False, wa

    if visualize:
        reset_visualizer(d3, G)
        await asyncio.sleep(2)

    time = linspace(0,T,int(T/dt)) # the array of time points spaced by dt
```

```
    A = array(nx.adjacency_matrix(G).todense().T)
    N = G.number_of_nodes()

    w, v = eig(A)
    max_eig_value = max(w)

    x = zeros(N) # initialize N size vector of 0s

    x[0] = 1 # initialize first value

    s = 1 - x # initialize susceptible values
    r = zeros(N) # initialize recovered values

    if visualize:
        color_by_value(d3,G,x)

    x_vals = []
    s_vals = []
    r_vals = []

    frac_x = []
    frac_s = []
    frac_r = []

    for i,t in enumerate(time):
        # at each time step update the value of x!

        ds = -beta * (A @ x) * (s) # susceptible dynamics
        dx = beta * (A @ x) * (s) - gamma * x # infected dynamics
        x += dx * dt
        s += ds * dt
        r += gamma * x * dt # recovered dynamics

        x_vals.append(copy(x))
        s_vals.append(copy(s))
        r_vals.append(copy(r))

        frac_x.append(sum(x) / N)
        frac_s.append(sum(s) / N)
        frac_r.append(sum(r) / N)

        if visualize:
            color_by_value(d3,G,x)  # update the visualizer

        if wait_to_visualize:
            await asyncio.sleep(0.1) # wait a little bit so the visualizer has time to

    return time, x_vals, s_vals, r_vals, frac_x, frac_s, frac_r, max_eig_value
```

In [65]:

```
def plot_SIR_three(plot1_vals, plot2_vals, plot3_vals):

    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 5))

    # first plot
    ax1.plot(plot1_vals[0], plot1_vals[1], label='infected')
    ax1.plot(plot1_vals[0], plot1_vals[2], label='susceptible')
    ax1.plot(plot1_vals[0], plot1_vals[3], label='recovered')
    ax1.set(xlabel='t', ylabel='Fraction of Population')
```

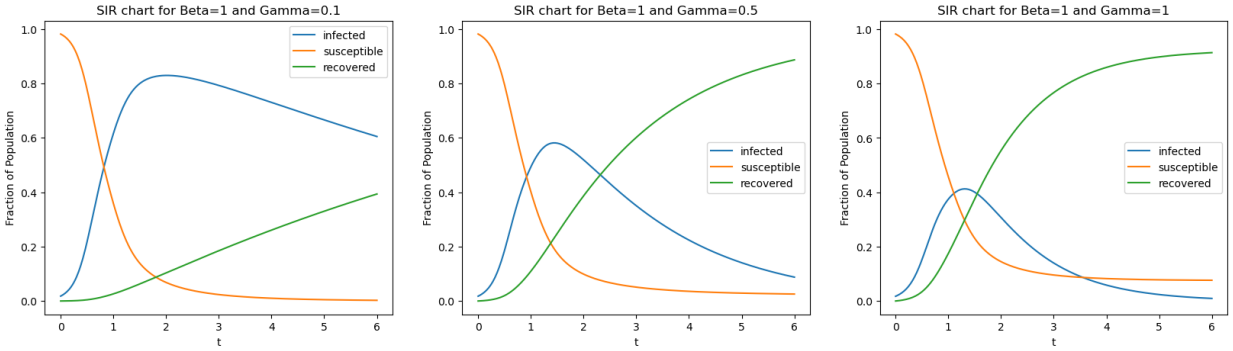
```
    ax1.set_title(f'SIR chart for Beta={plot1_vals[4]} and Gamma={plot1_vals[5]}')
    ax1.legend()

    # second plot
    ax2.plot(plot2_vals[0], plot2_vals[1], label='infected')
    ax2.plot(plot2_vals[0], plot2_vals[2], label='susceptible')
    ax2.plot(plot2_vals[0], plot2_vals[3], label='recovered')
    ax2.set(xlabel='t', ylabel='Fraction of Population')
    ax2.set_title(f'SIR chart for Beta={plot2_vals[4]} and Gamma={plot2_vals[5]}')
    ax2.legend()

    # third plot
    ax3.plot(plot3_vals[0], plot3_vals[1], label='infected')
    ax3.plot(plot3_vals[0], plot3_vals[2], label='susceptible')
    ax3.plot(plot3_vals[0], plot3_vals[3], label='recovered')
    ax3.set(xlabel='t', ylabel='Fraction of Population')
    ax3.set_title(f'SIR chart for Beta={plot3_vals[4]} and Gamma={plot3_vals[5]}')
    plt.legend()

    plt.show()
```

In [66]:
```
print('HIGHLIGHTED QUESTION -- Now make a plot of the expected fractions of susceptibl
```

```
HIGHLIGHTED QUESTION -- Now make a plot of the expected fractions of susceptible, inf
ected, and removed people over time using a SIR infection model
```

In [67]:
```
timeDGlow, _, _, _, frac_x_DGlow, frac_s_DGlow, frac_r_DGlow, max_eig_val_DGlow = awai
timeDGmid, _, _, _, frac_x_DGmid, frac_s_DGmid, frac_r_DGmid, max_eig_val_DGmid = awai
timeDGhigh, _, _, _, frac_x_DGhigh, frac_s_DGhigh, frac_r_DGhigh, max_eig_val_DGhigh =

print('Dolphin Network')
plot_SIR_three((timeDGlow, frac_x_DGlow, frac_s_DGlow, frac_r_DGlow, 1, 0.1),
               (timeDGmid, frac_x_DGmid, frac_s_DGmid, frac_r_DGmid, 1, 0.5),
               (timeDGhigh, frac_x_DGhigh, frac_s_DGhigh, frac_r_DGhigh, 1, 1))
print('I kept the beta values constant since we were inspecting the effect of gamma. W
```

```
C:\Users\Humza\AppData\Local\Temp\ipykernel_16864\3207312952.py:9: FutureWarning: adj
acency_matrix will return a scipy.sparse array instead of a matrix in Networkx 3.0.
  A = array(nx.adjacency_matrix(G).todense().T)
Dolphin Network
```

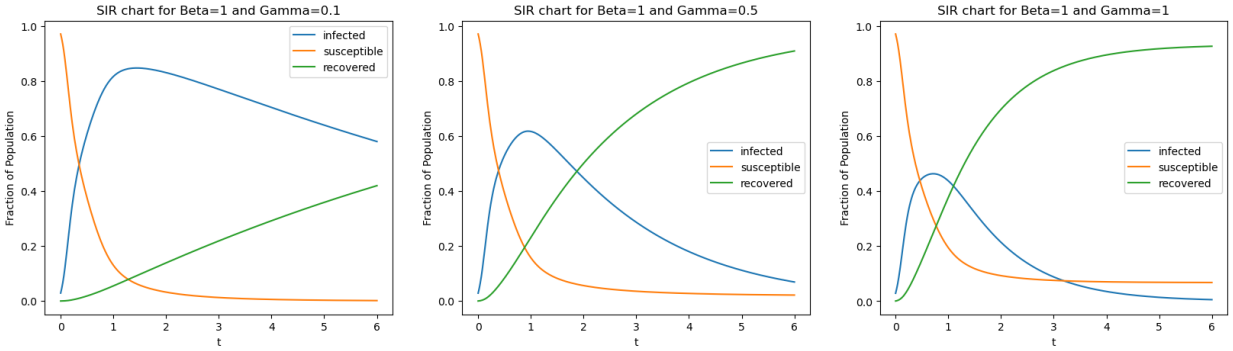



```
I kept the beta values constant since we were inspecting the effect of gamma. We see
that as gamma increases, the maximum fraction of the population that gets infected de
creases which tells us that people are reocvering much faster. One thing to note is t
hat for a low gamma value we still need time for our infected population to recover.
```

In [68]:
```
timeTGlow, _, _, _, frac_x_TGlow, frac_s_TGlow, frac_r_TGlow, max_eig_val_TGlow = awai
timeTGmid, _, _, _, frac_x_TGmid, frac_s_TGmid, frac_r_TGmid, max_eig_val_TGmid = awai
timeTGhigh, _, _, _, frac_x_TGhigh, frac_s_TGhigh, frac_r_TGhigh, max_eig_val_TGhigh =
```

```
print('Train Network')
plot_SIR_three((timeTGlow, frac_x_TGlow, frac_s_TGlow, frac_r_TGlow, 1, 0.1),
               (timeTGmid, frac_x_TGmid, frac_s_TGmid, frac_r_TGmid, 1, 0.5),
               (timeTGhigh, frac_x_TGhigh, frac_s_TGhigh, frac_r_TGhigh, 1, 1))
print('I kept the beta values constant since we were inspecting the effect of gamma. W
```

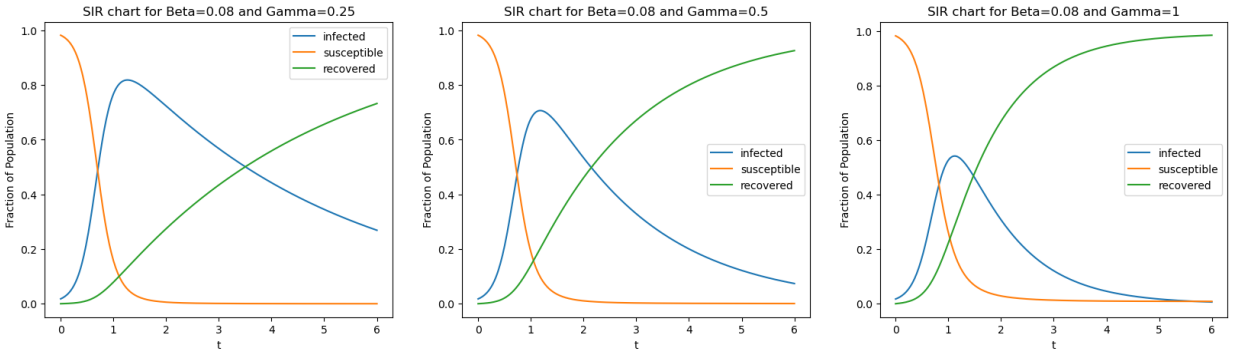
```
Train Network
C:\Users\Humza\AppData\Local\Temp\ipykernel_16864\3207312952.py:9: FutureWarning: adj
acency_matrix will return a scipy.sparse array instead of a matrix in Networkx 3.0.
  A = array(nx.adjacency_matrix(G).todense().T)
```




```
I kept the beta values constant since we were inspecting the effect of gamma. We see
that as gamma increases, the maximum fraction of the population that gets infected de
creases which tells us that people are reocvering much faster. One thing to note is t
hat the maximum fraction of population on this network is typically reached around t=
1 which tells us that this network is more strongly connected than the Dolphin networ
k. Another important note is that we may not always have our fraction of recovered po
pulation reach 1 since there may no longer be any more infected people.
```

In [69]:
```
timeMGlow, _, _, _, frac_x_MGlow, frac_s_MGlow, frac_r_MGlow, max_eig_val_MGlow = awai
timeMGmid, _, _, _, frac_x_MGmid, frac_s_MGmid, frac_r_MGmid, max_eig_val_MGmid = awai
timeMGhigh, _, _, _, frac_x_MGhigh, frac_s_MGhigh, frac_r_MGhigh, max_eig_val_MGhigh =

print('Macaque Network')
plot_SIR_three((timeMGlow, frac_x_MGlow, frac_s_MGlow, frac_r_MGlow, 0.08, 0.25),
               (timeMGmid, frac_x_MGmid, frac_s_MGmid, frac_r_MGmid, 0.08, 0.5),
               (timeMGhigh, frac_x_MGhigh, frac_s_MGhigh, frac_r_MGhigh, 0.08, 1))
print('I kept the beta values constant and really low based on previos observations si
```

```
Macaque Network
C:\Users\Humza\AppData\Local\Temp\ipykernel_16864\3207312952.py:9: FutureWarning: adj
acency_matrix will return a scipy.sparse array instead of a matrix in Networkx 3.0.
  A = array(nx.adjacency_matrix(G).todense().T)
```




```
I kept the beta values constant and really low based on previos observations since we
were inspecting the effect of gamma. In essense, we notice the same trends as we saw
before for the Dolphin and Train networks.
```

In [70]:
```
print('HIGHLIGHTED QUESTION -- Describe this briefly and explain why we see this effec
print('The maximum eigenvalue gives insight into the epidemic threshold. A smaller max
```

```
HIGHLIGHTED QUESTION -- Describe this briefly and explain why we see this effect.
The maximum eigenvalue gives insight into the epidemic threshold. A smaller max eigen
value tells us that it is harder for the disease to spread. A larger max eigenvalue t
ells us that it is easier for the disease to spread. In a network where it is easier
for the disease to spread we can choose a higher gamma value to contain the spread of
the disease or if we want to see how the disease spreads with a low gamma value (reco
very rate), we can also do that.
```

In [ ]:
```
```

In [71]:
```
reset_visualizer(d3, DG)
```

Independent Cascade

The following function implements an influence cascade model on the graph `G` and initial active node set `x` with the same probability `p` to activate a neighbor node along each edge.

In [72]:
```
# G: Graph
# p: uniform probability to activate across an edge
# x: initial active seed set (as a list/array)
def influence_cascade(G,p,x):
    G = deepcopy(G)
    x = deepcopy(x)
    activated_nodes = set([])
    for i,xi in enumerate(x):
        if xi > 0:
            activated_nodes.add(G.node_by_index(i))

    while len(activated_nodes) > 0:
        newly_activated = set([])
        for u in activated_nodes:
            x[G.node_index(u)] = 1
            nbrs = G.neighbors(u)
            to_rm = set([])
            for v in nbrs:
                if random.random() <= p:
                    newly_activated.add(v)
                to_rm.add((u,v))
            G.remove_edges_from(to_rm)
        activated_nodes = newly_activated
        #print sum(x)
    return x
```

Repeating the influence cascade many times...

In [73]:
```
def independent_cascade(G, p=0.5, perc_p=0.5, b=1, g=1):
    # IC_DG = D3Graph( nx.read_weighted_edgelist('dolphins.edgelist',create_using=nx.G

    online_x = zeros(G.number_of_nodes()) # initialize N size vector of 0s
    online_x[0] = 1 # initialize first value

    online_cum_x = zeros(G.number_of_nodes())
```

```
online_avg_x = zeros(G.number_of_nodes())

max_iter = 1000

for i in range(max_iter):
    online_tmp_x = influence_cascade(G, p, online_x)
    for j in range(len(online_tmp_x)):
        online_cum_x[j] += online_tmp_x[j]

for i in range(len(online_cum_x)):
    online_avg_x[i] = online_cum_x[i] / max_iter


#################################################################

perc_cum_x = zeros(G.number_of_nodes())
perc_avg_x = zeros(G.number_of_nodes())

for i in range(max_iter):

    perc_temp_G = deepcopy(G)
    perc_to_rm = set([])

    for e in perc_temp_G.edges(): # compute edges to percolate
        if random.random() <= perc_p:
            perc_to_rm.add(e)

    perc_temp_G.remove_edges_from(perc_to_rm) # percolate edges

    perc_x = zeros(perc_temp_G.number_of_nodes()) # initialize N size vector of 0s
    perc_x[0] = 1 # initialize first value

    perc_tmp_x = influence_cascade(perc_temp_G, 1, perc_x)

    for j in range(len(perc_tmp_x)):
        perc_cum_x[j] += perc_tmp_x[j]


for i in range(len(perc_cum_x)):
    perc_avg_x[i] = perc_cum_x[i] / max_iter


#################################################################

lti_x = zeros(G.number_of_nodes()) # initialize N size vector of 0s
lti_x[0] = 1 # initialize first value

lti_cum_x = zeros(G.number_of_nodes())
lti_avg_x = zeros(G.number_of_nodes())

beta = b
gamma = g
expected_tao = 1 / gamma
lti_p = 1 - exp(-beta*expected_tao)

for i in range(max_iter):
    lti_tmp_x = influence_cascade(G, lti_p, lti_x)
    for j in range(len(lti_tmp_x)):
        lti_cum_x[j] += lti_tmp_x[j]
```

```
    for i in range(len(lti_cum_x)):
        lti_avg_x[i] = lti_cum_x[i] / max_iter


    online_numbered_nodes = [i+1 for i in range(G.number_of_nodes())]
    perc_numbered_nodes = [i+1 for i in range(G.number_of_nodes())]
    lti_numbered_nodes = [i+1 for i in range(G.number_of_nodes())]


    print('HIGHLIGHTED QUESTION -- On your plot of average activation, now plot the fi
    print(f'Influence Cascade Probability={p}')
    print(f'Percolation probability={perc_p}')
    plt.figure() #figsize=(15,10))
    plt.plot(online_numbered_nodes, online_avg_x, color='blue', label='Online', linewi
    plt.plot(perc_numbered_nodes, perc_avg_x, color='green', label='Percolation', line
    plt.plot(lti_numbered_nodes, lti_avg_x, color='red', label=f'Late Time Infection c
    plt.xlabel('Node i')
    plt.ylabel('Probability of Activation')
    plt.legend()
    plt.show()
```

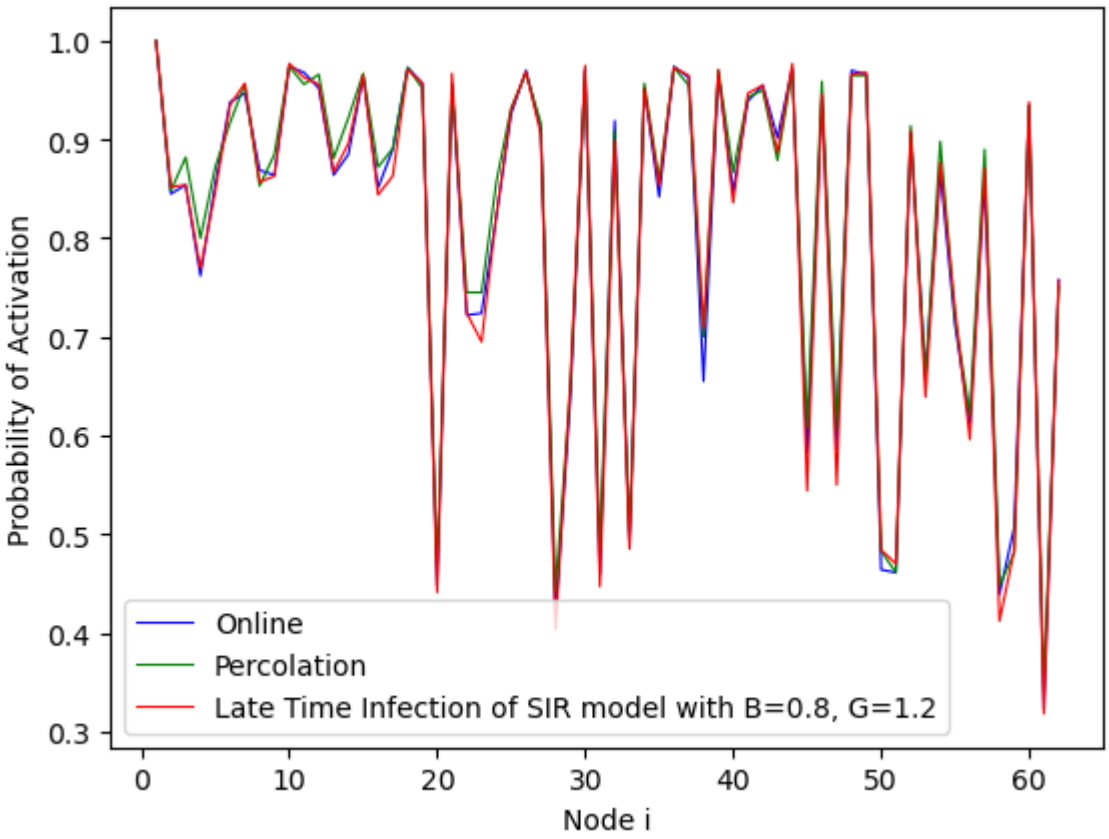
Using a percolation approach...

Using a late time infection of SIR model approach...

In [74]:
```
IC_DG = D3Graph( nx.read_weighted_edgelist('dolphins.edgelist',create_using=nx.Graph)
IC_TG = D3Graph( nx.read_weighted_edgelist('train.edgelist',create_using=nx.Graph) )
IC_MG = D3Graph( nx.read_weighted_edgelist('macaque.edgelist',create_using=nx.Graph) )
```

In [82]:
```
print('Dolphin Network')
independent_cascade(IC_DG, p=0.5, perc_p=0.5, b=0.8, g=1.2)
print('We see that for influence cascade done online, percolation, and late time infec
```

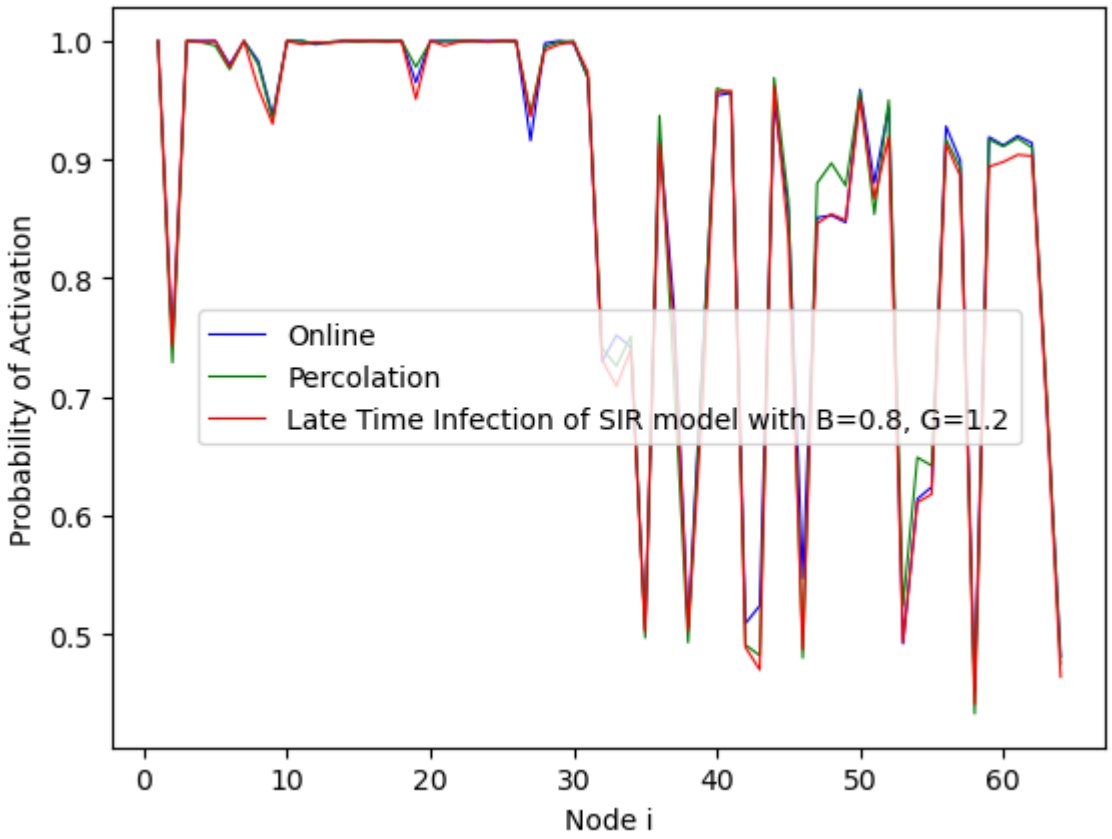
```
Dolphin Network
HIGHLIGHTED QUESTION -- On your plot of average activation, now plot the final recove
red probability for each node from the SIR model.
Influence Cascade Probability=0.5
Percolation probability=0.5
```



We see that for influence cascade done online, percolation, and late time infection p
roperties of the SIR model that for appropriate p, beta, and gamma values the probabi
lity of a node being infected/sick remains roughly similar - likely varying from rand
om seeds. An interesting thing to note is how the probability of activation varies th
roughout all nodes indicating that certain nodes are hubs and authorities but are spr
ead further throughout the network rather than being close together.

In [81]:
```
print('Train Network')
independent_cascade(IC_TG, p=0.5, perc_p=0.5, b=0.8, g=1.2)
print('We see that for influence cascade done online, percolation, and late time infec
```

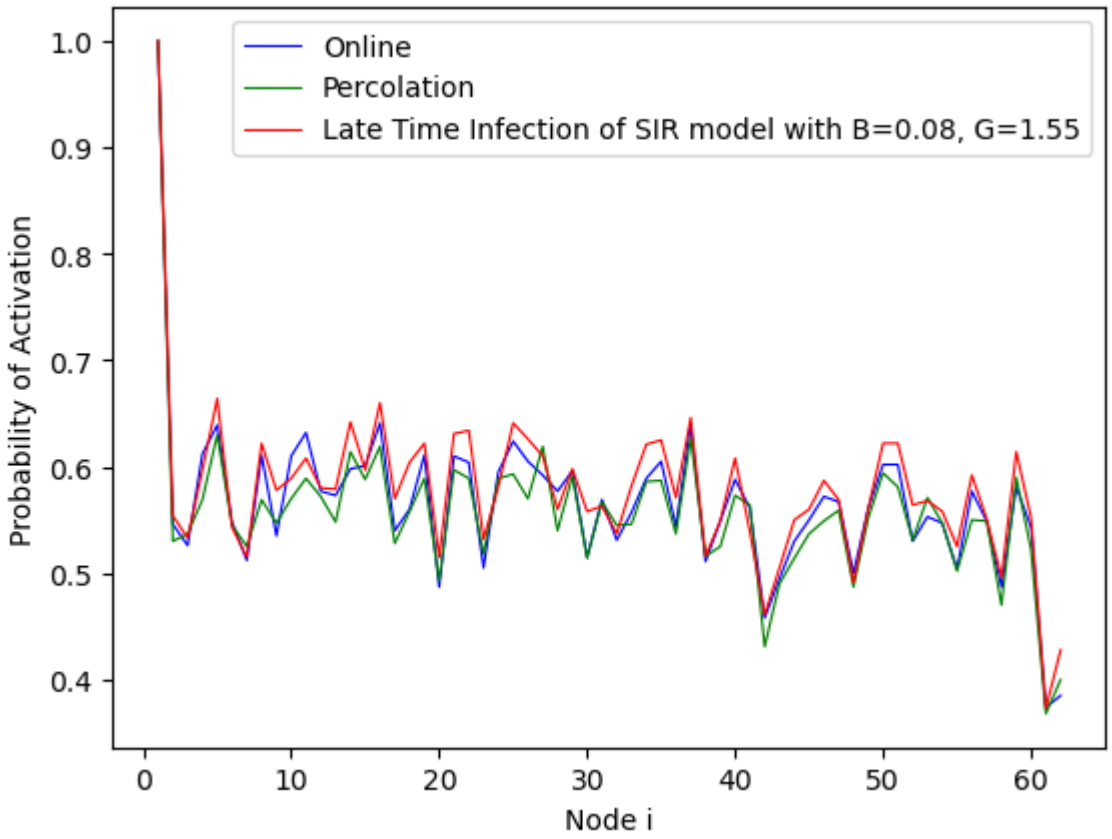
```
Train Network
HIGHLIGHTED QUESTION -- On your plot of average activation, now plot the final recove
red probability for each node from the SIR model.
Influence Cascade Probability=0.5
Percolation probability=0.5
```



We see that for influence cascade done online, percolation, and late time infection p
roperties of the SIR model that for appropriate p, beta, and gamma values the probabi
lity of a node being infected/sick remains roughly similar - likely varying from rand
om seeds. An interesting note is that for the first 30 nodes we mostly have higher pr
obabilities of activation whereas afterwards this probability of activation varies. T
his could be explained by our initial setup configuration and also how the train netw
ork may be arranged to have hubs and authorities within those first 30 nodes and then
scattered throughout.

In [86]:
```
print('Macaque Network')
independent_cascade(IC_MG, p=0.05, perc_p=0.95, b=0.08, g=1.55)
print('I chose a lower influence cascade probability and higher percolation probabilit
```

```
Macaque Network
HIGHLIGHTED QUESTION -- On your plot of average activation, now plot the final recove
red probability for each node from the SIR model.
Influence Cascade Probability=0.05
Percolation probability=0.95
```



I chose a lower influence cascade probability and higher percolation probability since we have previously found that the macaque network is very strongly connected, therefore it also requires a lower beta value and higher gamma value. This shows us that the activation probability for each node is roughly the same for each of the methods. An interesting thing to note is that the macaque network has mostly lower probabilities of activation for all nodes and they vary roughly the same for all nodes throughout - however we need to take into account we have already chosen low probability value and are percolating a lot of the edges within the network.

In [ ]: