Jawad Chowdhury : 801135477

In [3]:
```
#### import
import networkx as nx
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
```

1. Signed Networks (Slashdot)

1a. compute the number of triangles in the network

In [4]:
```
file_name = "datasets/soc-sign-Slashdot081106.txt"
uG_s = nx.read_edgelist(file_name, nodetype=int, data=(("sign", int),))
```

In [ ]:

In [5]:
```
dict_triangles = nx.triangles(uG_s)
```

In [6]:
```
print("The number of triangles in the network : ", sum(nx.triangles(uG_s).values()) / 3)
```

```
The number of triangles in the network :  548054.0
```

1b. Report the fraction of balanced triangles and unbalanced triangles. (assume network is undirected;

if there is a sign for each direction, randomly pick one.)

In [7]:
```
triangles = [c for c in nx.cycle_basis(uG_s) if len(c)==3]
triangle_types={}
for triangle in triangles:
    tri=nx.subgraph(uG_s,triangle)
    triangle_types[tuple(tri.nodes())]=np.product([x[2]['sign'] for x in tri.edges(data=True)])
```

In [8]:
```
n_bal = 0
n_imbal = 0
for k, v in triangle_types.items():
    if v==1:
        n_bal +=1
    else:
        n_imbal +=1
f_bal = n_bal/(n_bal+n_imbal)
f_imbal = n_imbal/(n_bal+n_imbal)
```

In [9]:
```
print("Fraction of balanced triangles : ", f_bal)
print("Fraction of unbalanced triangles : ", f_imbal)
```

```
Fraction of balanced triangles :  0.838768747820021
Fraction of unbalanced triangles :  0.16123125217997908
```

1c. Compare the frequency of signed triads in real and “shuffled” networks (refer slides) (assume

network is undirected; if there is a sign for each direction, randomly pick one.)

In [10]:
```
triangles = [c for c in nx.cycle_basis(uG_s) if len(c)==3]
```

In [11]:
```
triangle_types={}
for triangle in triangles:
    tri=nx.subgraph(uG_s,triangle)
    triangle_types[tuple(tri.nodes())]=np.sum([x[2]['sign'] for x in tri.edges(data=True)])
```

In [12]:
```
triangle_type_t3 = {}
triangle_type_t2 = {}
triangle_type_t1 = {}
triangle_type_t0 = {}
for k,v in triangle_types.items():
    if v==3:
        triangle_type_t3[k] = v
    elif v==1:
        triangle_type_t2[k] = v
    elif v==-1:
        triangle_type_t1[k] = v
    elif v==-3:
        triangle_type_t0[k] = v
```

In [13]:
```
n_real_t3=len(triangle_type_t3.items())
n_real_t2=len(triangle_type_t2.items())
n_real_t1=len(triangle_type_t1.items())
n_real_t0=len(triangle_type_t0.items())
```

In [14]:
```
n_real = n_real_t3 + n_real_t2 + n_real_t1 + n_real_t0
print(n_real_t3/n_real, n_real_t2/n_real, n_real_t1/n_real, n_real_t0/n_real)
```

```
0.6367718869898848 0.12722357865364492 0.20199686083013604 0.034007673526334145
```

In [15]:
```
cpos=0
cneg=0
for e in uG_s.edges(data=True):
    if e[2]['sign'] == 1:
        cpos +=1
    else:
        cneg +=1
cpos_cneg = [1]*cpos + [-1]*cneg
import random
random.shuffle(cpos_cneg)
```

In [16]:
```
uG_s_shuffled = uG_s.copy()
```

In [17]:
```
i=0
for e in uG_s_shuffled.edges(data=True):
    e[2]['sign']=cpos_cneg[i]
    i+=1
```

In [18]:
```
shuffled_triangles = [c for c in nx.cycle_basis(uG_s_shuffled) if len(c)==3]
```

In [19]:
```
shuffled_triangle_types={}
for triangle in shuffled_triangles:
    tri=nx.subgraph(uG_s_shuffled,triangle)
    shuffled_triangle_types[tuple(tri.nodes())]=np.sum([x[2]['sign'] for x in tri.edges(data=True)])
```

In [20]:
```
shuffled_triangle_type_t3 = {}
shuffled_triangle_type_t2 = {}
shuffled_triangle_type_t1 = {}
shuffled_triangle_type_t0 = {}
for k,v in shuffled_triangle_types.items():
    if v==3:
        shuffled_triangle_type_t3[k] = v
    elif v==1:
        shuffled_triangle_type_t2[k] = v
    elif v==-1:
        shuffled_triangle_type_t1[k] = v
    elif v==-3:
        shuffled_triangle_type_t0[k] = v
```

In [21]:
```
n_shuffled_t3=len(shuffled_triangle_type_t3.items())
n_shuffled_t2=len(shuffled_triangle_type_t2.items())
n_shuffled_t1=len(shuffled_triangle_type_t1.items())
n_shuffled_t0=len(shuffled_triangle_type_t0.items())
```

In [22]:
```
n_shuffled = n_shuffled_t3 + n_shuffled_t2 + n_shuffled_t1 + n_shuffled_t0
print(n_shuffled_t3/n_shuffled, n_shuffled_t2/n_shuffled, n_shuffled_t1/n_shuffled, n_shuffled_t0/n_shu
ffled)
```

```
0.4262690298674048 0.4238135631054958 0.1355864101075941 0.014330996919505335
```

Triad	Real, P(T)	Shuffled, $P(T_0)$	Consistent with Balance?
T_3 (Balanced)	0.64	0.43	Yes
T_1 (Balanced)	0.20	0.14	Yes
T_2 (Unbalanced)	0.13	0.42	Yes
T_0 (Unbalanced)	0.03	0.01	No

1d. Compute “Gen. Surprise” (assume directed signed networks) for each of the 16 types

In [23]:
```
# dG_s2 = nx.read_edgelist(file_name, nodetype=int, data=(("sign", int),), create_using=nx.DiGraph)
# dG_s=nx.subgraph(dG_s2,list(range(300)))

dG_s = nx.read_edgelist(file_name, nodetype=int, data=(("sign", int),), create_using=nx.DiGraph)
```

In [24]:
```
uG_s = dG_s.to_undirected()
triangles = [c for c in nx.cycle_basis(uG_s) if len(c)==3]
```

In [25]:
```
t={}
for i in range(1,17,1):
    t[i]=[]
```

In [26]:
```
def get_t(t,a,b,x,dG_s):
    sg = nx.subgraph(dG_s,[a,b,x])
    sg_edges = sg.edges()
    sg_edges_data = sg.edges(data=True)

    if (a,b) in sg_edges and (a,x,{'sign':1}) in sg_edges_data and (x,b,{'sign':1}) in sg_edges_data:
        if (a,b,x) not in t[1]:
            t[1].append((a,b,x))
    elif (a,b) in sg_edges and (a,x,{'sign':1}) in sg_edges_data and (x,b,{'sign':-1}) in sg_edges_data
:
        if (a,b,x) not in t[2]:
            t[2].append((a,b,x))
    elif (a,b) in sg_edges and (a,x,{'sign':-1}) in sg_edges_data and (x,b,{'sign':1}) in sg_edges_data
:
        if (a,b,x) not in t[5]:
            t[5].append((a,b,x))
    elif (a,b) in sg_edges and (a,x,{'sign':-1}) in sg_edges_data and (x,b,{'sign':-1}) in sg_edges_dat
a:
        if (a,b,x) not in t[6]:
            t[6].append((a,b,x))

    elif (a,b) in sg_edges and (a,x,{'sign':1}) in sg_edges_data and (b,x,{'sign':1}) in sg_edges_data:
        if (a,b,x) not in t[3]:
            t[3].append((a,b,x))
    elif (a,b) in sg_edges and (a,x,{'sign':1}) in sg_edges_data and (b,x,{'sign':-1}) in sg_edges_data
:
        if (a,b,x) not in t[4]:
            t[4].append((a,b,x))
    elif (a,b) in sg_edges and (a,x,{'sign':-1}) in sg_edges_data and (b,x,{'sign':1}) in sg_edges_data
:
        if (a,b,x) not in t[7]:
            t[7].append((a,b,x))
    elif (a,b) in sg_edges and (a,x,{'sign':-1}) in sg_edges_data and (b,x,{'sign':-1}) in sg_edges_dat
a:
        if (a,b,x) not in t[8]:
            t[8].append((a,b,x))

    elif (a,b) in sg_edges and (x,a,{'sign':1}) in sg_edges_data and (x,b,{'sign':1}) in sg_edges_data:
        if (a,b,x) not in t[9]:
            t[9].append((a,b,x))
    elif (a,b) in sg_edges and (x,a,{'sign':1}) in sg_edges_data and (x,b,{'sign':-1}) in sg_edges_data
:
        if (a,b,x) not in t[10]:
            t[10].append((a,b,x))
    elif (a,b) in sg_edges and (x,a,{'sign':-1}) in sg_edges_data and (x,b,{'sign':1}) in sg_edges_data
:
        if (a,b,x) not in t[13]:
            t[13].append((a,b,x))
    elif (a,b) in sg_edges and (x,a,{'sign':-1}) in sg_edges_data and (x,b,{'sign':-1}) in sg_edges_dat
a:
        if (a,b,x) not in t[14]:
            t[14].append((a,b,x))

    elif (a,b) in sg_edges and (x,a,{'sign':1}) in sg_edges_data and (b,x,{'sign':1}) in sg_edges_data:
        if (a,b,x) not in t[11]:
            t[11].append((a,b,x))
    elif (a,b) in sg_edges and (x,a,{'sign':1}) in sg_edges_data and (b,x,{'sign':-1}) in sg_edges_data
:
        if (a,b,x) not in t[12]:
            t[12].append((a,b,x))
    elif (a,b) in sg_edges and (x,a,{'sign':-1}) in sg_edges_data and (b,x,{'sign':1}) in sg_edges_data
:
        if (a,b,x) not in t[15]:
            t[15].append((a,b,x))
    elif (a,b) in sg_edges and (x,a,{'sign':-1}) in sg_edges_data and (b,x,{'sign':-1}) in sg_edges_dat
a:
        if (a,b,x) not in t[16]:
            t[16].append((a,b,x))
    return t
```

In [27]:
```
for utriad in triangles:
    (a,b,x) = utriad
    t=get_t(t,a,b,x,dG_s)
    (a,x,b) = utriad
    t=get_t(t,a,b,x,dG_s)
    (b,a,x) = utriad
    t=get_t(t,a,b,x,dG_s)
    (b,x,a) = utriad
    t=get_t(t,a,b,x,dG_s)
    (x,a,b) = utriad
    t=get_t(t,a,b,x,dG_s)
    (x,b,a) = utriad
    t=get_t(t,a,b,x,dG_s)
```

In [28]:
```
for k,v in t.items():
    print(k,len(v))
```

```
1 34308
2 2294
3 14375
4 1449
5 3749
6 1645
7 2491
8 1773
9 11811
10 1259
11 544
12 142
13 1213
14 2915
15 132
16 86
```

In [29]:
```
import math
def get_gen_surprise(v=[]):
    k=0
    list_pgai = []
    for t in v:
        if dG_s[t[0]][t[1]]['sign']==1:
            k+=1

        ai=t[0]
        ai_gen_edges = dG_s.out_edges(ai)
        n_ai_gen_edges = len(ai_gen_edges)
        if n_ai_gen_edges==0:
            list_pgai.append(0)
            continue
        cp=0
        for e in ai_gen_edges:
            if dG_s[e[0]][e[1]]['sign']==1:
                cp+=1
        prob_gen_ai = (cp/n_ai_gen_edges)
        list_pgai.append(prob_gen_ai)

    sum_of_pgai = 0
    sum_of_pgai_mul_comp_pgai = 0
    for e in list_pgai:
        sum_of_pgai +=e
        sum_of_pgai_mul_comp_pgai += e*(1-e)
    result = (k-sum_of_pgai)/math.sqrt(sum_of_pgai_mul_comp_pgai)
    return result
```

In [30]:
```
print("Gen. Surprise (for each of 16 types)")
for k,v in t.items():
    print('T=',k,' : ',get_gen_surprise(v))
```

```
Gen. Surprise (for each of 16 types)
T= 1  :  38.20467542285642
T= 2  :  -34.3655383422227
T= 3  :  24.758926384494785
T= 4  :  -18.55978615212179
T= 5  :  -9.382142936100609
T= 6  :  6.033886456950722
T= 7  :  -11.643346429787199
T= 8  :  17.288710145967435
T= 9  :  28.507429633591315
T= 10  :  -18.963198203816113
T= 11  :  4.7700135762088385
T= 12  :  -3.76480214332523
T= 13  :  -4.842128283177797
T= 14  :  1.0251980073754139
T= 15  :  -1.371184672192175
T= 16  :  0.4980906392657173
```

In [ ]:

1e. Rewrite the formula for “Rec. Surprise” using the idea introduced in “Gen. Surprise”

Formula for 'Rec. Surprise':

$$S_r(X) = \frac{k - \sum_{i=1}^{n} P_r(A_i)}{\sqrt{P_r(A_i) * (1 - P_r(A_i))}}$$

In [ ]:

1f. Compute “Rec. Surprise” for all each of the 16 types.

In [31]:
```
import math
def get_rec_surprise(v=[]):
    k=0
    list_prai = []
    for t in v:
        if dG_s[t[0]][t[1]]['sign']==1:
            k+=1

        ai=t[0]
        ai_rec_edges = dG_s.in_edges(ai)
        n_ai_rec_edges = len(ai_rec_edges)
        if n_ai_rec_edges==0:
            list_prai.append(0)
            continue
        cp=0
        for e in ai_rec_edges:
            if dG_s[e[0]][e[1]]['sign']==1:
                cp+=1
        prob_rec_ai = (cp/n_ai_rec_edges)
        list_prai.append(prob_rec_ai)

    sum_of_prai = 0
    sum_of_prai_mul_comp_prai = 0
    for e in list_prai:
        sum_of_prai +=e
        sum_of_prai_mul_comp_prai += e*(1-e)
    result = (k-sum_of_prai)/math.sqrt(sum_of_prai_mul_comp_prai)
    return result
```

In [32]:
```
print("Rec. Surprise (for each of 16 types)")
for k,v in t.items():
    print('T=',k,' : ',get_rec_surprise(v))
```

```
Rec. Surprise (for each of 16 types)
T= 1  :  81.24812745525661
T= 2  :  -35.34180242799213
T= 3  :  53.07907920727959
T= 4  :  -12.033104064438797
T= 5  :  -48.32696247477239
T= 6  :  -6.4338609629776276
T= 7  :  -43.775847935209235
T= 8  :  0.8675958669642168
T= 9  :  31.452897821542905
T= 10  :  -32.85991135169408
T= 11  :  4.757702452562508
T= 12  :  -4.544570803451088
T= 13  :  10.80308754750378
T= 14  :  29.737014750336794
T= 15  :  3.468028789194091
T= 16  :  6.542325796837684
```

In [ ]:

2. The SIR Model of Disease Spreading

In [57]:
```
### algorithm 1

from collections import Counter
import random
def subtract_list(x,y):
    return list((Counter(x) - Counter(y)).elements())

def algorithm_1(G):
    beta, delta = 0.05, 0.5
    V = list(G.nodes())
    I = [random.choice(V)]
    S = subtract_list(V,I)
    R = []

    while len(I) != 0:
        S_prime = []
        I_prime = []
        J_prime = []
        R_prime = []

        for u in V:
            if u in S:
                list_edge_uv = G.edges(u)
                for uv in list_edge_uv:
                    v = uv[0] if uv[0] != u else uv[1]
                    if v in I:
                        r = random.uniform(0, 1)
                        if r <= beta:
                            if u not in S_prime:
                                S_prime.append(u)
                            if u not in I_prime:
                                I_prime.append(u)
                            break
            elif u in I:
                r = random.uniform(0,1)
                if r <= delta:
                    if u not in J_prime:
                        J_prime.append(u)
                    if u not in R_prime:
                        R_prime.append(u)


        S = list(set(subtract_list(S, S_prime)))
        I = list(set(subtract_list((I + I_prime), J_prime)))
        R = list(set(R + R_prime))

    return len(S),len(I),len(R)
```

In [ ]:

In [58]:
```
# reading datasets
uG_actor = nx.read_edgelist('datasets/imdb_actor_edges.tsv', nodetype=int, data=(("num_movies", int),))
uG_erdos = nx.read_edgelist('datasets/SIR_erdos_renyi.txt', nodetype=int)
uG_prefer = nx.read_edgelist('datasets/SIR_preferential_attachment.txt', nodetype=int)
```

In [ ]:

2a. For a node with d neighbors, what is its probability of getting infected in a given round?

We know that a node can either be susceptible, infected or recoverd.

Let, for a given round,
the number of nodes in susceptible set is $= S$,
number of nodes in infected set is $= I$
and the number of nodes in recovered set is $= R$
therefore, total number of nodes, $n = S + I + R$

To get infected, the node has to be in the susceptible set and probability of a node in susceptible set is, $p(S) = \frac{S}{S+I+R}$

Also, a node can be infected only by its neighbors who are already infected,

Given, the node has d neighbors, probable number of infected neighbor, $n(neigh\_I) = \frac{d*I}{S+I+R}$

Each of these infected neighbor can infect that node with a probability of = β

so, the probability of that node getting infected is $= p(S) * n(neigh\_I) * \beta$
$= \beta * \frac{S}{S+I+R} * \frac{d*I}{S+I+R}$

2b. Run 100 simulations of SIR model with β = 0.05 and δ = 0.5 for each of the three graphs.

In [66]:
```
n=100
list_percent_actor = []
list_percent_erdos = []
list_percent_prefer = []
```

In [67]:
```
for sim in range(n):
    s1, i1, r1 = algorithm_1(uG_actor)
    p1 = r1/(s1+i1+r1)
    list_percent_actor.append(p1)
```

In [68]:
```
for sim in range(n):
    s2, i2, r2 = algorithm_1(uG_erdos)
    p2 = r2/(s2+i2+r2)
    list_percent_erdos.append(p2)
```

In [ ]:
```
for sim in range(n):
    s3, i3, r3 = algorithm_1(uG_prefer)
    p3 = r3/(s3+i3+r3)
    list_percent_prefer.append(p3)
```

In [ ]:

In [94]:
```
e1 = np.sum([1 for pa in list_percent_actor if pa >= 0.5])
e2 = np.sum([1 for pe in list_percent_erdos if pe >= 0.5])
e3 = np.sum([1 for pp in list_percent_prefer if pp >= 0.5])
```

Proportion of Epidemics

In [124]:
```
print("Proportion of simulations that infected at least 50% of the network (graph - Actor) : ", e1,
'%')
print("Proportion of simulations that infected at least 50% of the network (graph - Erdos) : ", e2,
'%')
print("Proportion of simulations that infected at least 50% of the network (graph - Preferential attac
hment) : ", e3, '%')
```

```
Proportion of simulations that infected at least 50% of the network (graph - Actor) :  60 %
Proportion of simulations that infected at least 50% of the network (graph - Erdos) :  69 %
Proportion of simulations that infected at least 50% of the network (graph - Preferential attachment)
:  74 %
```

In [ ]:

Pairwise Chi-Square Test (test statistic and p-values)

In [125]:
```
### (e1,e2), (e1,e3), (e2,e3)
from scipy.stats import chi2_contingency

X_e1_e2 = chi2_contingency([[e1, 100-e1], [e2, 100-e2]])
X_e1_e3 = chi2_contingency([[e1, 100-e1], [e3, 100-e3]])
X_e2_e3 = chi2_contingency([[e2, 100-e2], [e3, 100-e3]])
```

In [126]:
```
# print(X_e1_e2,X_e1_e3,X_e2_e3)
```

In [127]:
```
print("Pair (Actor-Erdos): chi-square statistic = ",X_e1_e2[0], " and p-value = ", X_e1_e2[1])
```

```
Pair (Actor-Erdos): chi-square statistic =  1.3975324817119774  and p-value =  0.23713715085236137
```

In [128]:
```
print("Pair (Actor-Preferential): chi-square statistic = ",X_e1_e3[0], " and p-value = ", X_e1_e3[1])
```

```
Pair (Actor-Preferential): chi-square statistic =  3.821800090456807  and p-value =  0.05058986011471
8624
```

In [129]:
```
print("Pair (Erdos-Preferential): chi-square statistic = ",X_e2_e3[0], " and p-value = ", X_e2_e3[1])
```

```
Pair (Erdos-Preferential): chi-square statistic =  0.39258986627407677  and p-value =  0.530941190070
1313
```

In [ ]:

Ques-Ans about the two synthetic (Erdos, Preferential) networks:

1. Does the Erdos-Renyi graph appear to be more/less susceptible to epidemics than the Preferential Attachment graph?

In [136]:
```
if (e2 > e3):
    print("The Erdos-Renyi (", e2, "%) graph appears to be MORE susceptible to epidemics than preferen
tial attachment (",e3,"%).")
elif (e2 == e3):
    print("The Erdos-Renyi (", e2, "%) graph appears to be EQUALLY susceptible to epidemics than prefe
rential attachment (",e3,"%).")
else:
    print("The Erdos-Renyi (", e2, "%) graph appears to be LESS susceptible to epidemics than preferen
tial attachment (",e3,"%).")
```

```
The Erdos-Renyi ( 69 %) graph appears to be LESS susceptible to epidemics than preferential attachmen
t ( 74 %).
```

In [ ]:

2. In cases where an epidemic does take off, does Erdos-Renyi graph appear to have higher/lower final percentage infected?

In [137]:
```
c_erdos = 0
c_prefer = 0
for i in range(100):
    if list_percent_erdos[i] >= 0.5 and list_percent_prefer[i] >= 0.5: # condition where epidemic take
s off
        if list_percent_erdos[i] > list_percent_prefer[i]:
            c_erdos +=1
        else:
            c_prefer +=1
print("In cases where an epidemic does take off --->")
if (c_erdos > c_prefer):
    print("Erdos-Renyi graph appears to have HIGHER final percentage infected in most cases.")
else:
    print("Erdos-Renyi graph appears to have LOWER final percentage infected in most cases.")
```

```
In cases where an epidemic does take off --->
Erdos-Renyi graph appears to have HIGHER final percentage infected in most cases.
```

In [ ]:

3. Overall, which of these two networks seems to be more susceptible to the spread of disease?

In [138]:
```
avg_erdos = np.mean(list_percent_erdos)
avg_prefer = np.mean(list_percent_prefer)
```

In [140]:
```
print(avg_erdos,avg_prefer)
```

```
0.5582118677817602 0.5491079251294306
```

In terms of epidemics, we have seen that erdos-renyi is **slightly less susceptible** than preferential attachment graph.
But, in those cases, erdos-renyi network used to converge with a **higher percentage of final infected population** than the preferential attachment.
Overall (including epidemic and non-epidemic cases both), if we take the mean % of the infected population after those 100 simulations, both of the networks (erdos-renyi and preferential attachment) have very close results where the erdos-renyi graph seenms slightly more susceptible by a very small margin.

In [ ]:

4. Give one good reason why you might expect to see these significant differences (or lack thereof) between Erdos-Renyi and Preferential Attachment? (2–3 sentences)

In [145]:
```
print(e2, e3)
print("Pair (Erdos-Preferential): chi-square statistic = ",X_e2_e3[0], " and p-value = ", X_e2_e3[1])
```

```
69 74
Pair (Erdos-Preferential): chi-square statistic =  0.39258986627407677  and p-value =  0.530941190070
1313
```

In [ ]:

We see that Preferential Attachment graph use to be slightly more susceptible to epidemics, but the statistical significance test comes up with a smaller test statistics. It appears from the results of these 100 simulations that the two observed graph sequences are not much far from each other.

The differences we have seen in these two graph networks, is more related to the structure through which these graph used to form (they have different strategies to form the network). And as we see that these differences are not much significant might be due to the fact that they both are simulated networks with similar graph properties and we see the test statistics is quite higher when we compare any of them with the real (Actor) network.

In [ ]:

In [ ]:

In [ ]:

In [ ]: