Group Name : Social outlet

SNA Project Report- Social circles: Twitter

Team Members:

1. Deeya Gupta: 18ucc072
2. Vani Agarwal: 18ucs098
3. Sameer Gupta: 18ucs008
4. Abhay singhal: 18ucs011

Question 1 - Dataset 1

Dataset : Social circles: Twitter

Twitter is an American microblogging and social networking service on which users post and interact with messages known as "tweets". Registered users can post, like and retweet tweets.

Dataset Information

This dataset consists of 'circles' (or 'lists') from Twitter. Twitter data was crawled from public sources. The dataset includes node features (proles), circles, and ego networks.

Attributes

Files:

1. nodeId.edges : The edges in the ego network for the node 'nodeId'. Edges are undirected for facebook, and directed (a follows b) for twitter and gplus. The 'ego' node does not appear, but it is assumed that they follow every node id that appears in this le.
2. nodeId.circles : The set of circles for the ego node. Each line contains one circle, consisting of a series of node ids. The rst entry in each line is the name of the circle.
3. nodeId.feat : The features for each of the nodes that appears in the edge le.
4. nodeId.egofeat : The features for the ego user.
5. nodeId.featnames : The names of each of the feature dimensions. Features are '1' if the user has this property in their prole, and '0' otherwise. This le has been anonymized for facebook users, since the names of the features would reveal private data.1.

Data Description

Nodes 81306

Edges 1768149

Nodes in largest WCC 81306 (1.000)

Edges in largest WCC 1768149 (1.000)

Nodes in largest SCC 68413 (0.841)

Edges in largest SCC 1685163 (0.953)

Average clustering coefficient 0.5653

Number of triangles 13082506

Fraction of closed triangles 0.06415

Diameter (longest shortest path) 7

90-percentile effective diameter 4.5

```
import random
import matplotlib.pyplot as plt
import networkx as nx
import numpy as np
```

```
import pandas as pd
import seaborn as sns
import numpy as np
plt.style.use('fivethirtyeight')
import pylab as plt
G1 =nx.read_edgelist("twitter_combined.txt", create_using = nx.Graph(), nodetype=int)


matrix = pd.read_csv('twitter_combined.txt')


matrix.describe()
```

	214328887 34428380
count	2420765
unique	1768149
top	43003845 40981798
freq	78

Random Sampling

Due to very large dataset following code was executed to sample the data set randomly:

```
#Random Sampling

filename = 'twitter_combined.txt'
result=[]

f = open("data.txt","w")

with open(filename) as fh:
 data = fh.readlines()

for line in data:
 if line:
 words = line.split()
 result.append(words)

sample = random.sample(result, 5000)

for item in sample:
 f.write(item[0]+" "+item[1]+"\n")
```

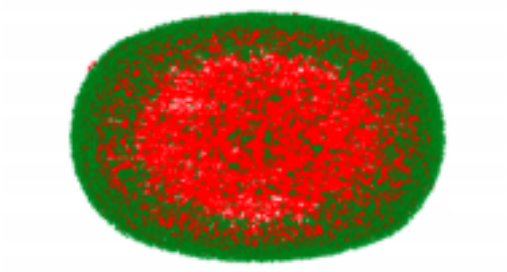
No. of Nodes and Edges

The number of nodes and edge used from the dataset are:

```
#Finding Edges and Nodes

G = nx.read_edgelist('data.txt',nodetype=int, create_using=nx.DiGraph())
print(nx.info(G))
```

```
     Name:
     Type: DiGraph
     Number of nodes: 7086
     Number of edges: 4896
     Average in degree: 0.6909
     Average out degree: 0.6909
```

```
nx.draw(G, pos=None, node_color='g',edge_color='r',node_size=7)
plt.show()
```



Centrality Measures

Centrality measures are a vital tool for understanding networks, often also known as graphs.

These algorithms use graph theory to calculate the importance of any given node in a network. They cut through noisy data, revealing parts of the network that need attention – but they all work differently.

Let's look at some social network analysis measures, how they work, and when to use them:

1. Degree Centrality
2. Eigenvector Centrality
3. Katz Centrality
4. PageRank
5. Betweenness Centrality
6. Closeness Centrality

Degree Centrality

Degree centrality assigns an importance score based simply on the number of links held by each node. The degree is a simple centrality measure that counts how many neighbors a node has. If the net- work is directed, we have two versions of the measure: *in-degree* is the number of incoming links or the number of predecessor nodes.

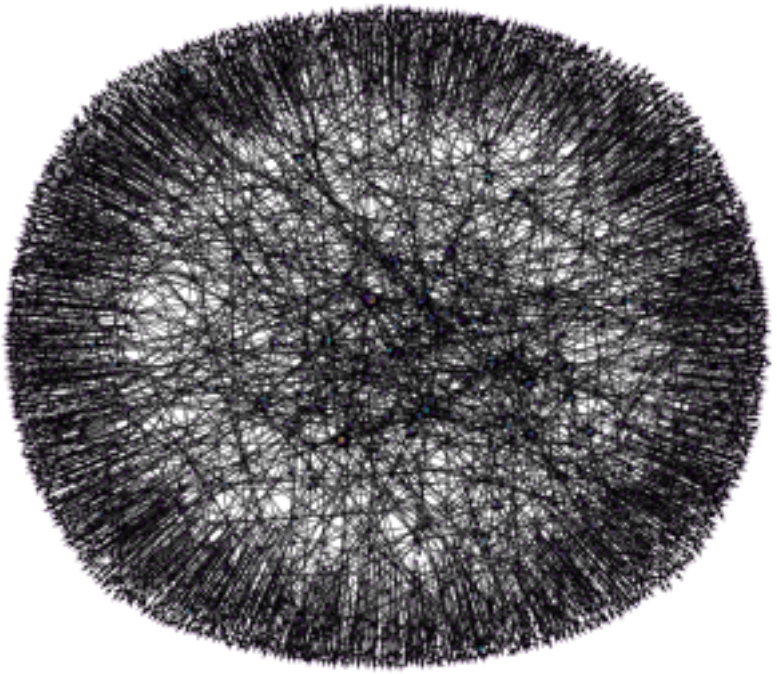
out-degree is the number of outgoing links or the number of successor nodes.

Typically, we are interested in in-degree,since in-links are given by other nodes in the network, while out-links are determined by the node itself.

Degree centrality thesis reads as follows: A node is important if it has many neighbors, or, in the directed case, if there are many other nodes that link to it, or if it links to many other nodes.

```
pos = nx.spring_layout(G)
degCent = nx.degree_centrality(G)
node_color = [20000.0 * G.degree(v) for v in G]
node_size = [v * 10000 for v in degCent.values()]
plt.figure(figsize=(15,15))
nx.draw_networkx(G, pos=pos, with_labels=False,
 node_color=node_color,
 node_size=node_size )
plt.axis('off')
```

```
sorted(degCent, key=degCent.get, reverse=True)[:5]
     [40981798, 22462180, 43003845, 7860742, 3359851]
```



```
deg_in = G.degree()
print("The nodes with maximum in degree is:")

i = max(nx.in_degree_centrality(G), key=(nx.in_degree_centrality(G)).get)
val = deg_in[i]

print("\nNOde Degree")
for p,r in deg_in:
 if r == val:
 print(str(p)+"\t"+str(r))

print("\nList of some nodes with their degree:")
print("\nNode Degree")
w = 1
for p,r in deg_in:
 if w <= 5:
 print(str(p)+"\t"+str(r))
 w=w+1
```

```
The nodes with maximum in degree is:

NOde Degree
40981798 26

List of some nodes with their degree:

Node Degree
204089551 4
63485337 10
444465215 1
294198566 4
439788025 4
```

Inference: According to in degree centrality there are several nodes having 26 in degree which is the maximum in degree.

```
deg_out = G.degree()
print("The nodes with maximum degree centrality is :")
i = max(nx.out_degree_centrality(G),key=(nx.out_degree_centrality(G)).get)
val = deg_out[i]
print("Node Degree")
for p,r in deg_out:
 if r == val:
 print(str(p)+"\t"+str(r))

print("\nList of some nodes with their degree:")
       w = 1
```

```
print("\nNode Degree")
for p,r in deg_out:
 if(w<=5):
 print(str(p)+"\t"+str(r))
 w=w+1
```

```
The nodes with maximum degree centrality is :
Node Degree
18776017 15
270449528 15

List of some nodes with their degree:

Node Degree
204089551 4
63485337 10
444465215 1
294198566 4
439788025 4
```

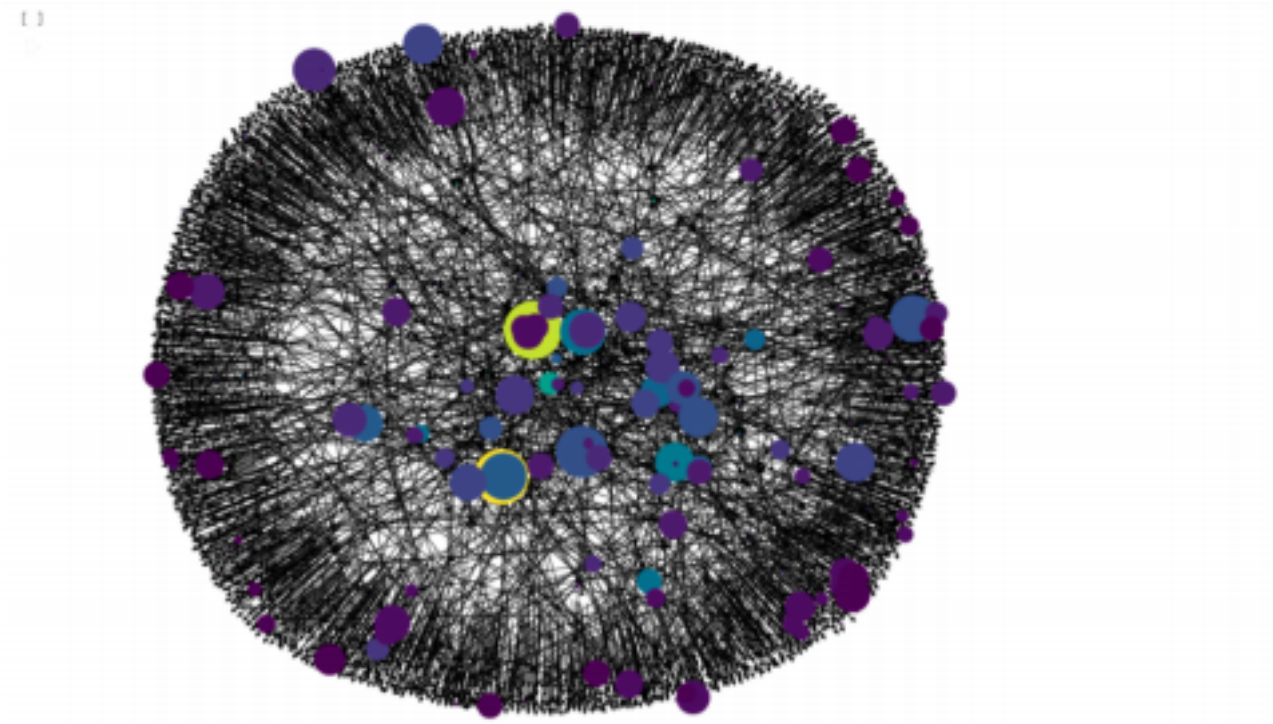
Result : We observe that the highest degree is “15” .

When to use Degree Centrality: For nding very connected individuals, popular individuals, individuals who are likely to hold most information or individuals who can quickly connect with the wider network

Eigenvector Centrality

Eigenvector centrality (also called eigencentrality) is a measure of the inuence of a node in a network. A high eigenvector score means that a node is connected to many nodes who themselves have high scores.Eigen Centrality can identify nodes with influence over the whole network, not just those directly connected to it.

```
#pos = nx.spring_layout(G)
eigCent = nx.eigenvector_centrality(G)
node_color = [20000.0 * G.degree(v) for v in G]
node_size = [v * 10000 for v in eigCent.values()]
plt.figure(figsize=(15,15))
nx.draw_networkx(G, pos=pos, with_labels=False,
 node_color=node_color,
 node_size=node_size )
plt.axis('off')
```



```
(-1.2094474756717681,
```

```
      1.2041853392124175,
      -1.2085665011405944,
      1.203864767551422)
```

```
eigen_dict = nx.eigenvector_centrality_numpy(G)
max_val = max(eigen_dict.values())
print("The nodes with maximum eigen centrality are :")
print("Node EigenVector Centrality")
for nid in eigen_dict.keys():
 if(eigen_dict[nid]==max_val):
 print(f'{nid}\t {eigen_dict[nid]}')
print("\nSome of the nodes with their EigenVector Centrality are:\n")
print("Node\tEigen Vector Centrality")
eigen_dict_sorted=sorted(eigen_dict, key=eigen_dict.get, reverse=True)
w = 1
for r in eigen_dict:
 if(w<=5):
 w = w+1
 print(str(r)+"\t "+str(eigen_dict[r]))
```

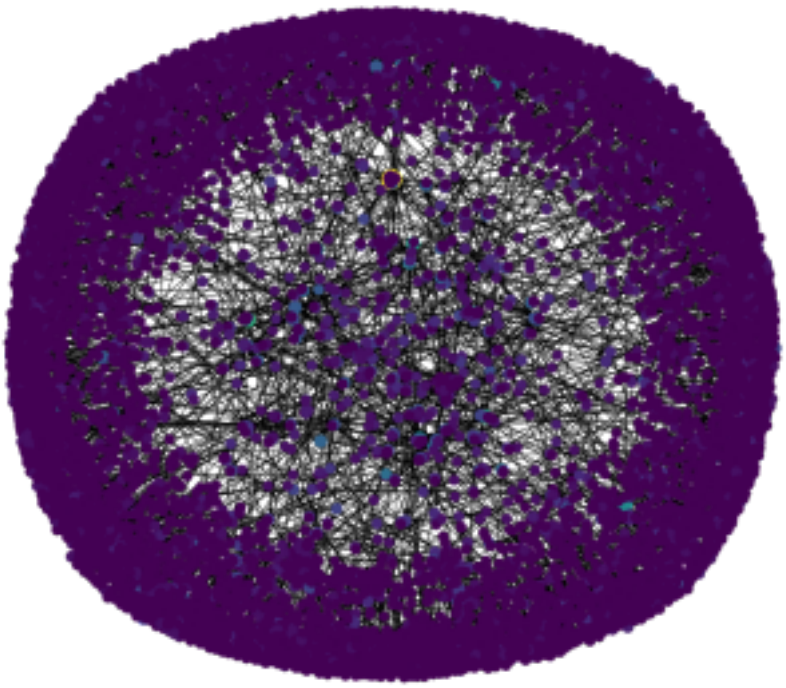
```
     The nodes with maximum eigen centrality are :
     Node EigenVector Centrality
     22462180 0.31602716820505944

     Some of the nodes with their EigenVector Centrality are:

     Node EigenVector Centrality
     204089551 6.528175468704939e-19
     63485337 -4.2761012985610506e-18
     444465215 -4.550769464534646e-19
     294198566 2.9540929189312838e-18
     439788025 3.1946361415931253e-19
```

Inference: Node 22462180 has the maximum eigenvector centrality. So, 22462180 has the most influence in the network. When to use it: EigenCentrality is a good ‘all-round’ SNA score, handy for understanding human social networks, but also for understanding networks like malware propagation

Katz Centrality

Katz centrality computes the relative influence of a node within a network by measuring the num- ber of the immediate neighbors (rst degree nodes) and also all other nodes in the network that connect to the node under consideration through these immediate neighbors.

```
pos = nx.spring_layout(G)
degCent = nx.katz_centrality(G)
node_color = [20000.0 * G.degree(v) for v in G]
node_size = [v * 10000 for v in degCent.values()]
plt.figure(figsize=(15,15))
nx.draw_networkx(G, pos=pos, with_labels=False,
 node_color=node_color,
 node_size=node_size )
plt.axis('off')

sorted(degCent, key=degCent.get, reverse=True)[:5]
     [40981798, 22462180, 43003845, 34428380, 15913]
```



```
katz_dict = nx.katz_centrality(G, max_iter=20000)
max_index = max(nx.katz_centrality(G, max_iter=20000), key=(nx.katz_centrality(G, max_iter=20000)).get)
max_val = katz_dict[max_index]

print("The node with maximum Katz centrality are :")
print("\nNode Degree")
for nid in katz_dict:
 if(katz_dict[nid]==max_val):
 print(f'{nid}\t{katz_dict[nid]}')

print("\nSome nodes with their Katz Centrality are:")
print("\nNode Katz Centrality")
w = 1
for nid in katz_dict:
 if(w<=8):
 print(f'{nid}\t{katz_dict[nid]}')
 w = w+1
```

```
The node with maximum Katz centrality are :

Node Degree
40981798 0.04229876399113487

Some nodes with their Katz Centrality are:

Node Katz Centrality
204089551 0.013332882306528869
63485337 0.022792512758857193
444465215 0.01100981197896686
294198566 0.015523834890343272
439788025 0.013211774374760232
535437378 0.012330989416442884
150334831 0.01100981197896686
14602259 0.012110793176863548
```

Inference : Node 40981798 has the maximum Katz centrality. So 40981798 are the node with most influence in the network.

Page Rank Centrality

PageRank is a link analysis algorithm and it assigns a numerical weighting to each element of a hyperlinked set of documents, such as the World Wide Web, with the purpose of “measuring” its relative importance within the set. The algorithm may be applied to any collection of entities with reciprocal quotations and references. PageRank is an adjustment of Katz centrality that takes into consideration this issue. There are three distinct factors that determine the PageRank of a node: (i) the number of links it receives, (ii) the link propensity of the linkers, and (iii) the centrality of the linkers

```
pg_dict = nx.pagerank(G, max_iter=20000)
max_val = max(pg_dict.values())
print("The nodes with maximum Page Rank centrality are :")
print("\nNode PageRank Centrality")

for nid in pg_dict.keys():
 if(pg_dict[nid]==max_val):
 print(f'{nid}\t{pg_dict[nid]}')
print("\nSome nodes with their Pagerank Centrality are:")
print("\nNode PageRank Centrality")
w = 1
for nid in pg_dict:
 if(w<=8):
 print(f'{nid}\t{pg_dict[nid]}')
 w = w + 1
```

```
The nodes with maximum Page Rank centrality are :

Node Page Rank Centrality
43003845 0.0046362437454150855

Some nodes with their Pagerank Centrality are:

Node Page Rank Centrality
204089551 0.00021335755694339227
63485337 0.0008795736982189135
444465215 8.425580633508312e-05
294198566 0.0003303969436249198
439788025 0.0001438678026815703
535437378 0.00014566684382260193
150334831 8.425580633508312e-05
14602259 0.00015579020195086776
```

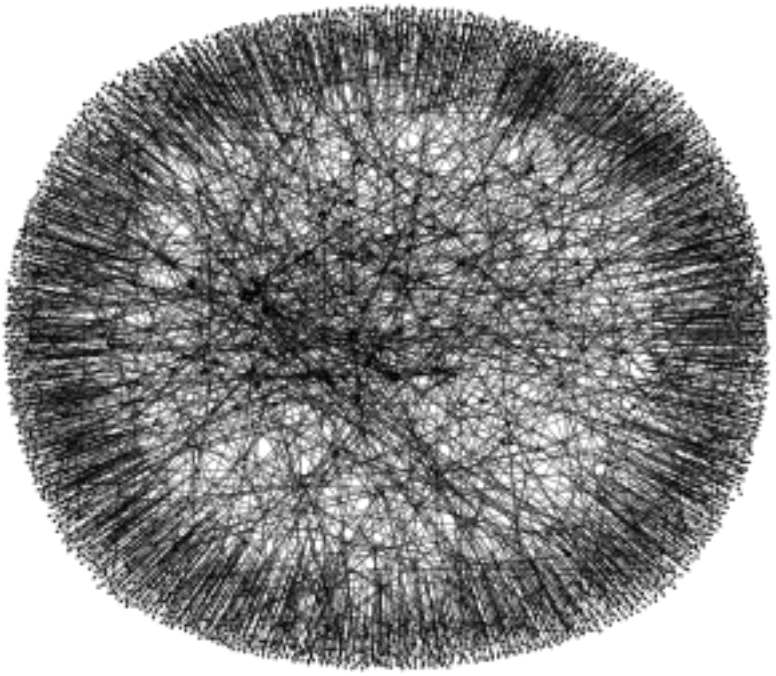
Inference : Node 43003845 has the maximum Page Rank centrality.

Betweenness Centrality

Betweenness centrality is a widely used measure that captures a person's role in allowing information to pass from one part of the network to the other.Betweenness centrality is a measure of the inuence of a vertex over the ow of information be- tween every pair of vertices.

```
pos = nx.spring_layout(G)
betCent = nx.betweenness_centrality(G, normalized=True, endpoints=True)
node_color = [20000.0 * G.degree(v) for v in G]
node_size = [v * 10000 for v in betCent.values()]
plt.figure(figsize=(20,20))
nx.draw_networkx(G, pos=pos, with_labels=False,
 node_color=node_color,
 node_size=node_size )
plt.axis('off')
sorted(betCent, key=betCent.get, reverse=True)[:5]
```

```
[43003845, 22462180, 46209291, 259842341, 276308596]
```



```
bet_dict = nx.betweenness_centrality(G)
max_val = max(bet_dict.values())
print("The nodes with maximum Betweenness Centrality are :")
print("\nNode Betweenness Centrality")
for nid in bet_dict:
 if(bet_dict[nid]==max_val):
 print(f'{nid}\t{bet_dict[nid]}')

print("\nSome nodes with their Betweenness Centrality are:")
print("\nNode Betweenness Centrality")
w = 1
for r in bet_dict:
 if(w<=5):
 w = w+1
 print(str(r)+"\t"+str(bet_dict[r]))
```

```
The nodes with maximum Betweenness Centrality are :

Node Betweenness Centrality
43003845 0.00033929990081337717

Some nodes with their Betweenness Centrality are:

Node Betweenness Centrality
204089551 1.5939385704044658e-07
63485337 0.0
444465215 0.0
294198566 0.0
439788025 1.5507694007893447e-06
```

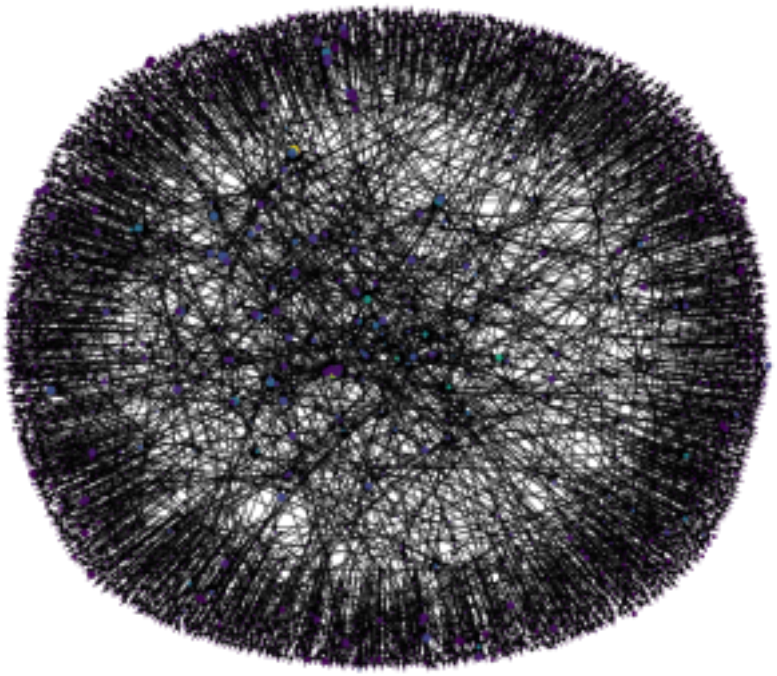
Node 16098603 has the maximum Betweenness Centrality. So,16098603 is at a central position in the network and it provides the fastest ow of information across the network.

Closeness Centrality

Closeness centrality indicates how close a node is to all other nodes in the network. It is calculated as the average of the shortest path length from the node to every other node in the network.

```
pos = nx.spring_layout(G)
cloCent = nx.closeness_centrality(G)
node_color = [20000.0 * G.degree(v) for v in G]
node_size = [v * 10000 for v in cloCent.values()]
plt.figure(figsize=(13,13))
nx.draw_networkx(G, pos=pos, with_labels=False,
 node_color=node_color,
 node_size=node_size )
plt.axis('off')
sorted(cloCent, key=cloCent.get, reverse=True)[:5]
```

```
[40981798, 22462180, 43003845, 173732041, 304462046]
```



```
clo_dict = nx.closeness_centrality(G)
max_val = max(clo_dict.values())

print("The nodes with maximum Closeness Centrality are :")
print("\nNode Closeness Centrality")
for nid in clo_dict:
 if(clo_dict[nid]==max_val):
 print(f'{nid}\t{clo_dict[nid]}')

print("\nSome nodes with their Closeness Centrality are:")

print("\nNode Closeness Centrality")
dict_sorted_keys = sorted(clo_dict, key=clo_dict.get, reverse=True)

w = 1
for r in dict_sorted_keys:
 if(w<=5):
```

```
 w = w+1
 print(str(r)+"\t"+str(clo_dict[r]))
```

```
     The nodes with maximum Closeness Centrality are :

     Node Closeness Centrality
     40981798 0.007956990599355897

     Some nodes with their Closeness Centrality are:

     Node Closeness Centrality
     40981798 0.007956990599355897
     22462180 0.00794813966431768
     43003845 0.00701214676959921
     173732041 0.006772869723432792
     304462046 0.0067155804118624015
```

Inference : Node 40981798 has the maximum Closeness Centrality. So, 40981798 can spread the information across the network most recently.

Clustering Coecient

Local Clustering Coefficient The local clustering coefficient of a vertex (node) in a graph quantities how close its neighbours are to being a clique.

```
print('Average Local Clustering Coefficient is:')
print(nx.average_clustering(G))

print('\nSome of the nodes with their clustering coefficient are:')
print('\nNode Local Clustering Coefficient')

w = 1
for nid, val in nx.clustering(G).items():
 if(w<=8):
 print(f'{nid}\t{val}')
 w = w + 1
```

```
     Average Local Clustering Coefficient is:
     0.0004959910464450251

     Some of the nodes with their clustering coefficient are:

     Node Local Clustering Coefficient
     204089551 0
     63485337 0
     444465215 0
     294198566 0
     439788025 0
     535437378 0
     150334831 0
     14602259 0
```

Inference : Average Local Clustering is : 0.0004959910464450251

Global Clustering Coefficient

The global clustering coefficient is Number of closed triplets (or 3 x triangles) over Total number of triplets.

```
print("The Global Clustering Coefficient is:",end="")
print(np.mean(list(nx.clustering(G).values())))
```

```
     The Global Clustering Coefficient is:0.0004959910464450252
```

Reciprocity

Reciprocity refers to responding to a positive action with another positive action. It creates, main- tains and strengthens various social bounds.

```
print("The reciprocity is:",end="")
print(nx.reciprocity(G))
```

```
The reciprocity is:0.0032679738562091504
```

Transitivity

Transitivity refers to the extent to which the relation that relates two nodes in a network that are connected by an edge.

```
print("The transitivity is:",end="")
print(nx.transitivity(G))
```

```
The transitivity is:0.00510204081632653
```

Question 1 - Dataset 2

Dataset : Epinions social network

This is a who-trust-whom online social network of a a general consumer review site Epinions.com. Members of the site can decide whether to "trust" each other. All the trust relationships interact and form the Web of Trust which is then combined with review ratings to determine which reviews are shown to the user.

Data Description

Nodes 75879

Edges 508837

Nodes in largest WCC 75877 (1.000)

Edges in largest WCC 508836 (1.000)

Nodes in largest SCC 32223 (0.425)

Edges in largest SCC 443506 (0.872)

Average clustering coefficient 0.1378

Number of triangles 1624481

Fraction of closed triangles 0.0229

Diameter (longest shortest path) 14

90-percentile effective diameter 5

```
import random
import matplotlib.pyplot as plt
import networkx as nx
import numpy as np
import pandas as pd
import seaborn as sns
import numpy as np
plt.style.use('fivethirtyeight')
import pylab as plt
```

Random Sampling

Due to very large dataset following code was executed to sample the data set randomly:

```
#Random Sampling

filename = 'soc-Epinions1.txt'
result=[]

f = open("data.txt","w")

with open(filename) as fh:
 data = fh.readlines()

for line in data:
 if line:
 words = line.split()
 result.append(words)

sample = random.sample(result, 3000)

for item in sample:
 f.write(item[0]+" "+item[1]+"\n")
```

No. of Nodes and Edges

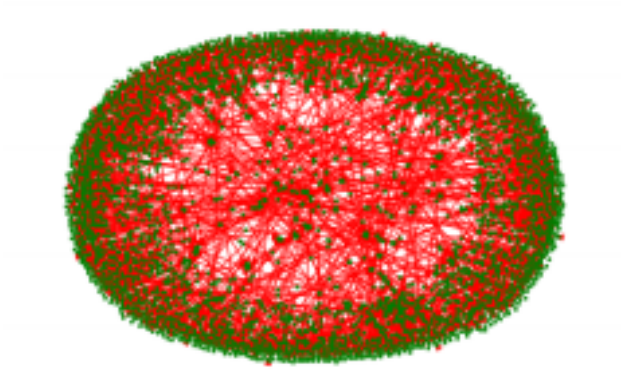
The number of nodes and edge used from the dataset are:

```
#Finding Edges and Nodes
G = nx.read_edgelist('data.txt',nodetype=int, create_using=nx.DiGraph())
print(nx.info(G))
```

```
     Name:
     Type: DiGraph
     Number of nodes: 3533
     Number of edges: 2428
     Average in degree: 0.6872
     Average out degree: 0.6872
```

Inference: So, Number of nodes are 3032 and Number of edges are 1992

```
nx.draw(G, pos=None, node_color='g',edge_color='r',node_size=7)
plt.show()
```



Centrality Measures

Centrality measures are a vital tool for understanding networks, often also known as graphs.

These algorithms use graph theory to calculate the importance of any given node in a network. They cut through noisy data, revealing parts of the network that need attention – but they all work differently.

Let's look at some social network analysis measures, how they work, and when to use them:

1. Degree Centrality
2. Eigenvector Centrality
3. Katz Centrality
4. PageRank
5. Betweenness Centrality
6. Closeness Centrality

Degree Centrality

Degree centrality assigns an importance score based simply on the number of links held by each node. The degree is a simple centrality measure that counts how many neighbors a node has. If the net- work is directed, we have two versions of the measure: *in-degree* is the number of incoming links or the number of predecessor nodes.

out-degree is the number of outgoing links or the number of successor nodes.

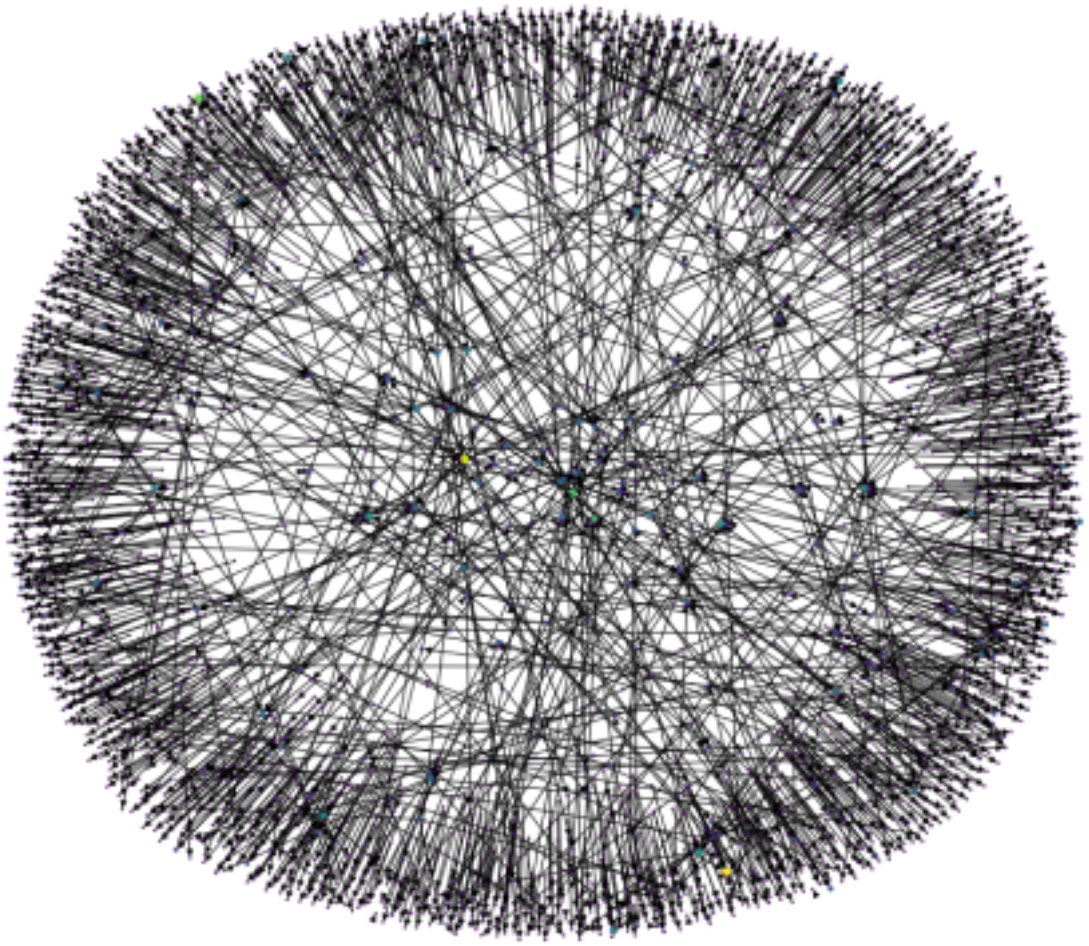
Typically, we are interested in in-degree,since in-links are given by other nodes in the network, while out-links are determined by the node itself.

Degree centrality thesis reads as follows: A node is important if it has many neighbors, or, in the directed case, if there are many other nodes that link to it, or if it links to many other nodes.

```
pos = nx.spring_layout(G)
degCent = nx.degree_centrality(G)
node_color = [20000.0 * G.degree(v) for v in G]
node_size = [v * 10000 for v in degCent.values()]
plt.figure(figsize=(15,15))
nx.draw_networkx(G, pos=pos, with_labels=False,
 node_color=node_color,
 node_size=node_size )
plt.axis('off')
```

```
   t d(d C t k d C t t T )[ 5]
sorted(degCent, key=degCent.get, reverse=True)[:5]
```

```
[18, 645, 143, 40, 763]
```



```
deg_in = G.degree()
print("The nodes with maximum in degree is:")

i = max(nx.in_degree_centrality(G), key=(nx.in_degree_centrality(G)).get)
val = deg_in[i]

print("\nNOde Degree")
for p,r in deg_in:
 if r == val:
 print(str(p)+"\t"+str(r))

print("\nList of some nodes with their degree:")
print("\nNode Degree")
w = 1
for p,r in deg_in:
 if w <= 5:
 print(str(p)+"\t"+str(r))
 w=w+1
```

```
The nodes with maximum in degree is:

NOde Degree
18 15

List of some nodes with their degree:

Node Degree
1424 1
3879 3
11555 1
1398 8
8106 1
```

Inference: According to in degree centrality there are several nodes having 15 in degree which is the maximum in degree.

```
deg_out = G.degree()
print("The nodes with maximum degree centrality is :")
i = max(nx.out_degree_centrality(G),key=(nx.out_degree_centrality(G)).get)
val = deg_out[i]
print("Node Degree")
for p,r in deg_out:
 if r == val:
 print(str(p)+"\t"+str(r))

print("\nList of some nodes with their degree:")
w = 1
print("\nNode Degree")
for p,r in deg_out:
 if(w<=5):
 print(str(p)+"\t"+str(r))
 w=w+1
```

```
The nodes with maximum degree centrality is :
Node Degree
40 11
763 11

List of some nodes with their degree:

Node Degree
1424 1
3879 3
11555 1
1398 8
8106 1
```

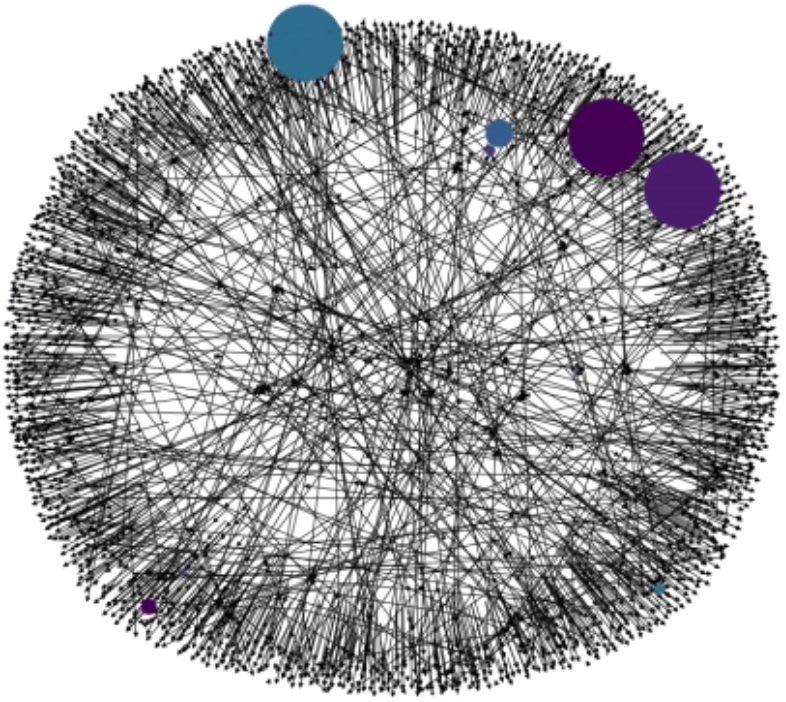
Result : We observe that the highest degree is 11. When to use Degree Centrality: For nding very connected individuals, popular individuals, individuals who are likely to hold most information or individuals who can quickly connect with the wider network

Eigenvector Centrality

Eigenvector centrality (also called eigencentrality) is a measure of the inuence of a node in a network. A high eigenvector score means that a node is connected to many nodes who themselves have high scores.Eigen Centrality can identify nodes with influence over the whole network, not just those directly connected to it.

```
#pos = nx.spring_layout(G)
eigCent = nx.eigenvector_centrality(G)
node_color = [20000.0 * G.degree(v)
for v in G]
node_size = [v * 10000 for v in
eigCent.values()]
plt.figure(figsize=(15,15))
nx.draw_networkx(G, pos=pos,
with_labels=False,
 node_color=node_color,
 node_size=node_size )
plt.axis('off')
```

```
(-1.2087054881453514,
 1.1963768038153648,
 -1.2020183494687082,
 1.2029028227925302)
```



```
eigen_dict = nx.eigenvector_centrality_numpy(G)
max_val = max(eigen_dict.values())
print("The nodes with maximum eigen centrality are :")
print("Node EigenVector Centrality")

for nid in eigen_dict.keys():
 if(eigen_dict[nid] == max_val):
 print(f"{nid}\t{eigen_dict[nid]}")
print("\nSome of the nodes with their EigenVector Centrality are:\n")
print("Node\tEigen Vector Centrality")


eigen_dict_sorted = sorted(eigen_dict, key=eigen_dict.get, reverse=True)
w = 1
for r in eigen_dict:
 if(w<=5):
 w = w + 1
 print(str(r)+"\t"+str(eigen_dict[r]))
```

```
The nodes with maximum eigen centrality are :
Node EigenVector Centrality
381 0.5773426053986949

Some of the nodes with their EigenVector Centrality are:

Node EigenVector Centrality
1424 -2.6795282882320468e-17
3879 2.7613190904606504e-13
11555 6.161095691823302e-17
1398 1.1917776814375459e-14
8106 -7.99383526187736e-18
```

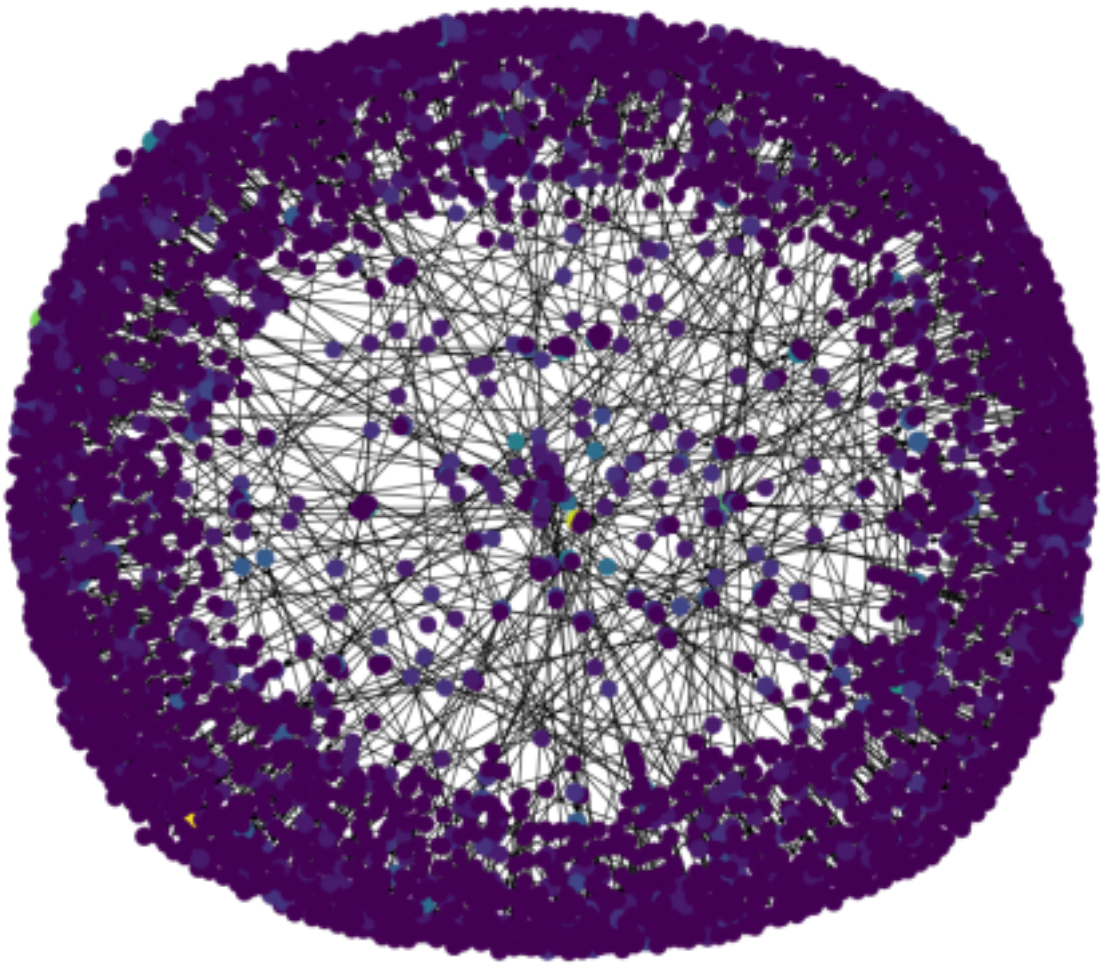
Inference: Node 381 has the maximum eigenvector centrality. So, 381 has the most influence in the network. When to use it: EigenCentrality is a good ‘all-round’ SNA score, handy for understanding human social networks, but also for understanding networks like malware propagation

Katz Centrality

Katz centrality computes the relative influence of a node within a network by measuring the num- ber of the immediate neighbors (rst degree nodes) and also all other nodes in the network that connect to the node under consideration through these immediate neighbors.

```
pos = nx.spring_layout(G)
degCent = nx.katz_centrality(G)
node_color = [20000.0 * G.degree(v) for v in G]
node_size = [v * 10000 for v in degCent.values()]
plt.figure(figsize=(15,15))
nx.draw_networkx(G, pos=pos, with_labels=False,
 node_color=node_color,
 node_size=node_size )
plt.axis('off')
sorted(degCent, key=degCent.get, reverse=True)[:5]
```

```
[18, 143, 726, 118, 1719]
```



```
katz_dict = nx.katz_centrality(G, max_iter=20000)
max_index = max(nx.katz_centrality(G, max_iter=20000), key=(nx.katz_centrality(G, max_iter=20000)).get)
max_val = katz_dict[max_index]

print("The node with maximum Katz centrality are :")
print("\nNode Degree")
for nid in katz_dict:
 if(katz_dict[nid]==max_val):
 print(f'{nid}\t{katz_dict[nid]}')

print("\nSome nodes with their Katz Centrality are:")
print("\nNode Katz Centrality")
w = 1
for nid in katz_dict:
 if(w<=8):
 print(f'{nid}\t{katz_dict[nid]}')
 w = w+1
```

```
The node with maximum Katz centrality are :

Node Degree
18 0.03988963739918542

Some nodes with their Katz Centrality are:

Node Katz Centrality
1424 0.015642995058504087
3879 0.020492323526640356
11555 0.015642995058504087
1398 0.026593091599456952
8106 0.015642995058504087
13000 0.018771594070204903
19792 0.015642995058504087
18765 0.0172072945643545
```

Inference : Node 18 has the maximum Katz centrality. So, 18 is the node with the most influence in the network.

Page Rank Centrality

PageRank is a link analysis algorithm and it assigns a numerical weighting to each element of a hyperlinked set of documents, such as the World Wide Web, with the purpose of “measuring” its relative importance within the set. The algorithm may be applied to any collection of entities with reciprocal quotations and references. PageRank is an adjustment of Katz centrality that takes into consideration this issue. There are three distinct factors that determine the PageRank of a node: (i) the number of links it receives, (ii) the link propensity of the linkers, and (iii) the centrality of the linkers

```
pg_dict = nx.pagerank(G, max_iter=20000)
max_val = max(pg_dict.values())
print("The nodes with maximum Page Rank centrality are :")
print("\nNode PageRank Centrality")

for nid in pg_dict.keys():
 if(pg_dict[nid]==max_val):
 print(f'{nid}\t{pg_dict[nid]}')
print("\nSome nodes with their Pagerank Centrality are:")
print("\nNode PageRank Centrality")
w = 1
for nid in pg_dict:
 if(w<=8):
 print(f'{nid}\t{pg_dict[nid]}')
 w = w + 1
```

```
     The nodes with maximum Page Rank centrality are :

     Node Page Rank Centrality
     18 0.002400242423470214

     Some nodes with their Pagerank Centrality are:

     Node Page Rank Centrality
     1424 0.00017411342824912707
     3879 0.0006489707622866421
     11555 0.00017411342824912707
     1398 0.001060672890927211
     8106 0.00017411342824912707
     13000 0.0003957532939186481
     19792 0.00017411342824912707
     18765 0.00024799338347230074
```

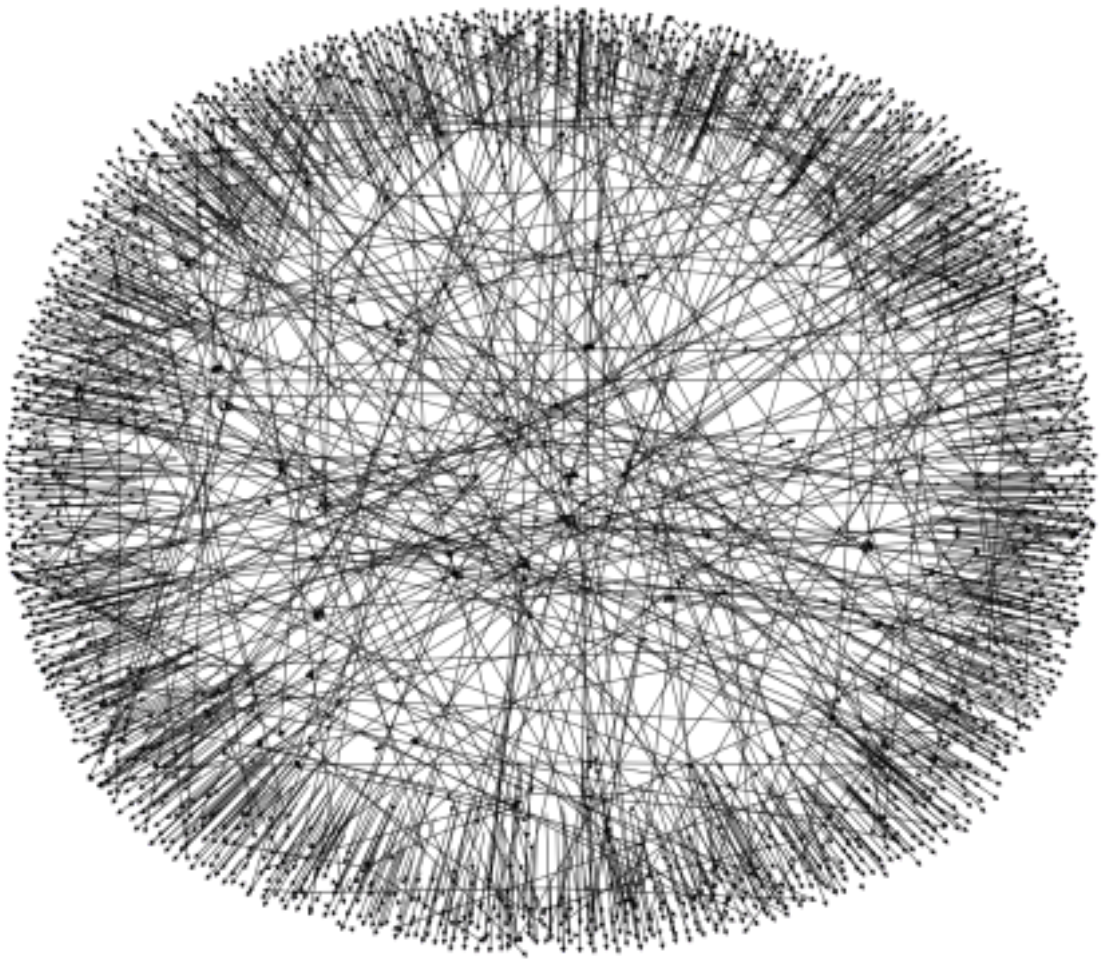
Inference : Node 18 has the maximum Page Rank centrality.

Betweenness Centrality

Betweenness centrality is a widely used measure that captures a person's role in allowing information to pass from one part of the network to the other.Betweenness centrality is a measure of the inuence of a vertex over the row of information be- tween every pair of vertices.

```
pos = nx.spring_layout(G)
betCent = nx.betweenness_centrality(G, normalized=True, endpoints=True)
node_color = [20000.0 * G.degree(v) for v in G]
node_size = [v * 10000 for v in betCent.values()]
plt.figure(figsize=(20,20))
nx.draw_networkx(G, pos=pos, with_labels=False,
 node_color=node_color,
 node_size=node_size )
plt.axis('off')
sorted(betCent, key=betCent.get, reverse=True)[:5]
```

```
     [40, 1314, 1167, 418, 19]
```



```
bet_dict = nx.betweenness_centrality(G)
max_val = max(bet_dict.values())
print("The nodes with maximum Betweenness Centrality are :")
print("\nNode Betweenness Centrality")
for nid in bet_dict:
 if(bet_dict[nid]==max_val):
 print(f'{nid}\t{bet_dict[nid]}')

print("\nSome nodes with their Betweenness Centrality are:")
print("\nNode Betweenness Centrality")
w = 1
for r in bet_dict:
 if(w<=5):
 w = w+1
 print(str(r)+"\t"+str(bet_dict[r]))
```

```
The nodes with maximum Betweenness Centrality are :

Node Betweenness Centrality
40 6.414629460532869e-06

Some nodes with their Betweenness Centrality are:

Node Betweenness Centrality
1424 0.0
3879 0.0
11555 0.0
1398 5.612800777966261e-07
8106 0.0
```

Node 40 has the maximum Betweenness Centrality. So, "40 is at a central position in the network and it provides the fastest ow of information across the network.

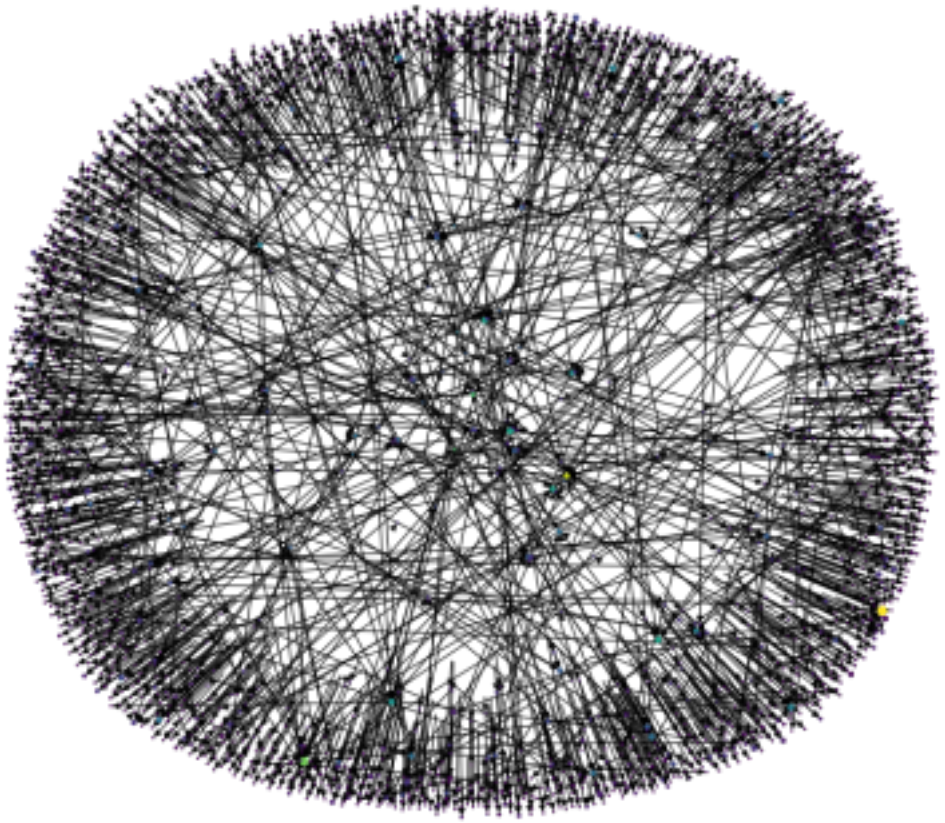
Closeness Centrality

Closeness centrality indicates how close a node is to all other nodes in the network. It is calculated as the average of the shortest path length

from the node to every other node in the network.

```
pos = nx.spring_layout(G)
cloCent = nx.closeness_centrality(G)
node_color = [20000.0 * G.degree(v) for v in G]
node_size = [v * 10000 for v in cloCent.values()]
plt.figure(figsize=(13,13))
nx.draw_networkx(G, pos=pos, with_labels=False,
 node_color=node_color,
 node_size=node_size )
plt.axis('off')
sorted(cloCent, key=cloCent.get, reverse=True)[:5]
```

```
[18, 143, 726, 3281, 118]
```



```
clo_dict = nx.closeness_centrality(G)
max_val = max(clo_dict.values())

print("The nodes with maximum Closeness Centrality are :")
print("\nNode Closeness Centrality")
for nid in clo_dict:
 if(clo_dict[nid]==max_val):
 print(f'{nid}\t{clo_dict[nid]}')

print("\nSome nodes with their Closeness Centrality are:")

print("\nNode Closeness Centrality")
dict_sorted_keys = sorted(clo_dict, key=clo_dict.get, reverse=True)

w = 1
for r in dict_sorted_keys:
 if(w<=5):
 w = w+1
 print(str(r)+"\t"+str(clo_dict[r]))
```

```
The nodes with maximum Closeness Centrality are :

Node Closeness Centrality
18 0.004530011325028313

Some nodes with their Closeness Centrality are:

Node Closeness Centrality
18 0.004530011325028313
143 0.0034177317586151106
```

```
726 0.0026844511555723333
3281 0.002359380898452246
118 0.0021778900601097657
```

Inference : Node 18 has the maximum Closeness Centrality. So, 18 can spread the information across the network most ecently.

Clustering Coecient

Local Clustering Coefficient The local clustering coefficient of a vertex (node) in a graph quantities how close its neighbours are to being a clique.

```
print('Average Local Clustering Coefficient is:')
print(nx.average_clustering(G))

print('\nSome of the nodes with their clustering coefficient are:')
print('\nNode Local Clustering Coefficient')

w = 1
for nid, val in nx.clustering(G).items():
 if(w<=8):
 print(f'{nid}\t{val}')
 w = w + 1
```

```
Average Local Clustering Coefficient is:
6.065262221503376e-05

Some of the nodes with their clustering coefficient are:

Node Local Clustering Coefficient
1424 0
3879 0
11555 0
1398 0
8106 0
13000 0
19792 0
18765 0
```

Inference : Average Local Clustering is : 6.065262221503376e-05

Global Clustering Coefficient

The global clustering coefficient is Number of closed triplets (or 3 x triangles) over Total number of triplets.

```
print("The Global Clustering Coefficient is:",end="")
print(np.mean(list(nx.clustering(G).values())))
```

```
The Global Clustering Coefficient is:6.065262221503376e-05
```

Reciprocity

Reciprocity refers to responding to a positive action with another positive action. It creates, main- tains and strengthens various social bounds.

```
print("The reciprocity is:",end="")
print(nx.reciprocity(G))
```

```
The reciprocity is:0.0
```

Transitivity

Transitivity refers to the extent to which the relation that relates two nodes in a network that are connected by an edge.

```
print("The transitivity is:",end="")
print(nx.transitivity(G))
```

```
The transitivity is:0.0008223684210526315
```

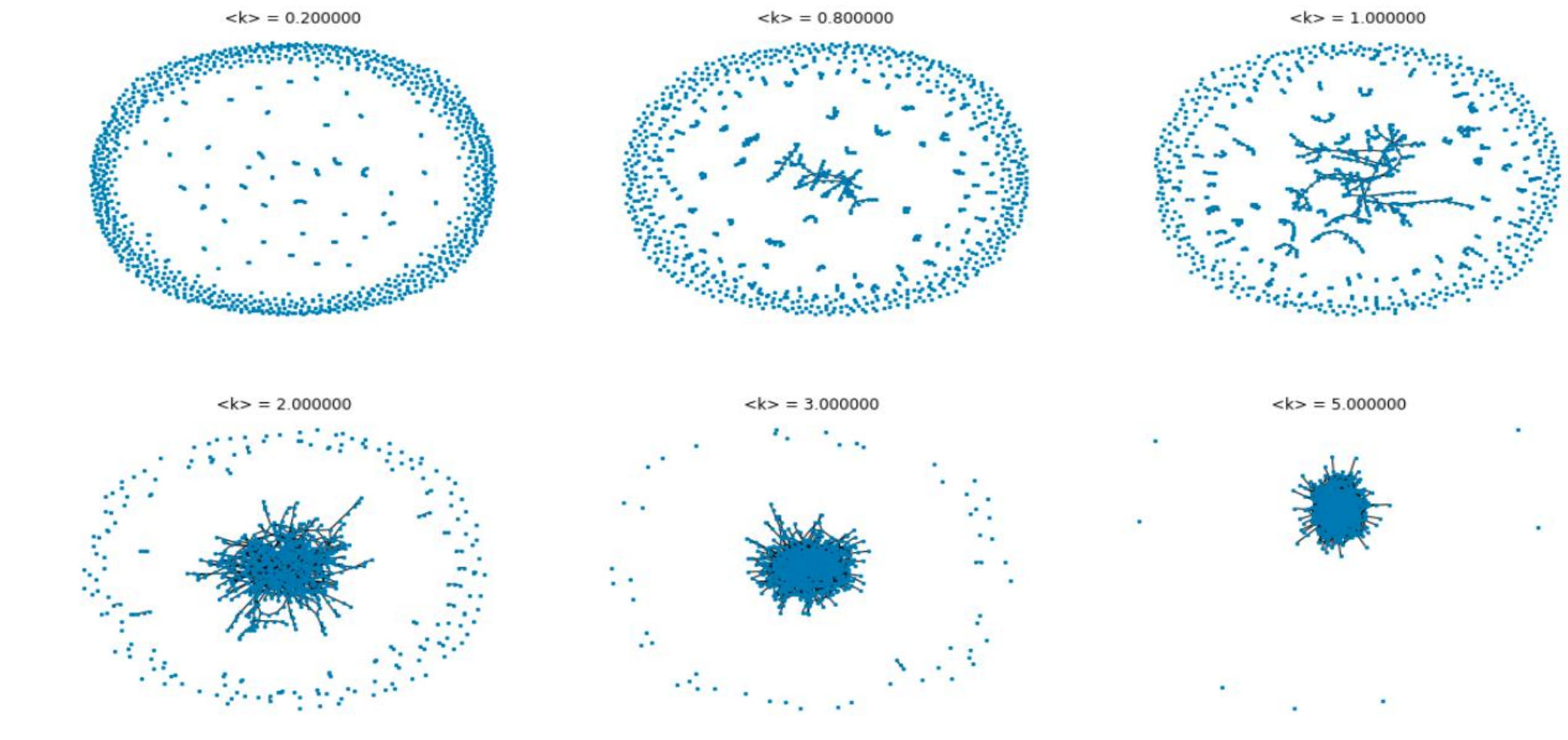
Round1(Problem 2)

```
[ ]  1  import networkx as nx
     2  import matplotlib.pyplot as plt
     3  import random
     4  from networkx.utils import not_implemented_for
     5  from itertools import chain
     6
```

Appearance of a Giant Component in Random Network

Figures below show a random network of 1000 nodes in which as we increase the value of k and after it exceeds a critical value large clusters start to form rapidly before that there are many tiny cluster formations.

```
 1  N = 1000
 2  k = [0.2,0.8,1,2,3,5]
 3  plt.figure(figsize=(20,10))
 4  pl = 1
 5  for kk in k:
 6      p = kk/(N-1)
 7      er_G = nx.erdos_renyi_graph(N,p)
 8      plt.subplot(2,3,pl)
 9      plt.title("<k> = %f"%kk)
10      pl+=1
11      nx.draw(er_G,node_size=5)
12
```



As we can see increasing the< k> (average degree) for the random network we see a bigger and bigger giant component. Initially giant components are sparse trees but as the < k> get > 3 or < k> >=5 we see a nearly connected graph.

Evolution of a Random Network

In this section we vary the average degree of each node in a random network and try to figure at what point all the nodes are inside the giant component!

```
[ ]  1  options = {
     2  'node_color': 'purple',
     3  'node_size': 40,
     4  'width': 0.1,
     5  }
     6  p=2.495/999
     7  G= nx.erdos_renyi_graph(1000,0.002)
     8  nx.draw(G,**options)
     9  plt.show()
    10
```

Above is a random graph with **p=0.002**

Below we counted the number of connected components in graph G

```
1  nx.number_connected_components(G)
```

```
195
```

Below we have the list L of sets in which each set represents a component of the graph and they are sorted respectively.

```
[ ]  1  L = sorted(nx.connected_components(G))
     2  print(sorted(L))
     3
```

```
[{0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 18, 20, 21, 23, 26, 27, 28, 29, 30, 31, 33, 34, 35, 36, 37, 38, 40,
41, 45, 46, 47, 49, 50, 51, 52, 53, 54, 55, 56, 57, 59, 60, 61, 62, 63, 64, 65, 67, 68, 70, 71, 72, 74, 75, 76, 77, 78,
79, 80, 81, 83, 84, 85, 86, 87, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 102, 103, 104, 105, 106, 107, 110, 111,
113, 114, 116, 118, 120, 121, 123, 124, 125, 126, 127, 131, 133, 134, 136, 137, 138, 139, 140, 141, 143, 147, 149, 150,
153, 154, 155, 157, 158, 163, 165, 167, 169, 171, 172, 173, 174, 175, 176, 178, 179, 182, 185, 186, 187, 189, 190, 191,
193, 195, 196, 197, 198, 201, 202, 203, 204, 205, 206, 208, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 221, 222,
224, 225, 226, 227, 229, 230, 231, 232, 233, 234, 236, 238, 239, 240, 242, 243, 244, 245, 247, 252, 253, 255, 256, 257,
259, 260, 261, 262, 264, 266, 267, 268, 270, 273, 274, 275, 276, 277, 278, 280, 281, 282, 283, 284, 285, 286, 288, 289,
290, 291, 292, 293, 294, 296, 298, 299, 300, 301, 302, 304, 305, 306, 307, 308, 309, 310, 313, 314, 316, 317, 319, 320,
321, 324, 325, 326, 327, 328, 330, 331, 332, 333, 334, 335, 338, 339, 340, 341, 342, 345, 346, 347, 348, 349, 350, 351,
352, 354, 356, 358, 360, 361, 362, 363, 366, 367, 368, 369, 370, 371, 373, 377, 378, 379, 380, 381, 382, 383, 384, 385,
386, ….., 999]
```

Now below we will take a set to print our biggest component of the graph by finding maximum length set in the list of sets L.

```
1  biggest_comp={}
2  biggest_comp_length=0
3  for a in L:
4      if(biggest_comp_length<len(a)):
5          biggest_comp_length=len(a)
6          biggest_comp=a
7  print(biggest_comp)
8
```

```
{0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 18, 20, 21, 23, 26, 27, 28, 29, 30, 31, 33, 34, 35, 36, 37, 38, 40,
41, 45, 46, 47, 49, 50, 51, 52, 53, 54, 55, 56, 57, 59, 60, 61, 62, 63, 64, 65, 67, 68, 70, 71, 72, 74, 75, 76, 77, 78,
79, 80, 81, 83, 84, 85, 86, 87, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 102, 103, 104, 105, 106, 107, 110, 111,
113, 114, 116, 118, 120, 121, 123, 124, 125, 126, 127, 131, 133, 134, 136, 137, 138, 139, 140, 141, 143, 147, 149, 150,
153, 154, 155, 157, 158, 163, 165, 167, 169, 171, 172, 173, 174, 175, 176, 178, 179, 182, 185, 186, 187, 189, 190, 191,
193, 195, 196, 197, 198, 201, 202, 203, 204, 205, 206, 208, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 221, 222,
224, 225, 226, 227, 229, 230, 231, 232, 233, 234, 236, 238, 239, 240, 242, 243, 244, 245, 247, 252, 253, 255, 256, 257,
259, 260, 261, 262, 264, 266, 267, 268, 270, 273, 274, 275, 276, 277, 278, 280, 281, 282, 283, 284, 285, 286, 288, 289,
290, 291, 292, 293, 294, 296, 298, 299, 300, 301, 302, 304, 305, 306, 307, 308, 309, 310, 313, 314, 316, 317, 319, 320,
321, 324, 325, 326, 327, 328, 330, 331, 332, 333, 334, 335, 338, 339, 340, 341, 342, 345, 346, 347, 348, 349, 350, 351,
352, 354, 356, 358, 360, 361, 362, 363, 366, 367, 368, 369, 370, 371, 373, 377, 378, 379, 380, 381, 382, 383, 384, 385,
386, 387, 388, 389, 390, 391, 392, 395, 396, 398, 399, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 413, 415,
416, 417, 420, 421, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 438, 439, 440, 441, 442, 444,
445, 446, 448, 450, 451, 452, 453, 454, 455, 457, 458, 459, 461, 463, 464, 467, 469, 470, 473, 475, 476, 478, 485, 486,
487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 498, 499, 500, 501, 502, 503, 506, 507, 508, 509, 510, 511, 512, 514,
515, 516, 517, 518, 520, 523, 524, 525, 527, 528, 530, 531, 532, 534, 535, 536, 537, 538, 539, 540, 543, 544, 545 …., 998}
```

Biggest set/component length

```
[ ]  1  Ng=biggest_comp_length
     2  print(Ng)
     3
```

```
752
```

Now we will increase the value of k from 0 to 5 by 0.1 and try to find the value to length of biggest component formed by each k in graph G.

We have created a function biggest_component() to find the length of the biggest component. There is a while loop which will find **NG/N** ratio using the above function and store it along with k in a dictionary known as dict_q.

NG : Number of nodes in biggest component and N : Number of nodes in original graph.

dict_q{key,value} : key-->k and value-->NG/n

```
def biggest_component(lk):
    biggest_comp={}
    biggest_comp_length=0
    for a in lk:
        if(biggest_comp_length<len(a)):
            biggest_comp_length=len(a)
            biggest_comp=a
    return(biggest_comp_length)
k=0.0
n=1000
list_k=[]
R=[]
p_list=[]
dict_q={}
while(k<=5):
    p=k/(n-1)
    p_list.append(p)
    GB= nx.erdos_renyi_graph(n,p)
    lk=sorted(nx.connected_components(GB))
    NG=biggest_component(lk)
    ratio=NG/n
    R.append(ratio)
    list_k.append(k)
    dict_q[k]=ratio
    k=(10*k + 1)/10
print(R)
```

```
22      R.append(ratio)
23      list_k.append(k)
24      dict_q[k]=ratio
25      k=(10*k + 1)/10
26
```

All the ratios (NG/n) :

```
[ ]  1  print(R)
```

```
[0.002, 0.006, 0.01, 0.008, 0.018, 0.032, 0.024, 0.024, 0.078, 0.072, 0.2, 0.098, 0.418, 0.506, 0.49, 0.62, 0.652, 0.758,
0.778, 0.746, 0.808, 0.822, 0.808, 0.898, 0.898, 0.914, 0.912, 0.936, 0.918, 0.928, 0.95, 0.954, 0.956, 0.974, 0.976,
0.978, 0.966, 0.978, 0.976, 0.98, 0.974, 0.984, 0.978, 0.97, 0.988, 0.994, 0.988, 0.99, 0.99, 0.996, 0.992]
```

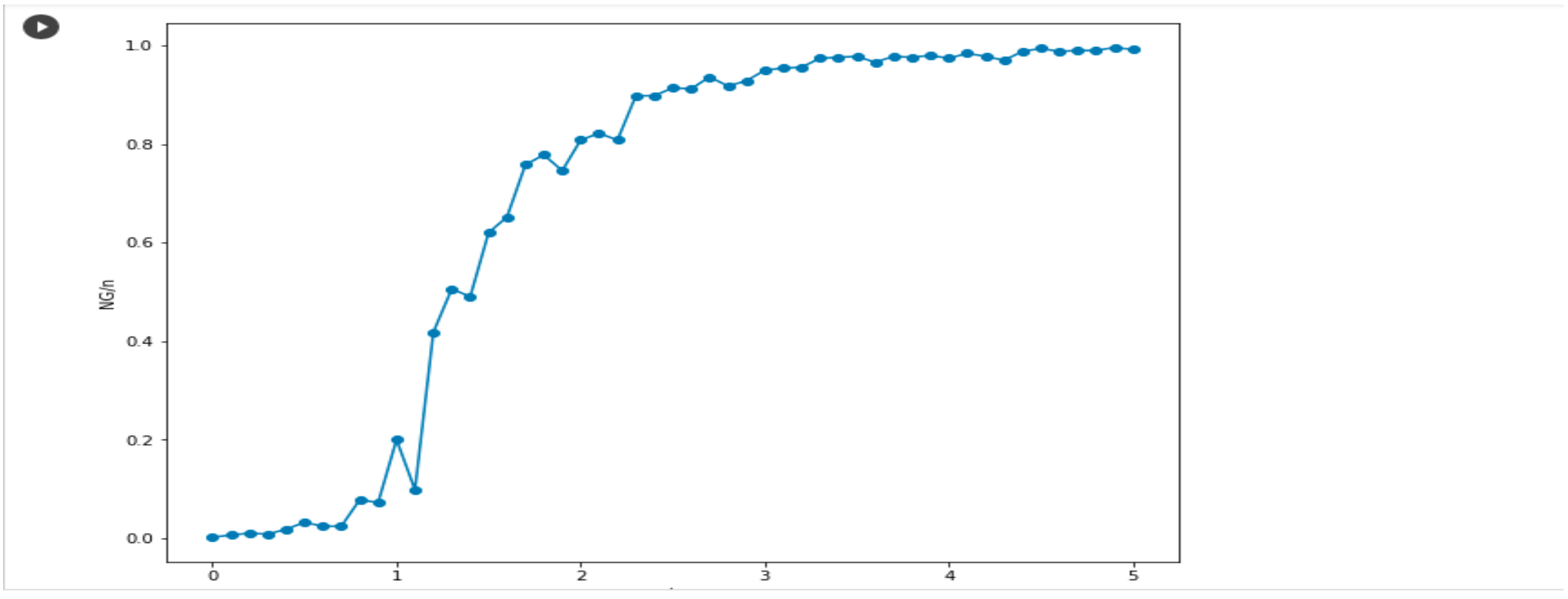
- All the ratios are values with K as their key in the dictionary

```
[ ]  1  print(dict_q)
     2
```

```
{0.0: 0.002, 0.1: 0.006, 0.2: 0.01, 0.3: 0.008, 0.4: 0.018, 0.5: 0.032, 0.6: 0.024, 0.7: 0.024, 0.8: 0.078, 0.9: 0.072,
1.0: 0.2, 1.1: 0.098, 1.2: 0.418, 1.3: 0.506, 1.4: 0.49, 1.5: 0.62, 1.6: 0.652, 1.7: 0.758, 1.8: 0.778, 1.9: 0.746, 2.0:
0.808, 2.1: 0.822, 2.2: 0.808, 2.3: 0.898, 2.4: 0.898, 2.5: 0.914, 2.6: 0.912, 2.7: 0.936, 2.8: 0.918, 2.9: 0.928, 3.0:
0.95, 3.1: 0.954, 3.2: 0.956, 3.3: 0.974, 3.4: 0.976, 3.5: 0.978, 3.6: 0.966, 3.7: 0.978, 3.8: 0.976, 3.9: 0.98, 4.0:
0.974, 4.1: 0.984, 4.2: 0.978, 4.3: 0.97, 4.4: 0.988, 4.5: 0.994, 4.6: 0.988, 4.7: 0.99, 4.8: 0.99, 4.9: 0.996, 5.0:
0.992}
```

Now we have plotted a graph of **ratio vs k**

```
1  lists = sorted(dict_q.items()) # sorted by key, return a list of tuples
2  x, y = zip(*lists) # unpack a list of pairs into two tuples
3  plt.figure(figsize=(10,8))
4  plt.plot(x, y)
5  plt.xlabel("k")
6  plt.ylabel("NG/n")
7  plt.scatter(x,y)
8  plt.show()
```



CONCLUSION

This project shows us that as we increase the value of k, the probability of joining two nodes increases and since the probability of joining every pair of nodes(random) increase, then we get a bigger or giant connected component.

Therefore increasing k increases the size of the biggest component or we can say increasing k decreases the number of components.

Using above graph we can say that

- Ratio of Ng/N almost 0 when avg. degree less than or equal to 1.
- Ratio of Ng/N increases exponentially when avg. degree greater than 1 and less than 4.5.
- Ratio of Ng/N almost constant but none zero when is greater than 4.5.

ROUND -2

Importing Libraries

```
:  import networkx as nx
   import csv
   import pandas as pd
   import numpy as np
   import matplotlib.pyplot as plt
   import random
```

Dataset:- Social circles: Twitter

Reading the dataset

```
In [2]: G =nx.read_edgelist(r"C:\Users\vania\Downloads\twitter_combined\twitter_combined _g.txt", create_using = nx.Graph(),
                          nodetype=int)
```

Finding the giant component in the network

```
n [3]: giant_comp = max(nx.connected_components(G),key=len)
       print("No of nodes in Graph = ",len(G.nodes()))
       print("No of nodes in Giant Component (NG) = ",len(giant_comp))
       print("Total no of components : ",nx.number_connected_components(G))
       print("NG/N = ",len(giant_comp)/len(G.nodes()))
```

```
No of nodes in Graph =  81306
No of nodes in Giant Component (NG) =  81306
Total no of components :  1
NG/N =  1.0
```

```
n [4]: print(nx.info(G))
```

```
Name:
Type: Graph
Number of nodes: 81306
Number of edges: 1342310
Average degree:  33.0187
```

Sampling the data

This is done because our dataset is very large and we are unable to apply algorithms on the original dataset as in above code we can see that the network has 81306 nodes and 1342310 edges.

```
import random
sample_edges = random.sample(G.edges(),100)
G3 = nx.Graph()
G3.add_edges_from(sample_edges)
print(nx.info(G3))
```

```
Name:
Type: Graph
Number of nodes: 198
Number of edges: 100
Average degree:   1.0101
```

Community Detection Algorithms:-

1. GIRVAN NEWMAN ALGORITHM

Step 1: Finding number of connected components.
Remove the edge with maximum edge betweenness centrality

In [6]:
```
def edge_to_remove(G):
    edge_bw_cent = nx.edge_betweenness_centrality(G).items()
    to_remove = max(edge_bw_cent,key=lambda x:x[1])
    return to_remove[0]
```

In [7]:
```
def girvan_newmman(G):
    sub = nx.number_connected_components(G)
    print("No of Subgraphs in the Graph Initially = ",sub)
    sub_c=sub
    print("REMOVED EDGES : ")
    while sub==sub_c:
        rem = edge_to_remove(G)
        G.remove_edge(*rem)
        sub_c = nx.number_connected_components(G)
        print("",rem,end=" ")
        print()
        return G
```

Step 2: Implementing Community Plot Function

```
def community_layout(g, partition):
    pos_communities = _position_communities(g, partition, scale=3.)
    pos_nodes =_position_nodes(g, partition, scale=1.)
    # combine positions
    pos = dict()
    for node in g.nodes():
        pos[node] = pos_communities[node] + pos_nodes[node]
    return pos
def _position_communities(g, partition, **kwargs):
# create a weighted graph, in which each node corresponds to a community,and each edge we
    between_community_edges = _find_between_community_edges(g, partition)
    communities = set(partition.values())
    hypergraph = nx.Graph()
    hypergraph.add_nodes_from(communities)
    for (ci, cj), edges in between_community_edges.items():
        hypergraph.add_edge(ci, cj, weight=len(edges))
        # find layout for communities
    pos_communities = nx.spring_layout(hypergraph, **kwargs)
        # set node positions to position of community
    pos = dict()
    for node, community in partition.items():
        pos[node] = pos_communities[community]
    return pos
```

```
def _find_between_community_edges(g, partition):
    edges = dict()
    for (ni, nj) in g.edges():
        ci = partition[ni]
        cj = partition[nj]
    if ci != cj:
        try:
            edges[(ci, cj)] += [(ni, nj)]
        except KeyError:
            edges[(ci, cj)] = [(ni, nj)]
    return edges


def _position_nodes(g, partition, **kwargs):
    communities = dict()
    for node, community in partition.items():
        try:
            communities[community] += [node]
        except KeyError:
             communities[community] = [node]
    pos = dict()
    for ci, nodes in communities.items():
        subgraph = g.subgraph(nodes)
        pos_subgraph = nx.spring_layout(subgraph, **kwargs)
        pos.update(pos_subgraph)
    return pos
```

```
def plot_community(G3,G4):
    gn_communities = list(nx.connected_components(G4))
    gn_dict_communities = {}
    for i, c in enumerate(gn_communities):
        for node in c:
            gn_dict_communities[node] = i + 1
    for node in G3:
        if node not in gn_dict_communities.keys():
            gn_dict_communities[node] = -1
        gn_pos = community_layout(G3, gn_dict_communities)
    from matplotlib import cm
    gn_colors = []
    for node in G3.nodes:
        gn_colors.append(cm.Set1(gn_dict_communities[node]))
        plt.figure(figsize=(5,5))
    nx.draw_networkx_nodes(G3, gn_pos, node_color=gn_colors, node_size=500)
    nx.draw_networkx_edges(G3, gn_pos,alpha=1)
    plt.axis('off')
    plt.show()
```

```
print("No.ofConnectedComponents=",nx.number_connected_components(G3))
```

```
No.ofConnectedComponents= 98
```

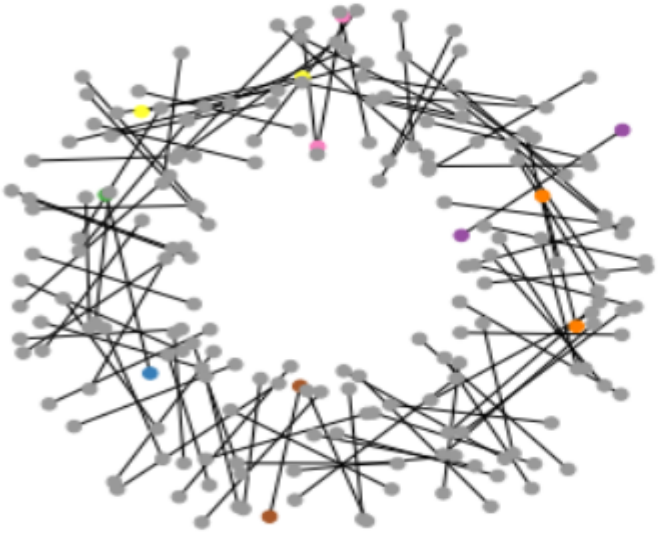
Step 3: Find Communities

We have iterated the algorithm 5 times.

Iteration 1

```
###1st_iteration
G4 = girvan_newmman(nx.Graph(G3))
print("Size of Components in the Graph :",[len(c) for c in
sorted(nx.connected_components(G))])
plot_community(G3,G4)
G3 = nx.Graph(G4)
```

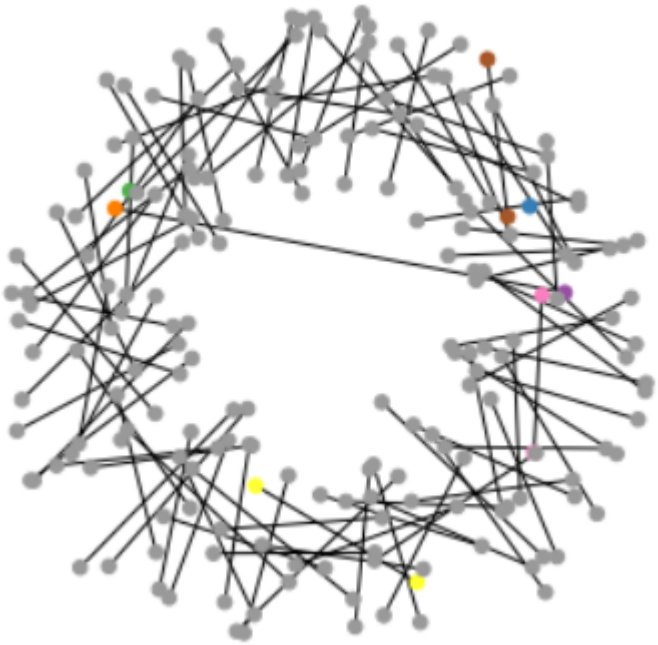
```
No of Subgraphs in the Graph Initially =  100
REMOVED EDGES :
 (320527944, 169157062)
Size of Components in the Graph : [81306]
```



Iteration 2

```
####2nd_iteration

G4 = girvan_newmman(nx.Graph(G3))
print("Size of Components in the Graph :",[len(c) for c in
sorted(nx.connected_components(G ))])
plot_community(G3,G4)
G3 = nx.Graph(G4)
```

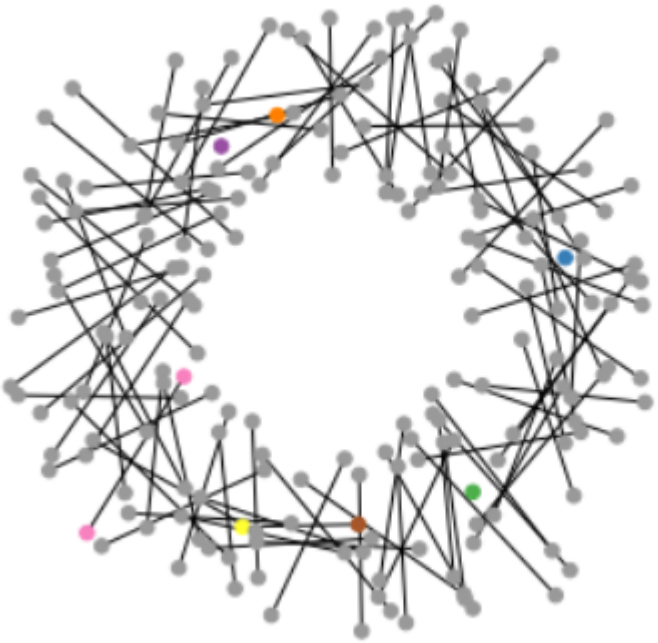
```
No of Subgraphs in the Graph Initially =  101
REMOVED EDGES :
 (19058681, 19624359)
Size of Components in the Graph : [81306]
```



Iteration 3

```
###3rd_iteration

G4 = girvan_newmman(nx.Graph(G3))
print("Size of Components in the Graph :",[len(c) for c in
sorted(nx.connected_components(G))])
plot_community(G3,G4)
G3 = nx.Graph(G4)
```

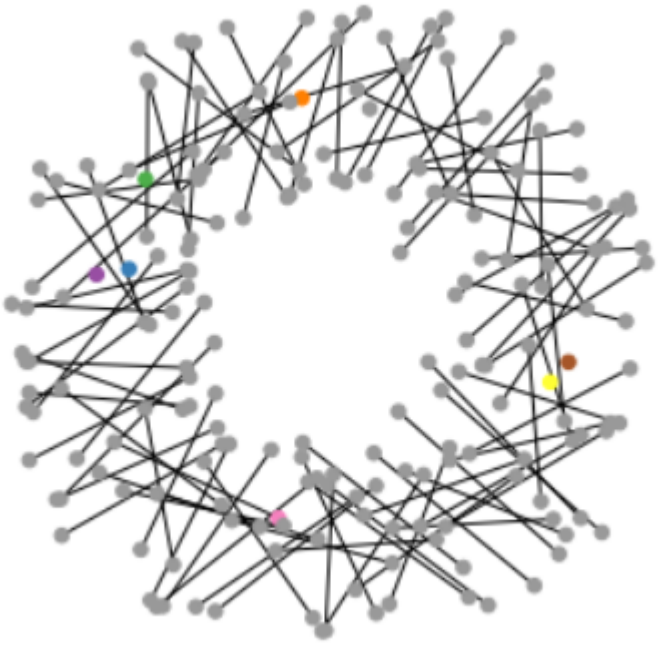
```
No of Subgraphs in the Graph Initially =  102
REMOVED EDGES :
 (19083676, 96569103)
Size of Components in the Graph : [81306]
```



Iteration 4

```
###4th_iteration
G4 = girvan_newmman(nx.Graph(G3))
print("Size of Components in the Graph :",[len(c) for c in
sorted(nx.connected_components(G))])
plot_community(G3,G4)
```

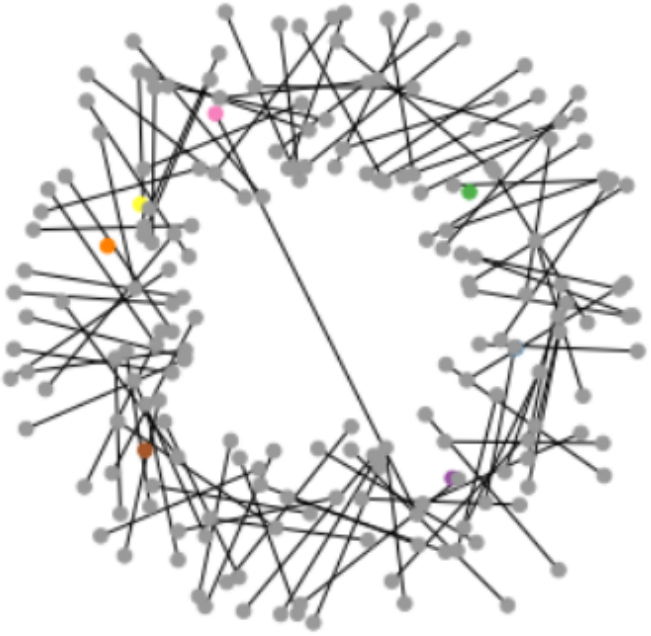
```
No of Subgraphs in the Graph Initially =  103
REMOVED EDGES :
 (57321825, 34041776)
Size of Components in the Graph : [81306]
```



Iteration 5

```
####5th_iteration

G4 = girvan_newmman(nx.Graph(G3))
print("Size of Components in the Graph :",[len(c) for c in
sorted(nx.connected_components(G ))])
plot_community(G3,G4)
G3 = nx.Graph(G4)
```

```
No of Subgraphs in the Graph Initially =  103
REMOVED EDGES :
 (57321825, 34041776)
Size of Components in the Graph : [81306]
```



RAVASZ ALGORITHM

In this method, we start with a **distance matrix** D, which is a matrix that provides the distance between two nodes i and j in network G. Using D we then perform hierarchical clustering in a recursive manner, group *similar* nodes at each step. We have used average distance for similarity grouping.

```
]:
def create_hc(G, t):
    ## Set-up the distance matrix D
    labels=G.nodes()    # keep node labels
    path_length=[n for n in nx.all_pairs_shortest_path_length(G)]
    distances=np.zeros((len(G),len(G)))
    g=list(G.nodes())
    for u,p in path_length:
        for v,d in p.items():
            distances[g.index(u)][g.index(v)] = d
            distances[g.index(v)][g.index(u)] = d
            if u==v:
                distances[g.index(u)][g.index(u)]=0

    # Create hierarchical cluster (HC)
    # There are various other routines for agglomerative clustering,
    # but here we create the HCs using the complete/max/farthest point linkage
    Y = distance.squareform(distances) ## the upper triangular of the distance matrix
    Z = hierarchy.average(Y)

    # This partition selection (t) is arbitrary, for illustrive purposes
    membership=list(hierarchy.fcluster(Z,t=t))

    # Create collection of lists for blockmodel
    partition = defaultdict(list)
    for n,p in zip(list(range(len(G))),membership):
        partition[p].append(labels)

    return Z, membership, partition
```

```
Z, membership, partition = create_hc(G3, t=1.15)
partition.items()
```

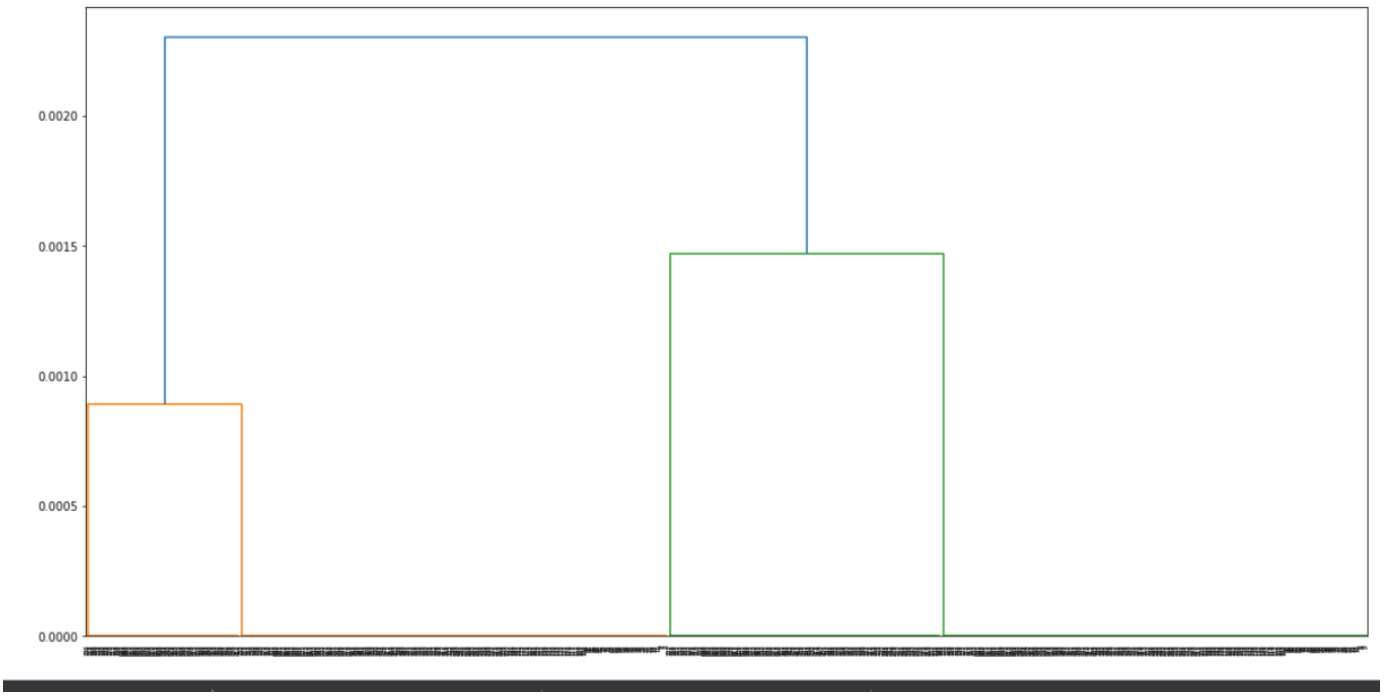
```
49]: dict_items([(4, [NodeView((18755518, 15394673, 30562359, 18151654, 10202, 15947897, 140233086, 121033334, 15647676, 87756
72, 45673, 74523, 263838766, 227339845, 22747624, 18731347, 127935456, 113124784, 22127274, 167105896, 50864457, 1433808
2, 15853073, 26095348, 147596311, 113238327, 54384394, 75142637, 221198749, 107758229, 12555, 4981271, 19793454, 37586286
0, 372393022, 325084625, 20273398, 29118453, 14888584, 50000478, 227315505, 232026363, 166374073, 19658506, 22027186, 127
11832, 8659362, 175122604, 179524189, 30680870, 46423291, 65673, 17023344, 29205301, 251797393, 38851803, 2841151, 153299
10, 44409004, 33443003, 172366206, 42338280, 20027082, 39013503, 19553306, 111517935, 54365543, 69175625, 18980714, 13454
9649, 15031479, 23374570, 269993140, 151739682, 17020962, 18774246, 238260874, 116905274, 128771911, 299776583, 46150233,
62545255, 14119053, 9180762, 33998183, 14217740, 12830, 771712, 114881835, 105150583, 413582060, 19824350, 6431392, 18215
158, 158506294, 100338161, 35097545, 54620720, 362286999, 25199993, 17272055, 1422311, 7377812, 7872262, 14461282, 946826
74, 25993, 14223705, 20797196, 16254920, 232294292, 28694307, 75974281, 23991290, 122439303, 199072435, 24542441, 1400784
32, 52216611, 21527405, 113657678, 88289104, 124524869, 21195122, 40492559, 16633144, 50374439, 130249139, 20942815, 2143
6558, 74294087, 165675053, 12916, 15875403, 18142375, 21746442, 20436059, 139913805, 16279105, 280329780, 188499505, 2818
44209, 32350369, 114775920, 21791940, 148148588, 17093617, 14187323, 3108351, 18994661, 7710472, 128194782, 204140367, 14
222518, 19262793, 115269295, 22169313, 263352652, 58895957, 107512718, 15929867, 100286635, 244404296, 69799377, 2817604
1, 51543792, 21226980, 14587578, 58019852, 382897912, 168735527, 166718532, 360268721, 464435768, 7098022, 111515806, 141
90551, 72515134, 34771668, 53372998, 82963512, 19432206, 160267464, 67474868, 57828817, 58511104, 19382500, 29411301, 477
7911, 15186910, 73731253, 15077795, 76966145, 26527783, 20785993, 179887262, 43996908, 9856232, 362139322, 277675909, 334
414655, 259379883, 267710893, 19078594, 14133447, 82030021, 29034979, 7152572, 18212352, 506688854, 298756207, 39506306,
325203020, 7081402, 13348, 14780915, 53717630, 24919888, 11211372, 257668899, 272064390, 19161407, 17831432, 19665244, 15
8844523, 234526497, 21225078, 325954622, 18848018, 18220175, 17053635, 20207789, 24586098, 22462567, 15871190, 18859658,
```

```
partition = {}
i = 0
for i in range(len(membership)):
    partition[i]=membership[i]
```

```
3]:  plt.figure(figsize=(10,10))
     plt.axis('off')
     pos = nx.fruchterman_reingold_layout(G3);
     nx.draw_networkx_edges(G3, pos, edge_color='black',alpha=0.5)
     plt.show()
```

We have plotted Dendrogram so as to visualise the network.

```
plt.figure(figsize=(20,10))
hierarchy.dendrogram(Z)
plt.show()
```



DATASET 2:- : Epinions social network

Reading the dataset

```
G =nx.read_edgelist(r"C:\Users\vania\Downloads\soc-Epinions1 (1).txt\soc-Epinions1 (1).txt", create_using = nx.Graph(),
                    nodetype=int)
```

Finding the giant component in the network

```
giant_comp = max(nx.connected_components(G),key=len)
print("No of nodes in Graph = ",len(G.nodes()))
print("No of nodes in Giant Component (NG) = ",len(giant_comp))
print("Total no of components : ",nx.number_connected_components(G))
print("NG/N = ",len(giant_comp)/len(G.nodes()))
```

```
No of nodes in Graph =  75879
No of nodes in Giant Component (NG) =  75877
Total no of components :  2
NG/N =  0.9999736422462078
```

```
print(nx.info(G))
```

```
Name:
Type: Graph
Number of nodes: 75879
Number of edges: 405740
Average degree:  10.6944
```

Sampling the data

This is done because our dataset is very large and we are unable to apply algorithms on the original dataset as in above code we can see that the network has 75879 nodes and 405740 edges.

```
import random
sample_edges = random.sample(G.edges(),100)
G3 = nx.Graph()
G3.add_edges_from(sample_edges)
print(nx.info(G3))
```

```
Name:
Type: Graph
Number of nodes: 200
Number of edges: 100
Average degree:   1.0000
```

Community Detection Algorithms:-

GIRVAN NEWMAN ALGORITHM:-

Step 1: Finding number of connected components.
Remove the edge with maximum edge betweenness centrality

In [6]:
```
def edge_to_remove(G):
    edge_bw_cent = nx.edge_betweenness_centrality(G).items()
    to_remove = max(edge_bw_cent,key=lambda x:x[1])
    return to_remove[0]
```

In [7]:
```
def girvan_newmman(G):
    sub = nx.number_connected_components(G)
    print("No of Subgraphs in the Graph Initially = ",sub)
    sub_c=sub
    print("REMOVED EDGES : ")
    while sub==sub_c:
        rem = edge_to_remove(G)
        G.remove_edge(*rem)
        sub_c = nx.number_connected_components(G)
        print("",rem,end=" ")
        print()
        return G
```

Step 2: Implementing Community Plot Function

```
def community_layout(g, partition):
    pos_communities = _position_communities(g, partition, scale=3.)
    pos_nodes =_position_nodes(g, partition, scale=1.)
    # combine positions
    pos = dict()
    for node in g.nodes():
        pos[node] = pos_communities[node] + pos_nodes[node]
    return pos
def _position_communities(g, partition, **kwargs):
# create a weighted graph, in which each node corresponds to a community,and each edge we
    between_community_edges = _find_between_community_edges(g, partition)
    communities = set(partition.values())
    hypergraph = nx.Graph()
    hypergraph.add_nodes_from(communities)
    for (ci, cj), edges in between_community_edges.items():
        hypergraph.add_edge(ci, cj, weight=len(edges))
        # find layout for communities
    pos_communities = nx.spring_layout(hypergraph, **kwargs)
        # set node positions to position of community
    pos = dict()
    for node, community in partition.items():
        pos[node] = pos_communities[community]
    return pos
```

```
def _find_between_community_edges(g, partition):
    edges = dict()
    for (ni, nj) in g.edges():
        ci = partition[ni]
        cj = partition[nj]
    if ci != cj:
        try:
            edges[(ci, cj)] += [(ni, nj)]
        except KeyError:
            edges[(ci, cj)] = [(ni, nj)]
    return edges


def _position_nodes(g, partition, **kwargs):
    communities = dict()
    for node, community in partition.items():
        try:
            communities[community] += [node]
        except KeyError:
             communities[community] = [node]
    pos = dict()
    for ci, nodes in communities.items():
        subgraph = g.subgraph(nodes)
        pos_subgraph = nx.spring_layout(subgraph, **kwargs)
        pos.update(pos_subgraph)
    return pos
```

```
def plot_community(G3,G4):
    gn_communities = list(nx.connected_components(G4))
    gn_dict_communities = {}
    for i, c in enumerate(gn_communities):
        for node in c:
            gn_dict_communities[node] = i + 1
    for node in G3:
        if node not in gn_dict_communities.keys():
            gn_dict_communities[node] = -1
        gn_pos = community_layout(G3, gn_dict_communities)
    from matplotlib import cm
    gn_colors = []
    for node in G3.nodes:
        gn_colors.append(cm.Set1(gn_dict_communities[node]))
        plt.figure(figsize=(5,5))
    nx.draw_networkx_nodes(G3, gn_pos, node_color=gn_colors, node_size=500)
    nx.draw_networkx_edges(G3, gn_pos,alpha=1)
    plt.axis('off')
    plt.show()
```

```
print("No.ofConnectedComponents=",nx.number_connected_components(G3))
```

```
No.ofConnectedComponents= 98
```

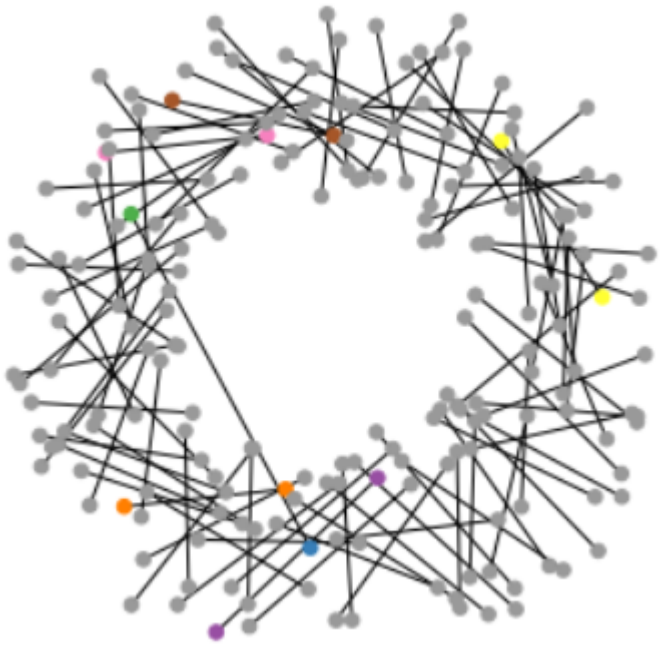
Step 3: Find Communities

We have iterated the algorithm 5 times.

Iteration 1

```
###1st_iteration
G4 = girvan_newmman(nx.Graph(G3))
print("Size of Components in the Graph :",[len(c) for c in
sorted(nx.connected_components(G))])
plot_community(G3,G4)
G3 = nx.Graph(G4)
```

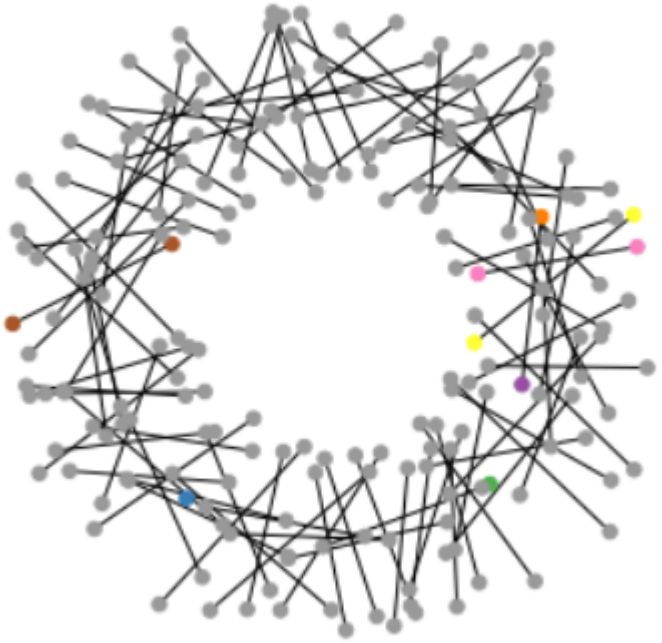
```
No of Subgraphs in the Graph Initially =  100
REMOVED EDGES :
 (4118, 33982)
Size of Components in the Graph : [75877, 2]
```



Iteration 2

```
####2nd_iteration

G4 = girvan_newmman(nx.Graph(G3))
print("Size of Components in the Graph :",[len(c) for c in
sorted(nx.connected_components(G ))])
plot_community(G3,G4)
G3 = nx.Graph(G4)
```

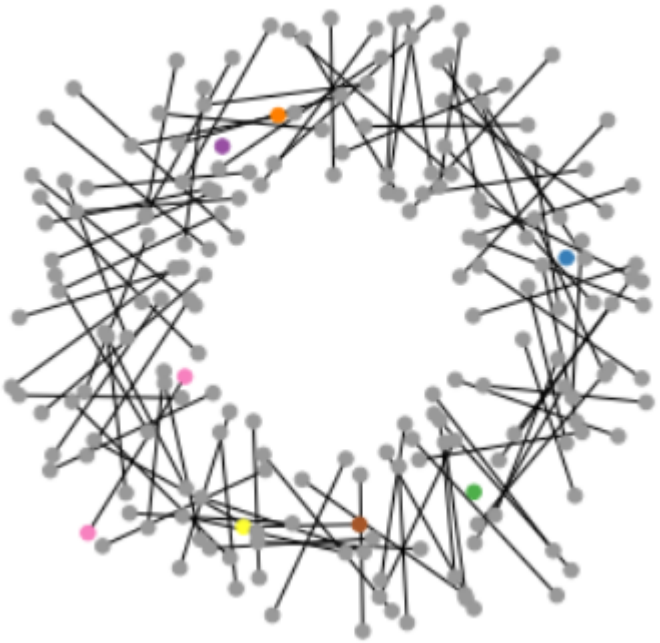
```
No of Subgraphs in the Graph Initially =  101
REMOVED EDGES :
 (634, 73748)
Size of Components in the Graph : [75877, 2]
```



Iteration 3

```
###3rd_iteration

G4 = girvan_newmman(nx.Graph(G3))
print("Size of Components in the Graph :",[len(c) for c in
sorted(nx.connected_components(G))])
plot_community(G3,G4)
G3 = nx.Graph(G4)
```

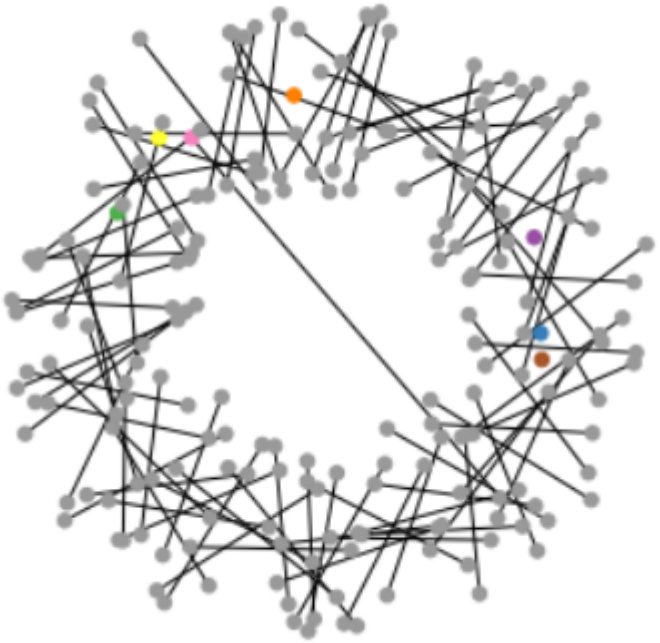
```
No of Subgraphs in the Graph Initially =  102
REMOVED EDGES :
 (19083676, 96569103)
Size of Components in the Graph : [81306]
```



Iteration 4

```
###4th_iteration
G4 = girvan_newmman(nx.Graph(G3))
print("Size of Components in the Graph :",[len(c) for c in
sorted(nx.connected_components(G))])
plot_community(G3,G4)
```

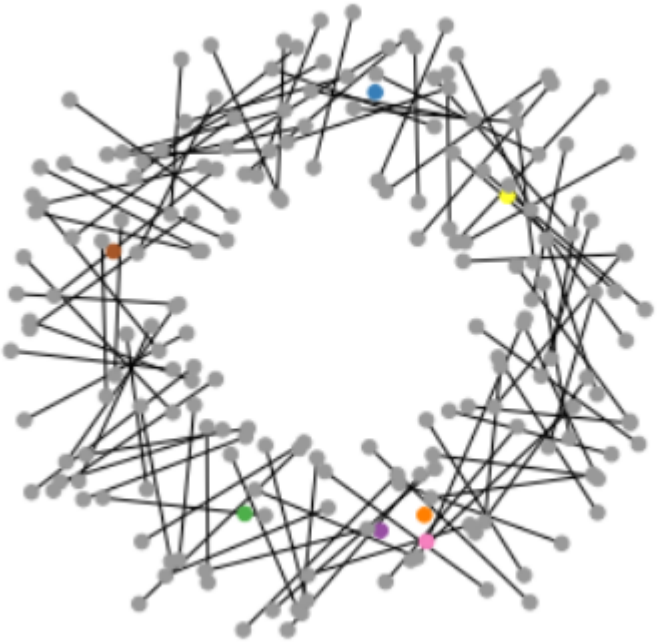
```
No of Subgraphs in the Graph Initially =  103
REMOVED EDGES :
 (1516, 17900)
Size of Components in the Graph : [75877, 2]
```



Iteration 5

```
####5th_iteration

G4 = girvan_newmman(nx.Graph(G3))
print("Size of Components in the Graph :",[len(c) for c in
sorted(nx.connected_components(G ))])
plot_community(G3,G4)
G3 = nx.Graph(G4)
```

```
No of Subgraphs in the Graph Initially =  103
REMOVED EDGES :
 (1516, 17900)
Size of Components in the Graph : [75877, 2]
```



RAVASZ ALGORITHM

In this method, we start with a **distance matrix** D, which is a matrix that provides the distance between two nodes i and j in network G. Using D we then perform hierarchical clustering in a recursive manner, group *similar* nodes at each step. We have used average distance for similarity grouping.

```
]:
def create_hc(G, t):
    ## Set-up the distance matrix D
    labels=G.nodes()    # keep node labels
    path_length=[n for n in nx.all_pairs_shortest_path_length(G)]
    distances=np.zeros((len(G),len(G)))
    g=list(G.nodes())
    for u,p in path_length:
        for v,d in p.items():
            distances[g.index(u)][g.index(v)] = d
            distances[g.index(v)][g.index(u)] = d
            if u==v:
                distances[g.index(u)][g.index(u)]=0

    # Create hierarchical cluster (HC)
    # There are various other routines for agglomerative clustering,
    # but here we create the HCs using the complete/max/farthest point linkage
    Y = distance.squareform(distances) ## the upper triangular of the distance matrix
    Z = hierarchy.average(Y)

    # This partition selection (t) is arbitrary, for illustrive purposes
    membership=list(hierarchy.fcluster(Z,t=t))

    # Create collection of lists for blockmodel
    partition = defaultdict(list)
    for n,p in zip(list(range(len(G))),membership):
        partition[p].append(labels)

    return Z, membership, partition
```

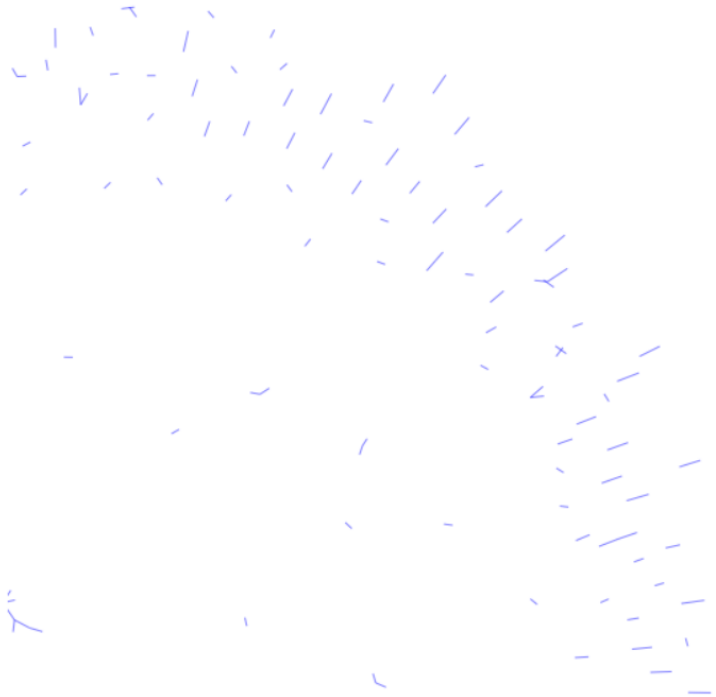
```
Z, membership, partition = ravasz_algorithm(G3, t=1.15)
partition.items()
```

```
: dict_items([(4, [NodeView((36372, 27995, 715, 15252, 1227, 17945, 436, 10565, 232, 1382, 58805, 69597, 49, 492, 16922, 54
841, 14580, 3850, 917, 2045, 770, 31410, 5216, 3937, 597, 2725, 71399, 71615, 2654, 12907, 318, 2289, 67275, 70763, 237,
2726, 1665, 1857, 790, 18349, 32762, 47696, 1776, 4422, 2246, 6810, 3852, 21469, 3514, 8745, 2105, 5373, 77, 602, 9479, 2
0168, 97, 1915, 5519, 15717, 76, 4904, 2634, 27936, 2687, 4061, 83, 2554, 5972, 36944, 15225, 6372, 1512, 2706, 1392, 350
3, 2514, 198, 6966, 38097, 2469, 2420, 154, 7429, 1106, 3292, 4208, 34373, 4319, 8140, 35011, 64485, 7063, 32690, 10499,
23650, 223, 3425, 644, 12626, 28909, 62244, 2478, 2984, 19468, 19488, 92, 4782, 442, 2923, 5667, 13613, 2953, 5393, 3830,
3117, 19053, 52381, 2672, 4957, 2649, 5227, 883, 566, 1178, 3769, 32372, 5250, 2212, 67272, 70760, 5952, 6394, 693, 4197,
8787, 19740, 138, 4577, 3073, 1680, 143, 1865, 3523, 31674, 540, 1497, 9072, 12635, 2318, 2852, 45325, 18488, 44, 23826,
40287, 79, 839, 18361, 55777, 63, 4393, 1037, 2897, 5113, 1531, 2302, 8598, 11968, 2197, 526, 11235, 2324, 837, 2160, 138
34, 1301, 18516, 4413, 12913, 768, 14444, 1715, 1996, 19360, 56651, 4984, 37145, 283, 8494, 3327, 7035, 737, 846, 2025, 1
3700, 1191, 68379, 395, 4920, 18095, 19530, 23298, 27531, 9764, 34957, 3924, 11998, 3524, 31684, 2786, 3564, 35, 952, 162
81, 19462, 8374, 8735, 10447, 40377, 17666, 12208, 3986, 5759, 9057, 17431, 6864, 4077, 2326, 912, 2497, 27414, 1649, 955
0, 46, 3102, 557, 6151, 14548, 29470, 6102, 1481, 5134, 22229, 40, 1640, 8156, 27986, 2201, 3863, 62588, 29970, 2225, 262
81, 7104, 5982, 18, 380, 1908, 12597, 35516, 64674, 1718, 1093, 13762, 52149, 2146, 16453, 32590, 2162, 6495, 1166, 1572,
16869, 7260, 14778, 50210, 2227, 8158, 26033, 9459, 5758, 15945, 1714, 212, 4614, 1641, 4565, 2106, 24808, 60552, 6786, 5
505, 16643, 26257, 34550, 9589, 5793, 1494, 27982, 5776, 34719, 6978, 15825, 1899, 17784, 1885, 24495, 59, 1510, 1935, 73
46, 335, 156, 5829, 4760, 1618, 1669, 3109, 5969, 3987, 14885, 2491, 1617, 30465, 16325, 22753, 1906, 997, 4276, 1620, 40
54, 141, 12458, 9266, 135, 348, 23552, 54693, 1450, 2192, 118, 1501, 4573, 1526, 699, 12675, 1320, 1102, 982, 2040, 847,
1765, 26153, 26154, 10698, 13507, 16316, 2194, 7041, 27573, 61706, 1818, 23751, 21067, 47053, 3884, 8718, 31062, 51871, 1
69, 2215, 398, 8541, 72, 102, 19220, 44165, 1731, 22950, 1170, 1342, 681, 145, 33069, 62208, 6478, 1503, 49266, 12162, 52
```

```
partition = {}
i = 0
for i in range(len(membership)):
    partition[i]=membership[i]
```

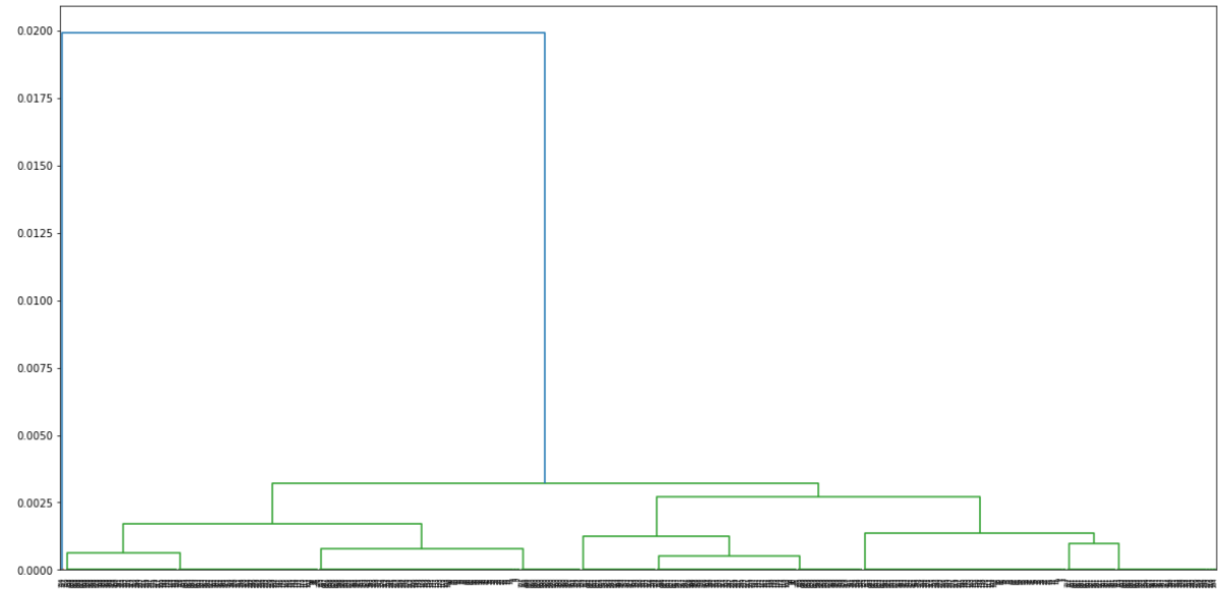
```
plt.figure(figsize=(10,10))
```

```
plt.figure(figsize=(10,10))
plt.axis('off')
pos = nx.fruchterman_reingold_layout(G3);
nx.draw_networkx_edges(G3, pos, edge_color='blue',alpha=0.5)
plt.show()
```



We have plotted Dendrogram so as to visualise the network.

```
plt.figure(figsize=(20,10))
hierarchy.dendrogram(Z)
plt.show()
```



PROBLEM 2

2) Create a scale-free network using appropriate Python package (find out!). Apply ICM (Independent Cascade Model) to find the maximum number steps required to get to the maximum number of nodes. This you may repeat 5 times by starting from different nodes and see how many steps are required for the above. Activation probabilities for the pair of nodes which is needed for ICM can be assigned randomly. When you are assigning it randomly note this point: from a node say v if there are three edges to different vertices w,x, and y. Then it should be p(v,w)+p(v,x)+p(v,y)=1.

1. SCALE FREE NETWORK

Scale free network: A scale-free network is a network whose degree distribution follows a power law, at least asymptotically.

```
[1]  1  import networkx as nx
     2  import numpy as np
     3  import random as rd
     4  import matplotlib.pyplot as plt
```

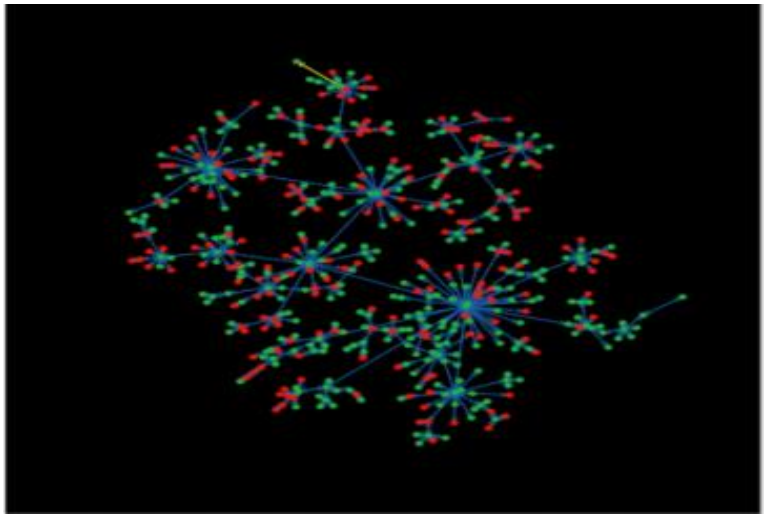
```
     1  COLOR = '#40a6d1'
```

Define helper functions

```
[ ]
```

```python
# Plotting

def k_distrib(graph, scale='lin', colour='#40a6d1', alpha=.8, fit_line=False, expct_lo=1, expct_hi=10, expct_const=1):

    plt.close()
    num_nodes = graph.number_of_nodes()
    max_degree = 0

    # Calculate the maximum degree to know the range of x-axis
    for n in graph.nodes():
        if graph.degree(n) > max_degree:
            max_degree = graph.degree(n)

    # X-axis and y-axis values
    x = []
    y_tmp = []

    # Loop over all degrees until the maximum to compute the portion of nodes for that degree
    for i in range(max_degree + 1):
        x.append(i)
        y_tmp.append(0)
        for n in graph.nodes():
            if graph.degree(n) == i:
                y_tmp[i] += 1
        y = [i / num_nodes for i in y_tmp]

    # Check for the lin / log parameter and set axes scale
    if scale == 'log':
        plt.xscale('log')
        plt.yscale('log')
        plt.title('Degree distribution (log-log scale)')
        plt.ylabel('log(P(k))')
        plt.xlabel('log(k)')
        plt.plot(x, y, linewidth = 0, marker = 'o', markersize = 8, color = colour, alpha = alpha)

        if fit_line:
            # Add theoretical distribution line k^-3
            # Note that you need to parametrize it manually
            w = [a for a in range(expct_lo,expct_hi)]
            z = []
            for i in w:
                x = (i**-3) * expct_const # set line's length and fit intercept
                z.append(x)

            plt.plot(w, z, 'k-', color='#7f7f7f')
```

```
47      else:
48          plt.plot(x, y, linewidth = 0, marker = 'o', markersize = 8, color = colour, alpha = alpha)
49          plt.title('Degree distribution (linear scale)')
50          plt.ylabel('P(k)')
51          plt.xlabel('k')
52
53      plt.show()
```



```python
# BA algo functions

def rand_prob_node():
    nodes_probs = []
    for node in G.nodes():
        node_degr = G.degree(node)
        #print(node_degr)
        node_proba = node_degr / (2 * len(G.edges()))
        #print("Node proba is: {}".format(node_proba))
        nodes_probs.append(node_proba)
        #print("Nodes probablities: {}".format(nodes_probs))
    random_proba_node = np.random.choice(G.nodes(),p=nodes_probs)
    #print("Randomly selected node is: {}".format(random_proba_node))
    return random_proba_node

def add_edge():
        if len(G.edges()) == 0:
            random_proba_node = 0
        else:
            random_proba_node = rand_prob_node()
        new_edge = (random_proba_node, new_node)
        if new_edge in G.edges():
            print("ככה לא בונים חומה!")
            add_edge()
        else:
            print("מול טוב!")
            G.add_edge(new_node, random_proba_node)
            print("Edge added: {} {}".format(new_node + 1, random_proba_node))
```

BA Algorithm

BA Algorithm

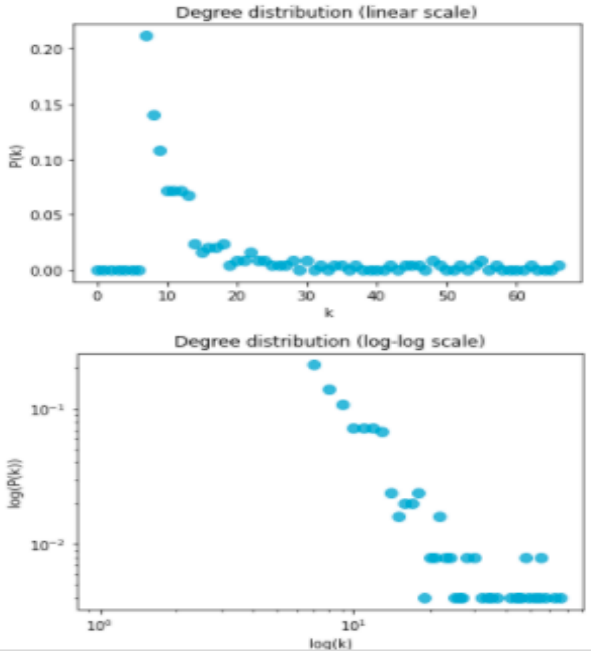
```python
# Get parameters
init_nodes = int(input("Please type in the initial number of nodes (m_0): "))
final_nodes = int(input("\nPlease type in the final number of nodes: "))
m_parameter = int(input("\nPlease type in the value of m parameter (m<=m_0): "))

print("\n")
print("Creating initial graph...")

G = nx.complete_graph(init_nodes)

print("Graph created. Number of nodes: {}".format(len(G.nodes())))
print("Adding nodes...")

count = 0
new_node = init_nodes

for f in range(final_nodes - init_nodes):
    print("----------> Step {} <----------".format(count))
    G.add_node(init_nodes + count)
    print("Node added: {}".format(init_nodes + count + 1))
    count += 1
    for e in range(0, m_parameter):
        add_edge()
    new_node += 1


print("\nFinal number of nodes ({}) reached".format(len(G.nodes())))
```

```
Please type in the initial number of nodes (m_0): 12

Please type in the final number of nodes: 250

Please type in the value of m parameter (m<=m_0): 7


Creating initial graph...
Graph created. Number of nodes: 12
Adding nodes...
----------> Step 0 <----------
Node added: 13
מזל טוב!
Edge added: 13 7
מזל טוב!
Edge added: 13 5
מזל טוב!
Edge added: 13 0
ככה לא בונים חומה!
מזל טוב!
Edge added: 13 11
מזל טוב!
Edge added: 13 2
ככה לא בונים חומה!
מזל טוב!
Edge added: 13 12
מזל טוב!
Edge added: 13 10
----------> Step 1 <----------
Node added: 14
מזל טוב!
Edge added: 14 1
מזל טוב!
Edge added: 14 0
```

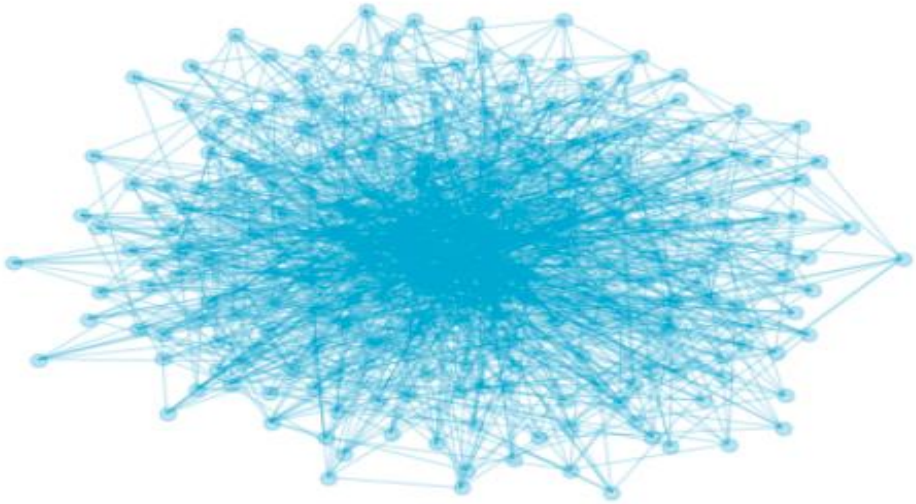
```
Edge added: 14 5
מזל טוב!
Edge added: 14 6
מזל טוב!
Edge added: 14 2
מזל טוב!
Edge added: 14 8
ככה לא בונים חומה!
מזל טוב!
Edge added: 14 7
----------> Step 2 <----------
Node added: 15
מזל טוב!
Edge added: 15 11
ככה לא בונים חומה!
ככה לא בונים חומה!
מזל טוב!
Edge added: 15 4
מזל טוב!
Edge added: 15 13
מזל טוב!
Edge added: 15 1
מזל טוב!
Edge added: 15 10
מזל טוב!
Edge added: 15 0
ככה לא בונים חומה!
ככה לא בונים חומה!
מזל טוב!
Edge added: 15 6
----------> Step 3 <----------
Node added: 16
מזל טוב!
Edge added: 16 10
```

Plot

```
[7]  1  k_distrib(G), k_distrib(G, scale = 'log', fit_line = False)
```



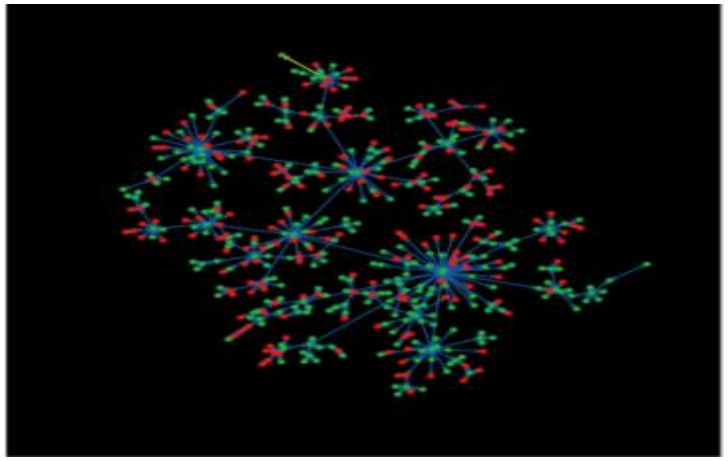
```
[8]  1  # Plot the network
     2  nx.draw(G, alpha = .3, edge_color = COLOR, node_color = COLOR, node_size=50)
```



2. Setting and Drawing up ICM (Information Cascade Model)

```
[ ]  1  import networkx as nx
     2  import csv
     3  import pandas as pd
     4  import numpy as np
     5  import matplotlib.pyplot as plt
```

```
     1  G= nx.barabasi_albert_graph(500,100)
     2  nx.draw(G, with_labels=False)
     3  plt.show()
     4
```



Return the active nodes of each diffusion step by independent cascade model
Parameters are as follows:-
G: A networxx graph

Seed: list of nodes
(It is the seed node for diffusion)

Steps: The number of steps to diffuse
(integer)

```
1  import copy
2  import networkx as nx
3  import random
```

```
[ ]  1  def independent_cascade(DG, seeds):
     2      A = copy.deepcopy(seeds)  # prevent side effect
     3      return _diffuse_all(DG, A)
```

```
[ ]
def _diffuse_all(G, A):
    tried_edges = set()
    layer_i_nodes = [ ]
    layer_i_nodes.append([i for i in A])  # prevent side effect
    while True:
        len_old = len(A)
        (A, activated_nodes_of_this_round, cur_tried_edges) = \
        _diffuse_one_round(G, A, tried_edges)
        layer_i_nodes.append(activated_nodes_of_this_round)
        tried_edges = tried_edges.union(cur_tried_edges)
        if len(A) == len_old:
            break
    return layer_i_nodes
```

```
[ ]
def _diffuse_one_round(G, A, tried_edges):
    activated_nodes_of_this_round = set()
    cur_tried_edges = set()
    for s in A:
        for nb in G.successors(s):
            if nb in A or (s, nb) in tried_edges or (s, nb) in cur_tried_edges:
                continue
            if _prop_success(G, s, nb):
                activated_nodes_of_this_round.add(nb)
            cur_tried_edges.add((s, nb))
    activated_nodes_of_this_round = list(activated_nodes_of_this_round)
    A.extend(activated_nodes_of_this_round)
    return A, activated_nodes_of_this_round, cur_tried_edges
```

```
[ ]
def _prop_success(G, src, dest):
    return random.random() <= G[src][dest]['act_prob']

```

```
[ ]
import datetime
import pickle
import csv
```

```
[ ]
r = 5#number of cascades to be started from each node
 #the input graph file with edge weights
cascadefilename = 'cascades_email'
```

```
[ ]  1  print("begin_app", datetime.datetime.now().time())
     2  G1=nx.read_edgelist(r"email_edgeweight.txt", data=(('act_prob',float),), encoding='utf-8', nodetype=int) #ge
     3  n = nx.number_of_nodes(G1)
```

```
begin_app 11:01:53.319152
```

```
[ ]  1  if not G1.is_directed():
     2      DG = G1.to_directed()
     3  else:
     4      DG = copy.deepcopy(G1)
     5  G1=DG
```

```
     1  cascades = [] #an empty list for storing cascades
     2
     3  print("begin_IC", datetime.datetime.now().time())
```

```
begin_IC 11:01:53.719536
```

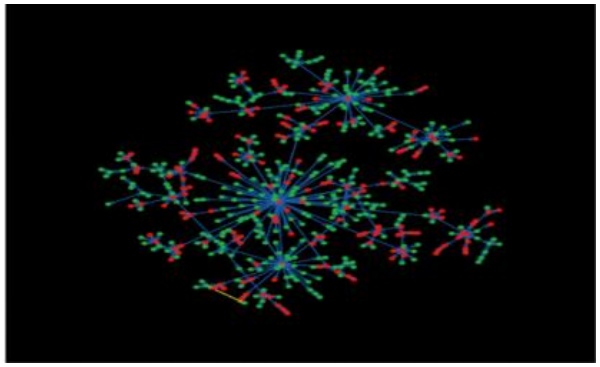
```
     1  for i in range(n): #generate r cascades for each node and append to the cascade list
     2      for j in range (r):
     3          cascades.append(independent_cascade(G1, [i]))
     4
     5  print("end_IC", datetime.datetime.now().time())
     6
```

```
end_IC 11:34:55.877456
```

```
[ ]  1  with open(cascadefilename, 'wb') as fp: #save the cascades to a file. this file will be used by Prediction.py
     2      pickle.dump(cascades, fp)
```

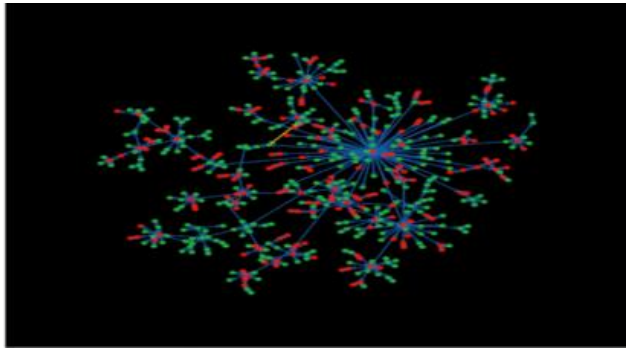
3. Simulation – Starting with different sets of starting nodes.

3.1. Iteration 1 - Starting Nodes = 60



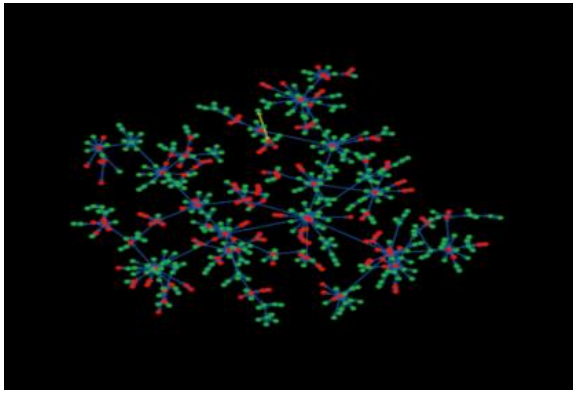
In this maximum Spread achieved is 150 in 3 steps

3.2. Iteration 2 - Starting Nodes = 100



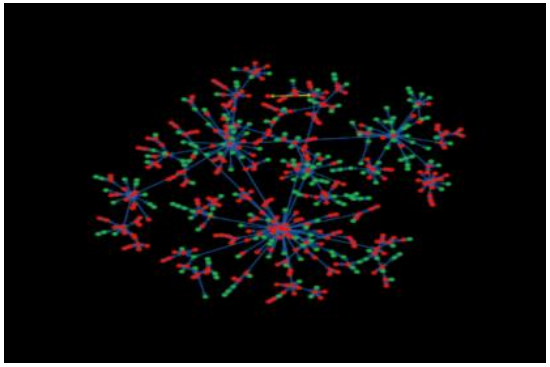
In this maximum Spread achieved is 173 in 3 steps

3.3. Iteration 3 - Starting Nodes = 76



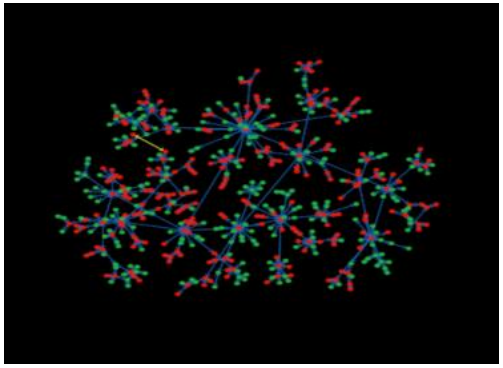
In this maximum Spread achieved is 178 in 3 steps

3.3. Iteration 4- Starting Nodes = 90



In this maximum Spread achieved is 123 in 3 steps

3.3. Iteration 5- Starting Nodes = 150



In this maximum Spread achieved is 200 in 3 steps