SNA Project

14.03.2021 and 12.05.2021

Team FALCONS:

Mehak Singhal (18ucc112)

Raghav Lakhotia (18ucs058)

Deepanshu Garg (18ucs068)

Supervisor:

Dr. Sakti Balan

INDEX

PROJECT ROUND 2

Problem Statement 1:

12. Calculation of N_G/N and its intuition
13. Finding Communities unitil 5 Steps with In Depth Explanation
    a. Girvan Newman Algorithm
    b. Ravasaz Algorithm

Problem Statement 2:

14. Making Random graph with Scale free Property
15. ICM
16. Summary
17. Reference

PROJECT ROUND 1

Overview

The Project has two datasets (Facebook social media and wikipedia vote network), implementation of different centralities along with clustering coefficient, reciprocity, and transitivity has been made. Code is written in python using a network library for manipulation graph dataset.

Goals

1. Learning about the Network package of python.
2. Implementing different centralities for the actual dataset.

Dataset Specifications

Following are two Dataset; one is the social circle of Facebook, which is an undirected graph because in Facebook if we accept the friendship, it is a two-way network. Another Dataset is the Wikipedia Vote network. It is a Directed type of graph because it's not necessary if a webpage has a backlink to another website, vice versa may not be true.

1. Social Circle : Facebook (UNDIRECTED EDGES)

This dataset consists of 'circles' (or 'friends lists') from Facebook. Facebook data was collected from survey participants using Facebook App. The dataset includes node features (profiles), circles, and ego networks.

Facebook data has been anonymized by replacing the Facebook-internal ids for each user with a new value. Also, while feature vectors from this dataset have been provided, the interpretation of those features has been obscured.

Network Specification:

Dataset statistics	
Nodes	4039
Edges	88234
Nodes in largest WCC	4039 (1.000)
Edges in largest WCC	88234 (1.000)
Nodes in largest SCC	4039 (1.000)
Edges in largest SCC	88234 (1.000)
Average clustering coefficient	0.6055
Number of triangles	1612010
Fraction of closed triangles	0.2647
Diameter (longest shortest path)	8
90-percentile effective diameter	4.7

Dataset Link: https://snap.stanford.edu/data/ego-Facebook.html

2. Wikipedia Vote Network (DIRECTED EDGES)

Wikipedia is a free encyclopedia written collaboratively by volunteers around the world. A small part of Wikipedia contributors are administrators, who are users with access to additional technical features that aid in maintenance. In order for a user to become an administrator, a Request for adminship (RfA) is issued, and the Wikipedia community via a public discussion or a vote decides who to promote to adminship. Using the latest complete dump of Wikipedia page edit history (from January 3, 2008), we extracted all administrator elections and voting history data. This gave us 2,794 elections with 103,663 total votes and 7,066 users participating in the elections (either casting a vote or being voted on). Out of these, 1,235 elections resulted in a successful promotion, while 1,559 elections did not result in the promotion. About half of the dataset's votes are by existing admins, while the other half comes from ordinary Wikipedia users.

Network Specification:

Dataset statistics	
Nodes	7115
Edges	103689
Nodes in largest WCC	7066 (0.993)
Edges in largest WCC	103663 (1.000)
Nodes in largest SCC	1300 (0.183)
Edges in largest SCC	39456 (0.381)
Average clustering coefficient	0.1409
Number of triangles	608389
Fraction of closed triangles	0.04564
Diameter (longest shortest path)	7
90-percentile effective diameter	3.8

Dataset Link: https://snap.stanford.edu/data/wiki-Vote.html

Playing with Graph:

1. Importing Libraries

```
import networkx as nx
import matplotlib.pyplot as plt
```

2. Importing Dataset

```
G1 =nx.read_edgelist("social_facebook_dataset.txt", create_using = nx.Graph(), nodetype=int)
G2 =nx.read_edgelist("Wiki-Vote.txt", create_using = nx.Graph(), nodetype=int)
```

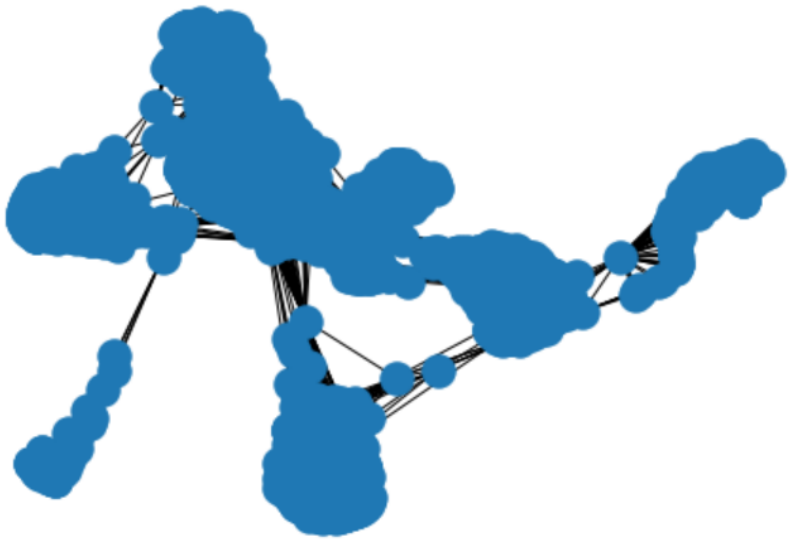
3. Graph Information

```
# Infomatio About the Graph
print(nx.info(G1))
print(nx.info(G2))
```

```
Name:
Type: Graph
Number of nodes: 4039
Number of edges: 88234
Average degree:  43.6910
Name:
Type: Graph
Number of nodes: 7115
Number of edges: 100762
Average degree:  28.3238
```

4. Plotting Graph

```
#Plotting Graph : Social Facebook Network
nx.draw(G1)
```



```
#Plotting Graph : Wikipedia Network
nx.draw(G2)
```

Problem Statement 1:

Find all centrality measures, clustering coefficients (both local and global) and reciprocity and transitivity. We have studied in the class using appropriate algorithms (you may use specific packages for this or write your own algorithm for the same).

1. All Centrality Measure:

- **Degree Centrality**

Degree centrality is a simple count of the total number of connections linked to a vertex. Degree is the measure of the total number of edges connected to a particular vertex. For our data set, the user rating a large number of books and a book with a large number of ratings will be more central according to this metric.

There are two types of degree centrality for a directed graph:

1. In Degree : no of Incoming edges

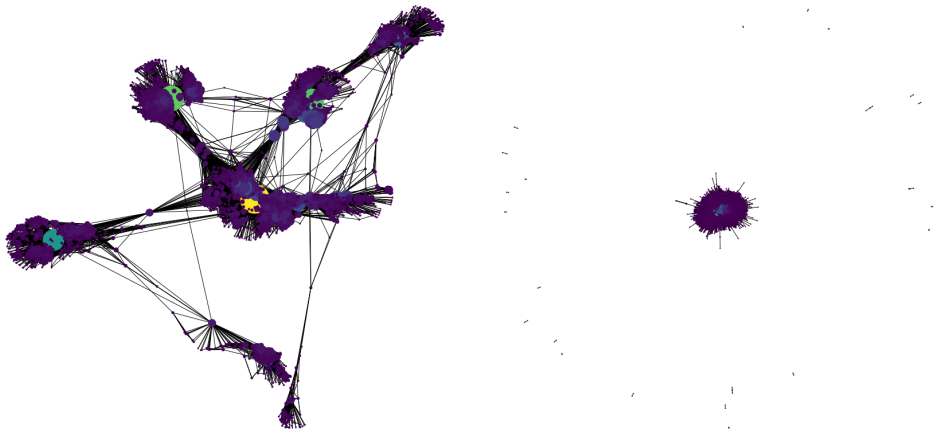
2. Out Degree : no of Outgoing edges

The degree distribution obtained for our sample was:

```python
def degreeCentrality(G):
    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')
    #Printing Top 10 Nodes as per Degree Centrality
    print("Printing Top 10 Nodes as per Degree Centrality")
    sorted_degree=sorted(degCent, key=degCent.get, reverse=True)[:10]
    for d in sorted_degree[:10]:
        print("Node Label: ",d,"=>",G.degree[d])

#Facebook Network Centality
print("Facebook Network Degree Centality:\n")
degreeCentrality(G1)
# Wikipedia Network Degree Centality
print("\nWikipedia Network Degree Centality:\n")
degreeCentrality(G2)
```

```
Facebook Network Degree Centality:

Printing Top 10 Nodes as per Degree Centrality
Node Label:  107 => 1045
Node Label:  1684 => 792
Node Label:  1912 => 755
Node Label:  3437 => 547
Node Label:  0 => 347
Node Label:  2543 => 294
Node Label:  2347 => 291
Node Label:  1888 => 254
Node Label:  1800 => 245
Node Label:  1663 => 235
```

```
Wikipedia Network Degree Centality:

Printing Top 10 Nodes as per Degree Centrality
Node Label:  2565 => 1065
Node Label:  766 => 773
Node Label:  11 => 743
Node Label:  1549 => 740
Node Label:  457 => 732
Node Label:  1166 => 688
Node Label:  2688 => 618
Node Label:  1374 => 533
Node Label:  1151 => 517
Node Label:  5524 => 495
```



- **Eigenvector Centrality**

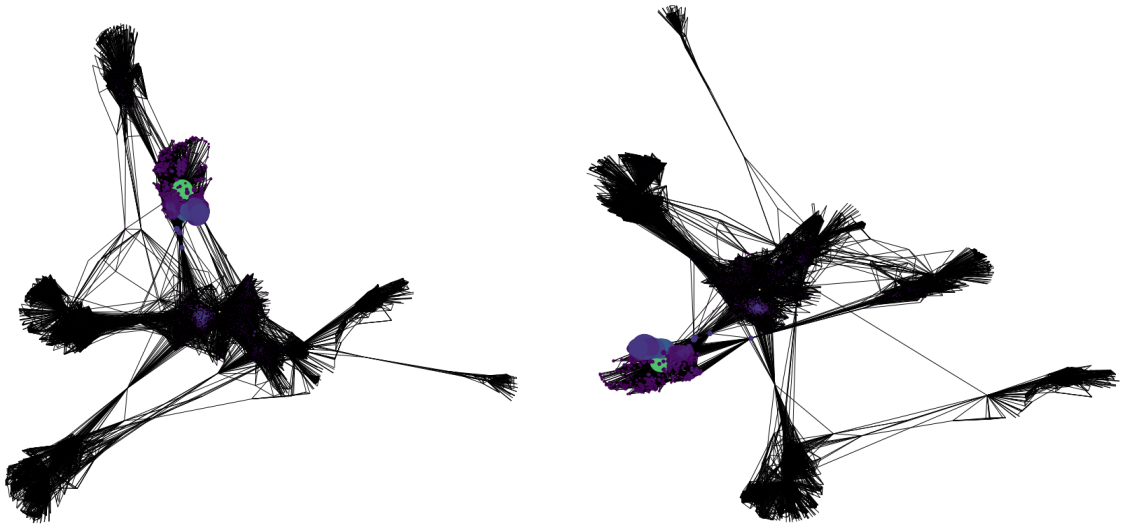
Eigenvector centrality measures a node's importance while considering the importance of its neighbors. It is sometimes used to measure a node's influence in the network.

It is determined by performing a matrix calculation to determine what is called the principal eigenvector using the adjacency matrix.

```
def eigenVectorCentrality(G):
    pos = nx.spring_layout(G1)
    eigCent = nx.eigenvector_centrality(G1)
    node_color = [20000.0 * G1.degree(v) for v in G1]
    node_size =  [v * 10000 for v in eigCent.values()]
    plt.figure(figsize=(15,15))
    nx.draw_networkx(G1, pos=pos, with_labels=False,
                     node_color=node_color,
                     node_size=node_size )
    plt.axis('off')
    print("Printing Top 10 Nodes as per Eigenvector Centrality")
    sorted_degree=sorted(eigCent, key=eigCent.get, reverse=True)[:10]
    for d in sorted_degree[:10]:
        print("Node Label: ",d,"=>",G1.degree[d])

#Facebook Network Eigenvector Centality
print("Facebook Network Eigenvector Centality:\n")
eigenVectorCentrality(G1)
# Wikipedia Network Eigenvector Centality
print("\nWikipedia Network Eigenvector Centality:\n")
eigenVectorCentrality(G2)
```

```
Facebook Network Eigenvector Centality:

Printing Top 10 Nodes as per Degree Centrality
Node Label:  107 => 1045
Node Label:  1684 => 792
Node Label:  1912 => 755
Node Label:  3437 => 547
Node Label:  0 => 347
Node Label:  2543 => 294
Node Label:  2347 => 291
Node Label:  1888 => 254
Node Label:  1800 => 245
Node Label:  1663 => 235
```

```
Wikipedia Network Eigenvector Centality:

Printing Top 10 Nodes as per Eigenvector Centrality
Node Label:  1912 => 755
Node Label:  2266 => 234
Node Label:  2206 => 210
Node Label:  2233 => 222
Node Label:  2464 => 202
Node Label:  2142 => 221
Node Label:  2218 => 205
Node Label:  2078 => 204
Node Label:  2123 => 203
Node Label:  1993 => 203
```



- **KatzCentrality**

Eigenvector centrality is not ideal for directed graphs because eigenvector centrality would not take zero in-degree nodes into account in directed graphs. Katz centrality computes the relative influence of a node within a network by measuring the number of the immediate neighbors (first degree nodes) and also all other nodes in the network that connect to the node under consideration through these immediate neighbors.

```
def katzCentrality(G):
    pos = nx.spring_layout(G)
    katzCent = nx.katz_centrality(G)
    node_color = [20000.0 * G.degree(v) for v in G]
    node_size =  [v * 10000 for v in katzCent.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')
    #Printing Top 10 Nodes as per Eigenvector Centrality
    sorted_degree=sorted(katzCent, key=katzCent.get, reverse=True)[:10]
    for d in sorted_degree[:10]:
        print("Node Label: ",d,"=>",G.degree[d])
#Facebook Network Katz Centality
print("Facebook Network Katz Centality:\n")
katzCentrality(G1)
# Wikipedia Network Katz Centality
print("\nWikipedia Network Katz Centality:\n")
katzCentrality(G2)
```

- ## Pagerank

Page Rank can be considered as an extension of Katz centrality . The websites on the web can be represented as a directed graph, where hypermedia links between websites determine the edges.

Let's consider a popular web directory website with high Katz centrality value which has millions of links to other websites. It would contribute to every single website significantly, nevertheless not all of them are important.

PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more

important websites are likely to receive more links from other websites.

```
def pagerankCentrality(G):
    pos = nx.spring_layout(G)
    pr = nx.pagerank(G, alpha = 0.8)
    node_color = [20000.0 * G.degree(v) for v in G]
    node_size =  [v * 10000 for v in pr.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')
    #Printing Top 10 Nodes as per Eigenvector Centrality
    sorted_degree=sorted(pr, key=pr.get, reverse=True)[:10]
    for d in sorted_degree[:10]:
        print("Node Label: ",d,"=> Degree: ",G.degree[d])
#Facebook Network Pagerank Centality
print("Facebook Network Pagerank Centality:\n")
pagerankCentrality(G1)
# Wikipedia Network Pagerank Centality
print("\nWikipedia Network Pagerank Centality:\n")
pagerankCentrality(G2)
```

```
Facebook Network Pagerank Centality:

Node Label:  3437 => Degree:  547
Node Label:  107 => Degree:  1045
Node Label:  1684 => Degree:  792
Node Label:  0 => Degree:  347
Node Label:  1912 => Degree:  755
Node Label:  348 => Degree:  229
Node Label:  686 => Degree:  170
Node Label:  3980 => Degree:  59
Node Label:  414 => Degree:  159
Node Label:  698 => Degree:  68
```

```
Wikipedia Network Pagerank Centality:

Node Label:  2565 => Degree:  1065
Node Label:  4037 => Degree:  467
Node Label:  11 => Degree:  743
Node Label:  457 => Degree:  732
Node Label:  766 => Degree:  773
Node Label:  1549 => Degree:  740
Node Label:  1166 => Degree:  688
Node Label:  2688 => Degree:  618
Node Label:  15 => Degree:  403
Node Label:  2237 => Degree:  387
```

Betweenness Centrality

Betweenness centrality measures how important a node is to the shortest paths through the network.

To compute betweenness for a node N, we select a pair of nodes and find all the shortest paths between those nodes. Then we compute the fraction of those shortest paths that include node N.

We repeat this process for every pair of nodes in the network. We then add up the fractions we computed, and this is the betweenness centrality for node N

```
def betweenessCentrality(G):
    pos = nx.spring_layout(G)
    betCent = nx.betweenness_centrality(G)
    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')
    #Printing Top 10 Nodes as per Betweeness Centrality
    sorted_degree=sorted(betCent, key=betCent.get, reverse=True)[:10]
    for d in sorted_degree[:10]:
        print("Node Label: ",d,"=> Degree: ",G.degree[d])
#Facebook Network Betweeness Centality
print("Facebook Network Betweeness Centality:\n")
betweenessCentrality(G1)
# Wikipedia Network Betweeness Centality
print("\nWikipedia Network Betweeness Centality:\n")
betweenessCentrality(G2)
```

```
Facebook Network Betweeness Centality:

Node Label:  107 => Degree:  1045
Node Label:  1684 => Degree:  792
Node Label:  3437 => Degree:  547
Node Label:  1912 => Degree:  755
Node Label:  1085 => Degree:  66
Node Label:  0 => Degree:  347
Node Label:  698 => Degree:  68
Node Label:  567 => Degree:  63
Node Label:  58 => Degree:  12
Node Label:  428 => Degree:  115
```

Wikipedia Network Betweeness Centality:

```
Node Label:  2565 => Degree:  1065
Node Label:  11 => Degree:  743
Node Label:  457 => Degree:  732
Node Label:  4037 => Degree:  467
Node Label:  1549 => Degree:  740
Node Label:  766 => Degree:  773
Node Label:  1166 => Degree:  688
Node Label:  15 => Degree:  403
Node Label:  1374 => Degree:  533
Node Label:  2237 => Degree:  387
```

• Closeness Centrality

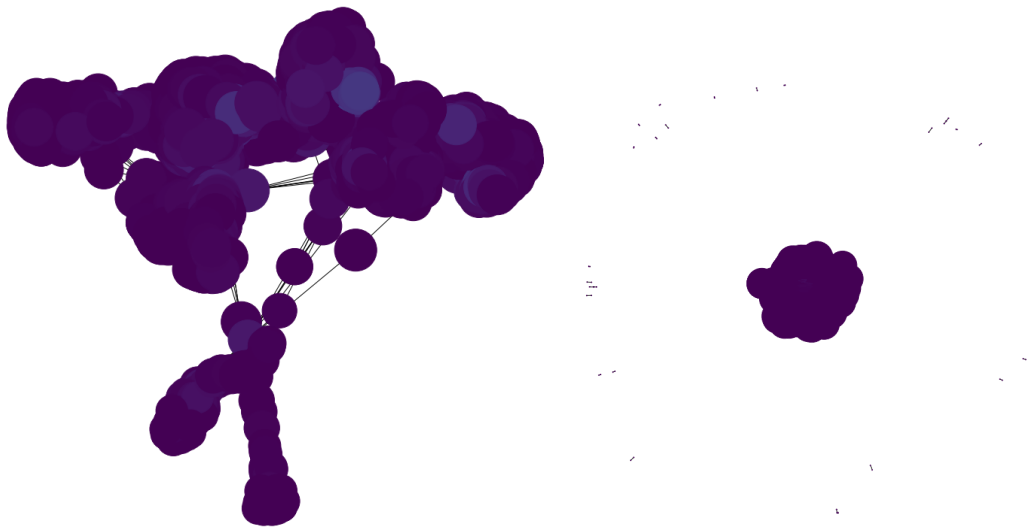
For each node, the Closeness Centrality algorithm calculates the sum of its distances to all other nodes,based on calculating the shortest paths between all pairs of nodes.

The closeness centrality for our sample came out to be:

```
def closenessCentrality(G):
    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')
    #Printing Top 10 Nodes as per Closeness Centrality
    sorted_degree=sorted(cloCent, key=cloCent.get, reverse=True)[:10]
    for d in sorted_degree[:10]:
        print("Node Label: ",d,"=> Degree: ",G.degree[d])
#Facebook Network Closeness Centality
print("Facebook Network Closeness Centality:\n")
closenessCentrality(G1)
# Wikipedia Network Closeness Centality
print("\nWikipedia Network Closeness Centality:\n")
closenessCentrality(G2)
```

```
Facebook Network Closeness Centality:

Node Label:  107 => Degree:  1045
Node Label:  58 => Degree:  12
Node Label:  428 => Degree:  115
Node Label:  563 => Degree:  91
Node Label:  1684 => Degree:  792
Node Label:  171 => Degree:  22
Node Label:  348 => Degree:  229
Node Label:  483 => Degree:  231
Node Label:  414 => Degree:  159
Node Label:  376 => Degree:  133
```

```
Wikipedia Network Closeness Centality:

Node Label:  2565 => Degree:  1065
Node Label:  766 => Degree:  773
Node Label:  457 => Degree:  732
Node Label:  1549 => Degree:  740
Node Label:  1166 => Degree:  688
Node Label:  1374 => Degree:  533
Node Label:  11 => Degree:  743
Node Label:  1151 => Degree:  517
Node Label:  2688 => Degree:  618
Node Label:  2485 => Degree:  435
```



- Group Betweenness Centrality

```
def groupBetweenCentrality(G):
    pos = nx.spring_layout(G)
    gbCent = nx.group_betweenness_centrality(G)
    node_color = [20000.0 * G.degree(v) for v in G]
    node_size =  [v * 10000 for v in gbCent.values()]
    plt.figure(figsize=(13,13))
    nx.draw_networkx(G1, pos=pos, with_labels=False,
                     node_color=node_color,
                     node_size=node_size )
    plt.axis('off')
    #Printing Top 10 Nodes as per Closeness Centrality
    sorted_degree=sorted(gbCent, key=gbCent.get, reverse=True)[:10]
    for d in sorted_degree[:10]:
        print("Node Label: ",d,"=> Degree: ",G.degree[d])

#Facebook Network Group Betweeeness Centality
print("Facebook Network Group Betweeeness Centality:\n")
groupBetweenCentrality(G1)
# Wikipedia Network Group Betweeeness Centality
print("\nWikipedia Network Group Betweeeness Centality:\n")
groupBetweenCentrality(G2)
```

2. Clustering Coefficient

The clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together.

In most real-world networks, and in particular social networks, nodes tend to create tightly knit groups characterized by a relatively high density of ties, this likelihood tends to be greater than the average probability of a tie randomly established between two nodes.

Two versions of this measure exist: the global and the local. The global version was designed to give an overall indication of the clustering in the network, whereas the local gives an indication of the embeddedness of single nodes.

- Local and Global Clustering Coefficient|

```
all_nodes = list(G1.nodes())
for node in all_nodes:
    local_clustering_coefficient=nx.clustering(G1,node)
    global_clustering_coefficient=(3*nx.triangles(G1,node))/nx.transitivity(G1)
    print("Local Clustering Coefficient of ",node," = ",local_clustering_coefficient)
    print("Global Clustering Coefficient of ",node," = ",global_clustering_coefficient)
```

- Average Clustering Coefficient

```
print("No. Of Nodes of Facebook Social Graph:",G1.number_of_nodes())
print("No. Of Edges of Facebook Social Graph",G1.number_of_edges())
print("No. Of Nodes of Wikipedia Network Graph:",G2.number_of_nodes())
print("No. Of Edges of Wikipedia Network Graph:",G2.number_of_edges())

# Average Clustering Coefficient
average_clustering_coefficient_G1=nx.average_clustering(G1)
average_clustering_coefficient_G2=nx.average_clustering(G2)
print("Average Clustering Coefficient of Facebook Social Graph: ", average_clustering_coefficient_G1)
print("Average Clustering Coefficient of Wikipedia Network Graph: ", average_clustering_coefficient_G2)
```

```
No. Of Nodes of Facebook Social Graph: 4039
No. Of Edges of Facebook Social Graph 88234
No. Of Nodes of Wikipedia Network Graph: 7115
No. Of Edges of Wikipedia Network Graph: 100762
Average Clustering Coefficient of Facebook Social Graph:  0.6055467186200876
Average Clustering Coefficient of Wikipedia Network Graph:  0.14089784589308738
```

3. Reciprocity:

Reciprocity is a measure of the likelihood of vertices in a directed network to be mutually linked. Like the clustering coefficient, scale-free degree distribution, or community structure, reciprocity is a quantitative measure used to study complex networks.

1.3 Reciprocity

```
print("Reciprocity of Facebook Social Graph is: ",nx.overall_reciprocity(G1))
print("Reciprocity of Wikipedia Vote Network Graph is: ",nx.overall_reciprocity(G2))
```

```
Reciprocity of Facebook Social Graph is:  0.0
Reciprocity of Wikipedia Vote Network Graph is:  0.0
```

4. Transitivity:

Transitivity refers to the extent to which the relation that relates two nodes in a network that are connected by an edge is transitive. Perfect transitivity implies that if x is connected to y , and y is connected to z , then x is connected to z as well.

1.4 Transitivity

```
print("Transitivity of Facebook Social Graph is: ",nx.transitivity(G1))
print("Transitivity of Wikipedia Vote Network Graph is: ",nx.transitivity(G2))
```

```
Transitivity of Facebook Social Graph is:  0.5191742775433075
Transitivity of Wikipedia Vote Network Graph is:  0.12547914899233995
```

Problem Statement 2:

Try to get an algorithm package in Python to find the maximum connected component (called a giant component in the class) in a given graph G. Let us denote the number of nodes in the giant component of a graph G as N_G. Vary ⟨k⟩ from 0 to 5 with increment of 0.1. For each value of ⟨k⟩ find the ratio N_G/N where N is the number of nodes in the graph. Plot this ratio with respect to ⟨k⟩. Take ⟨k⟩ as x-axis and ratio N_G/N as y-axis.

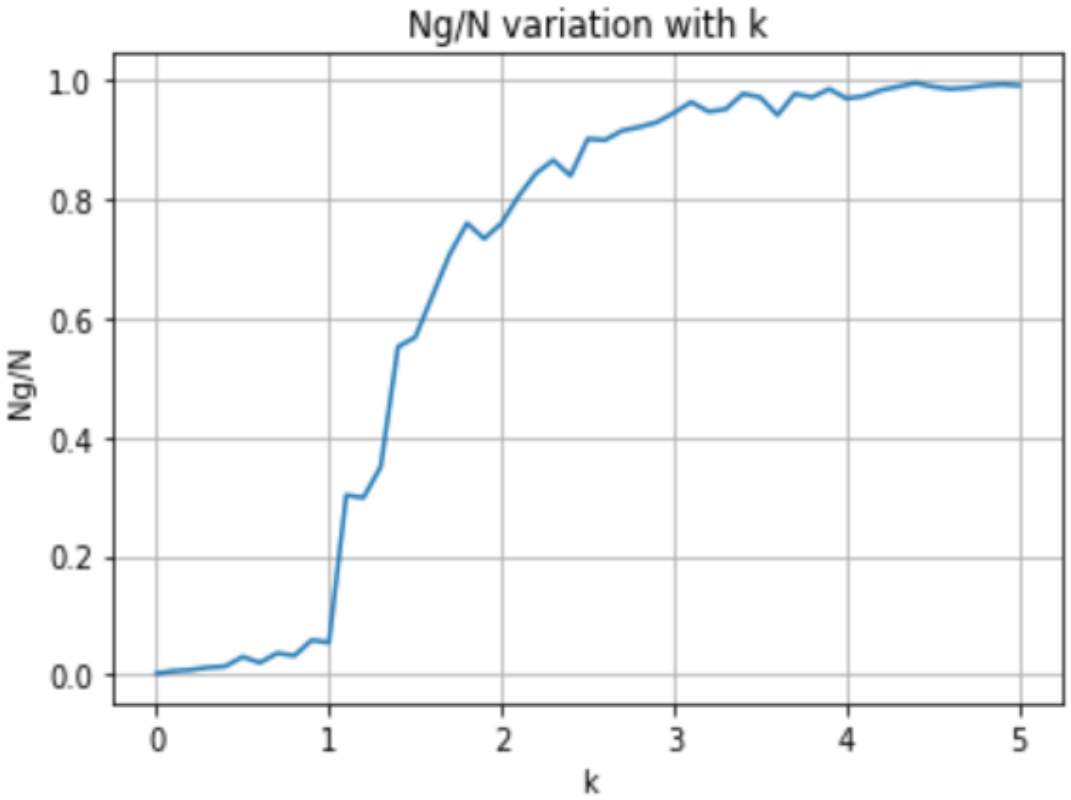
Problem Statement 2: Giant component variation

```
giant_facebook = len(max(nx.connected_components(G1), key=len))
print("Number of node in the Giant Componenet of G ="+str(giant_facebook)+'\n')
giant_wiki = len(max(nx.connected_components(G2), key=len))
print("Number of node in the Giant Componenet of G ="+str(giant_wiki)+'\n')
```

```
Number of node in the Giant Componenet of G =4039

Number of node in the Giant Componenet of G =7066
```

```
# Importing required libraries
import networkx as nx
import matplotlib.pyplot as plt
import pylab as plt

n = 500
arr = {} # Dictionary to store k and corresponding Ng/N value
for i in range(0,51):
    k = float(i/10) # k varies by 0.1
    p = k/(n-1)
    g = nx.erdos_renyi_graph(n, p)
    # finding Giant component and ng
    giant = max(nx.connected_components(g), key=len)
#     ng = giant_components(giant)
    ng = len(giant)
    # adding the ratio to dictionary
    arr[k] = float(ng/n)
# since dictionary is unsorted collection, sorting wrt k for plotting
temp = sorted(arr.items())
# unzipping keys and values of dictionary into x and y
# x takes the value of k , y takes the ratio in sorted order
x,y = zip(*temp)
```

```
# plotting the histogram
plt.figure()
plt.grid(True)
plt.plot(x,y)
plt.title('Ng/N variation with k')
plt.xlabel( 'k' )
plt.ylabel( 'Ng/N' )
plt.show()
plt.close()
```



1. Subcritical Regime:
k varies from 0 to 1. In this region we get tiny clusters.
2. Supercritical regime: k>1. This regime lasts until all nodes are absorbed by the giant component.

3. Connected Regime: K> lnN. In the absence of isolated nodes the network becomes connected.
4. Critical Point: k=1

Conclusion:

Erdős-Rényi Network model of random graphs was taken with n=500 nodes. The average degree varied from 0 to 5 with incrementation of 0.1 for each iteration and a giant component was observed in each graph iteration.The plot increased from 0 and as we kept increasing the average degree, the ratio kept getting closer and closer to 1. This observation is in accordance with our preliminary analysis, where we said that as the probability of edge formation p increases, (p = k/n-1, which implies p is directly proportional to average degree k), the size of the giant component increases.

PROJECT ROUND 2

Problem Statement 1:

Using the dataset that you had selected for the first round create a network (for example, if the network is not already there as a dataset then if it is a social media twitter data create the follower/following graph/reply/retweet networks). Once you get the network do the following:

a) Find the giant component G in the network (note that giant components are the largest connected subgraph of the constricted network/graph). Let N_G denote the number of nodes in G. Find N_G/N where N is the total number of nodes in the network.

b)Apply Girvan Newman algorithm and Ravasaz algorithm to find the communities step by step and illustrate each step the communities got and stop after 5 steps

Show all the communities and give your understanding about the communities that you got through the algorithm. All these things should be reported in the report that you are going to submit. Looking at the output given by the two algorithms compare and contrast the two algorithms.

Part a) Value of Ratio: N_G/N for both the Graph

```
# Calculation of N for both networks
total_nodes_facebook = G1.number_of_nodes()
total_nodes_wiki = G2.number_of_nodes()

# Calculation of N_G for both networks
nodes_in_giant_component_facebook_ = len(max(nx.connected_components(G1), key=len))
nodes_in_giant_component_wiki = len(max(nx.connected_components(G2), key=len))

# Calculation of N_G/N for both networks
cal_facebook = nodes_in_giant_component_facebook_/total_nodes_facebook;
cal_wiki = nodes_in_giant_component_wiki/total_nodes_wiki;
print("Value of N_G/N for Facebook: "+str(cal_facebook)+'\n')
print("Value of N_G/N for Wiki: "+str(cal_wiki)+'\n')
```

```
Value of N_G/N for Facebook: 1.0

Value of N_G/N for Wiki: 0.9931131412508785
```

Intuition:

Giant Component of facebook (node that we selected) are all connected and its equal to the number of total nodes, where in wiki there are few nodes which are not been a part of the giant component.

Part b.1) Girvan Newman Algorithm Communities after each step till 5 Rounds

Girvan Newman Algorithm :

The Girvan Newman technique for the detection and analysis of community structure depends upon the iterative elimination of edges with the highest number of the shortest paths that pass through them. By getting rid of the edges, the network breaks down into smaller networks, i.e. communities. The idea behind this algorithm is to find which edges in a network occur most frequently between other pairs of nodes by finding the edges betweenness. The edges joining communities are then expected to have high edge betweenness. The underlying community structure of the network will be much more fine-grained once we eliminate edges with high edge betweenness. For the removal of each edge, the calculation of edge betweenness is O(EN), therefore, this algorithm's time complexity is O(E^2N).

We can express the Girvan-Newman algorithm in the following procedure:

1) Calculate edge betweenness for every edge in the graph.
2) Remove the edge with the highest edge betweenness.
3) Calculate edge betweenness for remaining edges.
4) Repeat steps 2–4 until all edges are removed.

Creation of Random Network from Dataset 1: Facebook

As the dataset of Facebook is immensely large we will take Random Graph with 500 Edges.

```
# DATSET 2 : FaceBook
import random
s = random.sample(G1.edges(),500)
G = nx.Graph()
G.add_edges_from(s)
print(nx.info(G))

# Ng/N for Random Graph Generated
total = nx.number_connected_components(G);
print("Number of Nodes in Connected Random Graph of Facebook: "+str(total)+'\n')
nodes_in_giant_random_ = len(max(nx.connected_components(G), key=len))
print("Number of Nodes in Giant Random Graph of Facebook: "+str(nodes_in_giant_random_)+'\n')
val = nodes_in_giant_random_/total
print("Value of N_G/N for Random Graph of Facebook: "+str(val)+'\n')

# We can express the Girvan-Newman algorithm in the following procedure:
# 1) Remove the edge with the highest edge betweenness.
# 2) Calculate edge betweenness for remaining edges.
# 3) Repeat steps 1-3 until all edges are removed.
```

OUTPUT:

```
Name:
Type: Graph
Number of nodes: 780
Number of edges: 500
Average degree:   1.2821
Number of Nodes in Connected Random Graph of Facebook: 280

Number of Nodes in Giant Random Graph of Facebook: 17

Value of N_G/N for Random Graph of Facebook: 0.060714285714285714
```

ITERATION 1:

Code Snippet for Applying Girvan :

```
# Iteration 1
print("Iteration 1:")
#Step1:
print("Initial Value of Connected Componenets:"+str(nx.number_connected_components(G)))
b = max(nx.edge_betweenness_centrality(G));
print("Value of highest edge betweenness: "+str(b))

#Step2:
print("Highest Edge Betweenness Removed:")
G.remove_edge(*b)

#Step3:
print("Value of Connected Componenets after Removal of Edge:"+str(nx.number_connected_components(G)))
nx.draw(G,pos=None, node_color="r",edge_color="b", node_size=1)

#All the Communities
S = [G.subgraph(c).copy() for c in nx.connected_components(G)]
for i in S:
    print(i.nodes())
    print("************************************************")
```

Output:

```
Iteration 1:
Initial Value of Connected Componenets:297
Value of highest edge betweenness: (3980, 4009)
Highest Edge Betweenness Removed:
Value of Connected Componenets after Removal of Edge:298
[2560, 2564, 2181, 2183, 2190, 2324, 2069, 2454, 2073, 2460, 2590, 2206, 2464, 1953, 2339,
2104, 2360, 2237, 1983, 2495, 2114, 2630, 2508, 2395, 2526, 2655, 2276, 2278, 2026, 2290,
************************************************
[2138, 1987]
************************************************
[2912, 2673, 3108, 2669]
************************************************
[1059, 1199, 1556, 1399, 1535]
************************************************
[3737, 3611]
************************************************
[1752, 1793, 1589, 1305]
************************************************
[1024, 1861, 1130, 1809, 1214]
************************************************
```

Graph:

Take-Away:

Number of connected nodes rose from 297 to 298.

ITERATION 2:

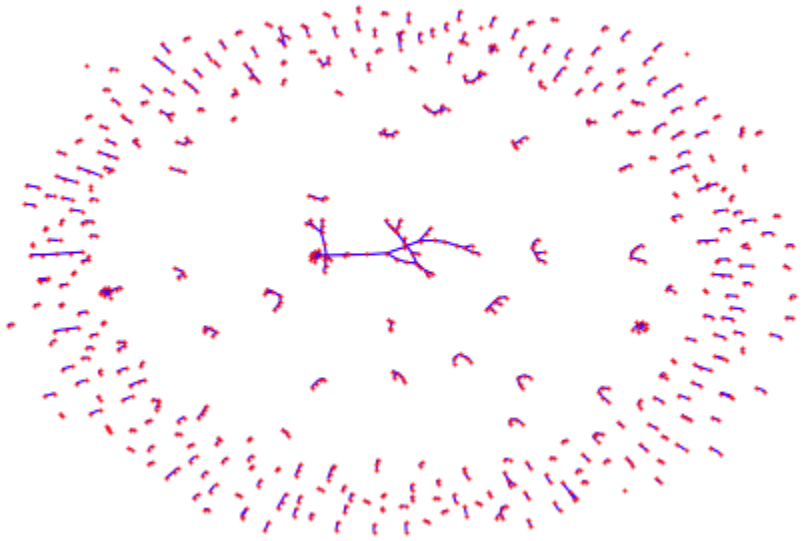
Code Snippet for Applying Girvan :

```
# Iteration 2
print("Iteration 2:")
#Step1:
print("Initial Value of Connected Componenets:"+str(nx.number_connected_components(G)))
b = max(nx.edge_betweenness_centrality(G));
print("Value of highest edge betweenness: "+str(b))

#Step2:
print("Highest Edge Betweenness Removed:")
G.remove_edge(*b)

#Step3:
print("Value of Connected Componenets after Removal of Edge:"+str(nx.number_connected_components(G)))
nx.draw(G,pos=None, node_color="r",edge_color="b", node_size=1)

#All the Communities
S = [G.subgraph(c).copy() for c in nx.connected_components(G)]
for i in S:
    print(i.nodes())
    print("************************************************")
```

Output:

```
Iteration 2:
Initial Value of Connected Componenets:298
Value of highest edge betweenness: (3930, 3941)
Highest Edge Betweenness Removed:
Value of Connected Componenets after Removal of Edge:299
[2560, 2564, 2181, 2183, 2190, 2324, 2069, 2454, 2073, 2460, 2590, 2206, 2464, 1953, 2339, 2340,
2104, 2360, 2237, 1983, 2495, 2114, 2630, 2508, 2395, 2526, 2655, 2276, 2278, 2026, 2290, 2164,
************************************************
[2138, 1987]
************************************************
[2912, 2673, 3108, 2669]
************************************************
[1059, 1199, 1556, 1399, 1535]
************************************************
[3737, 3611]
************************************************
[1752, 1793, 1589, 1305]
************************************************
[1024, 1861, 1130, 1809, 1214]
************************************************
```

Graph:



ITERATION 3:

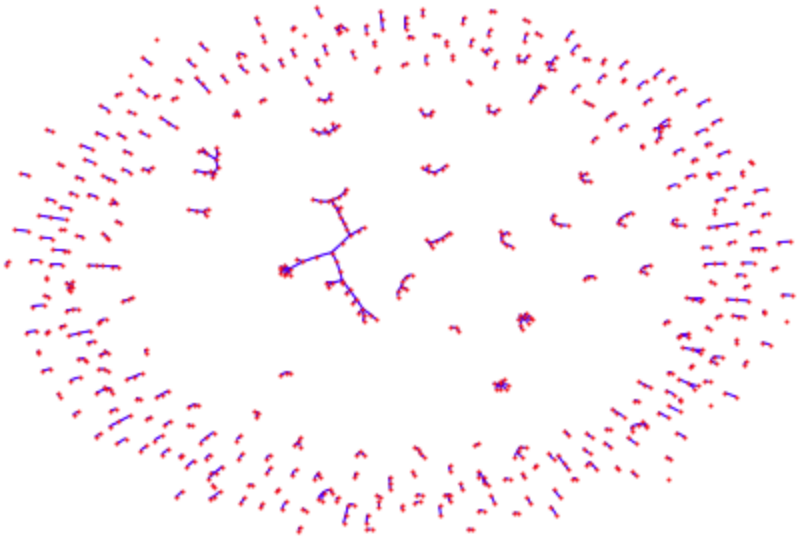
Code Snippet for Applying Girvan :

```
# Iteration 3
print("Iteration 3:")
#Step1:
print("Initial Value of Connected Componenets:"+str(nx.number_connected_components(G)))
b = max(nx.edge_betweenness_centrality(G));
print("Value of highest edge betweenness: "+str(b))

#Step2:
print("Highest Edge Betweenness Removed:")
G.remove_edge(*b)

#Step3:
print("Value of Connected Componenets after Removal of Edge:"+str(nx.number_connected_components(G)))
nx.draw(G,pos=None, node_color="r",edge_color="b", node_size=1)

#All the Communities
S = [G.subgraph(c).copy() for c in nx.connected_components(G)]
for i in S:
    print(i.nodes())
    print("************************************************")
```

Output:

```
Iteration 3:
Initial Value of Connected Componenets:299
Value of highest edge betweenness: (3886, 3552)
Highest Edge Betweenness Removed:
Value of Connected Componenets after Removal of Edge:300
[2560, 2564, 2181, 2183, 2190, 2324, 2069, 2454, 2073, 2460, 2590, 2206, 2464, 1953, 2339,
2104, 2360, 2237, 1983, 2495, 2114, 2630, 2508, 2395, 2526, 2655, 2276, 2278, 2026, 2290,
************************************************
[2138, 1987]
************************************************
[2912, 2673, 3108, 2669]
************************************************
[1059, 1199, 1556, 1399, 1535]
************************************************
[3737, 3611]
************************************************
[1752, 1793, 1589, 1305]
************************************************
[1024, 1861, 1130, 1809, 1214]
************************************************
```

Graph:



ITERATION 4:

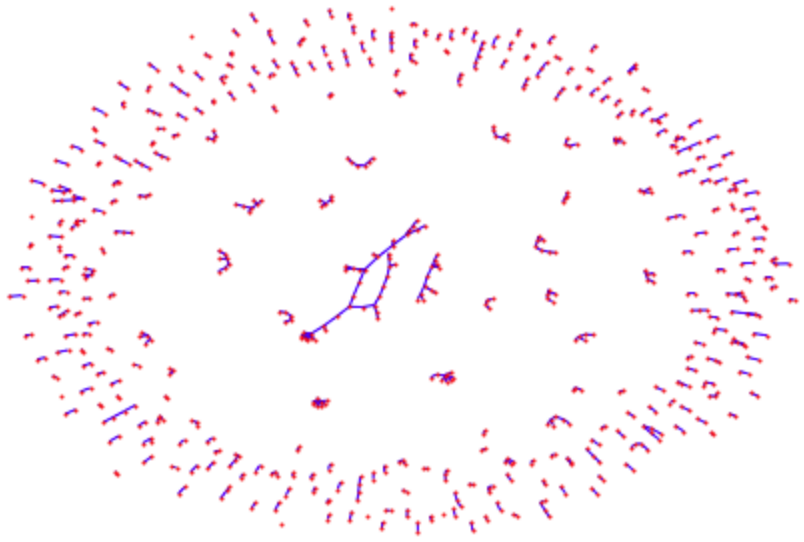
Code Snippet for Applying Girvan :

```
# Iteration 4
print("Iteration 4:")
#Step1:
print("Initial Value of Connected Componenets:"+str(nx.number_connected_components(G)))
b = max(nx.edge_betweenness_centrality(G));
print("Value of highest edge betweenness: "+str(b))

#Step2:
print("Highest Edge Betweenness Removed:")
G.remove_edge(*b)

#Step3:
print("Value of Connected Componenets after Removal of Edge:"+str(nx.number_connected_components(G)))
nx.draw(G,pos=None, node_color="r",edge_color="b", node_size=1)

#All the Communities
S = [G.subgraph(c).copy() for c in nx.connected_components(G)]
for i in S:
    print(i.nodes())
    print("************************************************")
```

Output:

```
Iteration 4:
Initial Value of Connected Componenets:300
Value of highest edge betweenness: (3869, 3931)
Highest Edge Betweenness Removed:
Value of Connected Componenets after Removal of Edge:301
[2560, 2564, 2181, 2183, 2190, 2324, 2069, 2454, 2073, 2460, 2590, 2206, 2464, 1953, 2339, 2340, 2
2104, 2360, 2237, 1983, 2495, 2114, 2630, 2508, 2395, 2526, 2655, 2276, 2278, 2026, 2290, 2164, 25
**************************************************
[2138, 1987]
**************************************************
[2912, 2673, 3108, 2669]
**************************************************
[1059, 1199, 1556, 1399, 1535]
**************************************************
[3737, 3611]
**************************************************
[1752, 1793, 1589, 1305]
**************************************************
[1024, 1861, 1130, 1809, 1214]
**************************************************
```

Graph:



ITERATION 5:

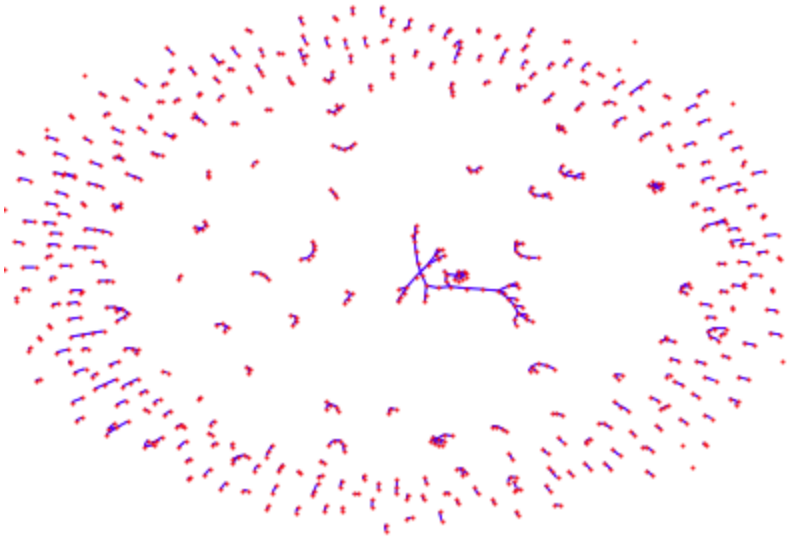
Code Snippet for Applying Girvan :

```
# Iteration 5
print("Iteration 5:")
#Step1:
print("Initial Value of Connected Componenets:"+str(nx.number_connected_components(G)))
b = max(nx.edge_betweenness_centrality(G));
print("Value of highest edge betweenness: "+str(b))

#Step2:
print("Highest Edge Betweenness Removed:")
G.remove_edge(*b)

#Step3:
print("Value of Connected Componenets after Removal of Edge:"+str(nx.number_connected_components(G)))
nx.draw(G,pos=None, node_color="r",edge_color="b", node_size=1)

#All the Communities
S = [G.subgraph(c).copy() for c in nx.connected_components(G)]
for i in S:
    print(i.nodes())
    print("**********************************************")
```

Output:

```
Iteration 5:
Initial Value of Connected Componenets:301
Value of highest edge betweenness: (3869, 3794)
Highest Edge Betweenness Removed:
Value of Connected Componenets after Removal of Edge:302
[2560, 2564, 2181, 2183, 2190, 2324, 2069, 2454, 2073, 2460, 2590, 2206, 2464, 1953, 2339, 2340,
2104, 2360, 2237, 1983, 2495, 2114, 2630, 2508, 2395, 2526, 2655, 2276, 2278, 2026, 2290, 2164,
**********
[2138, 1987]
**********
[2912, 2673, 3108, 2669]
**********
[1059, 1199, 1556, 1399, 1535]
**********
[3737, 3611]
**********
[1752, 1793, 1589, 1305]
**********
[1024, 1861, 1130, 1809, 1214]
**********
```

Graph:



WIKI

```
# DATSET 1 : Wiki
import random
s = random.sample(G2.edges(),400)
G = nx.Graph()
G.add_edges_from(s)
print(nx.info(G))

# Ng/N for Random Graph Generated
total = nx.number_connected_components(G);
print("Number of Nodes in Connected Random Graph of Wiki: "+str(total)+'\n')
nodes_in_giant_random_ = len(max(nx.connected_components(G), key=len))
print("Number of Nodes in Giant Random Graph of Wiki: "+str(nodes_in_giant_random_)+'\n')
val = nodes_in_giant_random_/total
print("Value of N_G/N for Random Graph of Wiki: "+str(val)+'\n')
```

```
Name:
Type: Graph
Number of nodes: 642
Number of edges: 400
Average degree:   1.2461
Number of Nodes in Connected Random Graph of Wiki: 242

Number of Nodes in Giant Random Graph of Wiki: 11

Value of N_G/N for Random Graph of Wiki: 0.045454545454545456
```

ITERATION 1:

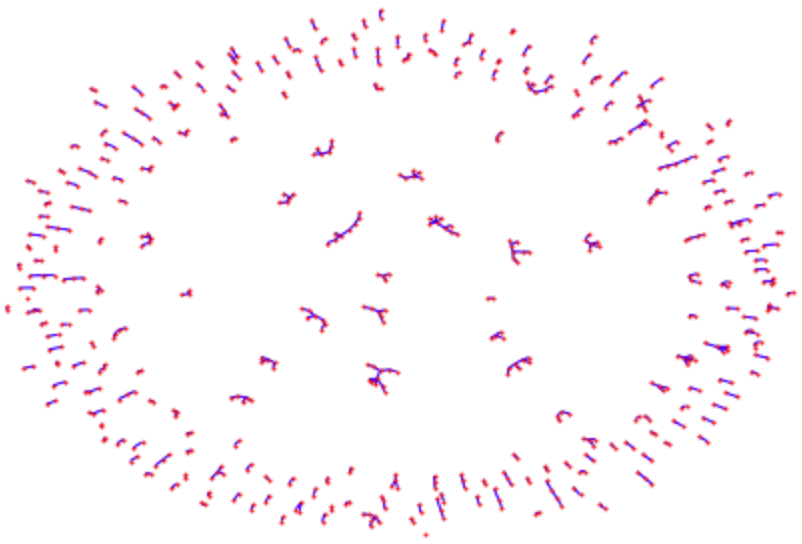
Code Snippet for Applying Girvan :

```
# Iteration 1
print("Iteration 1:")
#Step1:
print("Initial Value of Connected Componenets:"+str(nx.number_connected_components(G)))
b = max(nx.edge_betweenness_centrality(G));
print("Value of highest edge betweenness: "+str(b))

#Step2:
print("Highest Edge Betweenness Removed:")
G.remove_edge(*b)

#Step3:
print("Value of Connected Componenets after Removal of Edge:"+str(nx.number_connected_components(G)))
nx.draw(G,pos=None, node_color="r",edge_color="b", node_size=1)

#All the Communities
S = [G.subgraph(c).copy() for c in nx.connected_components(G)]
for i in S:
    print(i.nodes())
    print("************************************************")
```

Output:

```
Iteration 1:
Initial Value of Connected Componenets:242
Value of highest edge betweenness: (8292, 2822)
Highest Edge Betweenness Removed:
Value of Connected Componenets after Removal of Edge:243
[2972, 1799]
**************************************************
[1859, 11, 722, 54, 439, 2294]
**************************************************
[122, 3180, 2958]
**************************************************
[2565, 936, 8, 4713, 7373, 538, 7386, 29, 4031]
**************************************************
[416, 226, 223]
**************************************************
[5176, 4276]
**************************************************
[72, 813, 3284, 2264, 5695]
**************************************************
```

Graph:



ITERATION 2:

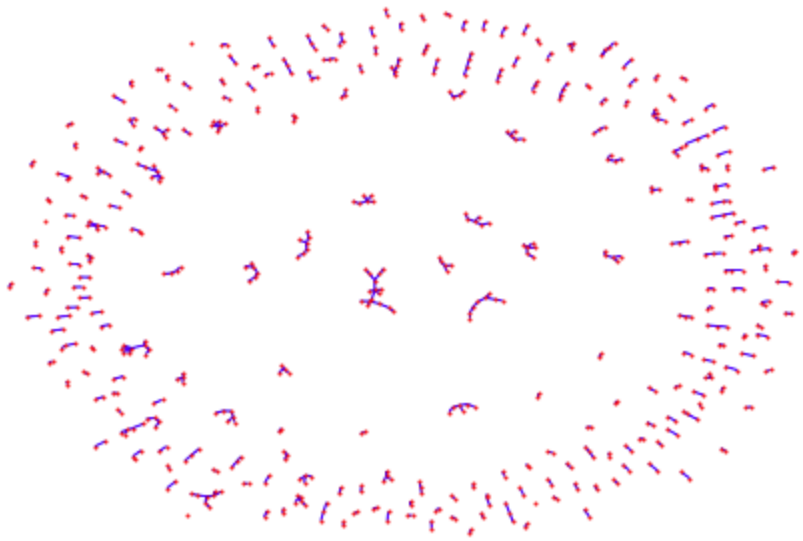
Code Snippet for Applying Girvan :

```
# Iteration 2
print("Iteration 2:")
#Step1:
print("Initial Value of Connected Componenets:"+str(nx.number_connected_components(G)))
b = max(nx.edge_betweenness_centrality(G));
print("Value of highest edge betweenness: "+str(b))

#Step2:
print("Highest Edge Betweenness Removed:")
G.remove_edge(*b)

#Step3:
print("Value of Connected Componenets after Removal of Edge:"+str(nx.number_connected_components(G)))
nx.draw(G,pos=None, node_color="r",edge_color="b", node_size=1)

#All the Communities
S = [G.subgraph(c).copy() for c in nx.connected_components(G)]
for i in S:
    print(i.nodes())
    print("************************************************")
```

Output:

```
Iteration 2:
Initial Value of Connected Componenets:243
Value of highest edge betweenness: (8219, 7685)
Highest Edge Betweenness Removed:
Value of Connected Componenets after Removal of Edge:244
[2972, 1799]
**************************************************
[1859, 11, 722, 54, 439, 2294]
**************************************************
[122, 3180, 2958]
**************************************************
[2565, 936, 8, 4713, 7373, 538, 7386, 29, 4031]
**************************************************
[416, 226, 223]
**************************************************
[5176, 4276]
**************************************************
[72, 813, 3284, 2264, 5695]
**************************************************
```

Graph:



ITERATION 3:

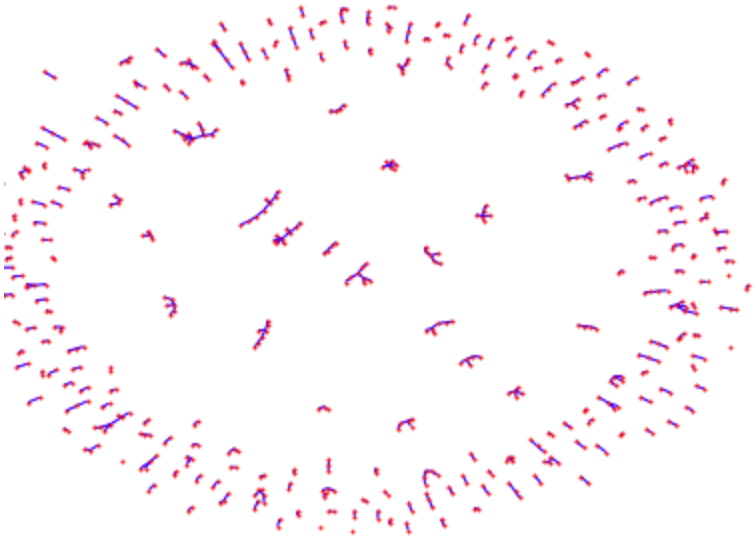
Code Snippet for Applying Girvan :

```
# Iteration 3
print("Iteration 3:")
#Step1:
print("Initial Value of Connected Componenets:"+str(nx.number_connected_components(G)))
b = max(nx.edge_betweenness_centrality(G));
print("Value of highest edge betweenness: "+str(b))

#Step2:
print("Highest Edge Betweenness Removed:")
G.remove_edge(*b)

#Step3:
print("Value of Connected Componenets after Removal of Edge:"+str(nx.number_connected_components(G)))
nx.draw(G,pos=None, node_color="r",edge_color="b", node_size=1)

#All the Communities
S = [G.subgraph(c).copy() for c in nx.connected_components(G)]
for i in S:
    print(i.nodes())
    print("************************************************")
```

Output:

```
Iteration 3:
Initial Value of Connected Componenets:244
Value of highest edge betweenness: (8212, 7865)
Highest Edge Betweenness Removed:
Value of Connected Componenets after Removal of Edge:245
[2972, 1799]
**************************************************
[1859, 11, 722, 54, 439, 2294]
**************************************************
[122, 3180, 2958]
**************************************************
[2565, 936, 8, 4713, 7373, 538, 7386, 29, 4031]
**************************************************
[416, 226, 223]
**************************************************
[5176, 4276]
**************************************************
[72, 813, 3284, 2264, 5695]
**************************************************
```

Graph:



ITERATION 4:

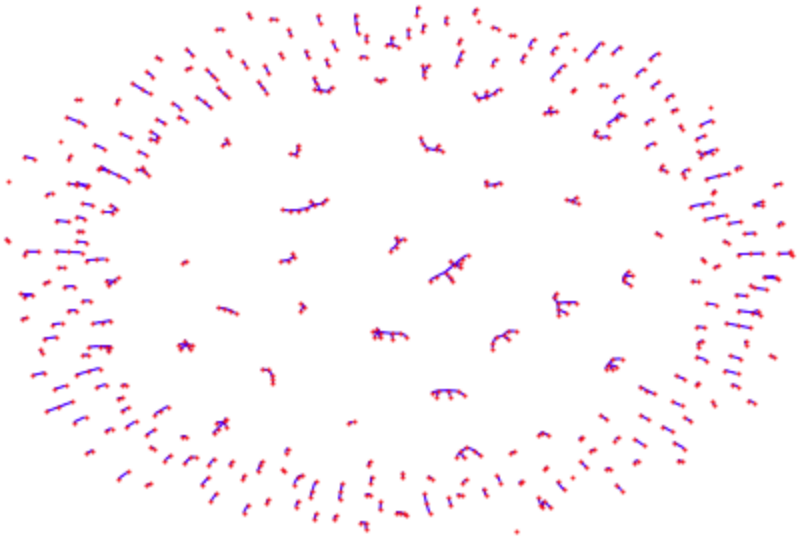
Code Snippet for Applying Girvan :

```
# Iteration 4
print("Iteration 4:")
#Step1:
print("Initial Value of Connected Componenets:"+str(nx.number_connected_components(G)))
b = max(nx.edge_betweenness_centrality(G));
print("Value of highest edge betweenness: "+str(b))

#Step2:
print("Highest Edge Betweenness Removed:")
G.remove_edge(*b)

#Step3:
print("Value of Connected Componenets after Removal of Edge:"+str(nx.number_connected_components(G)))
nx.draw(G,pos=None, node_color="r",edge_color="b", node_size=1)

#All the Communities
S = [G.subgraph(c).copy() for c in nx.connected_components(G)]
for i in S:
    print(i.nodes())
    print("************************************************")
```

Output:

```
Iteration 4:
Initial Value of Connected Componenets:245
Value of highest edge betweenness: (8192, 4943)
Highest Edge Betweenness Removed:
Value of Connected Componenets after Removal of Edge:246
[2972, 1799]
**************************************************
[1859, 11, 722, 54, 439, 2294]
**************************************************
[122, 3180, 2958]
**************************************************
[2565, 936, 8, 4713, 7373, 538, 7386, 29, 4031]
**************************************************
[416, 226, 223]
**************************************************
[5176, 4276]
**************************************************
[72, 813, 3284, 2264, 5695]
**************************************************
```

Graph:



ITERATION 5:

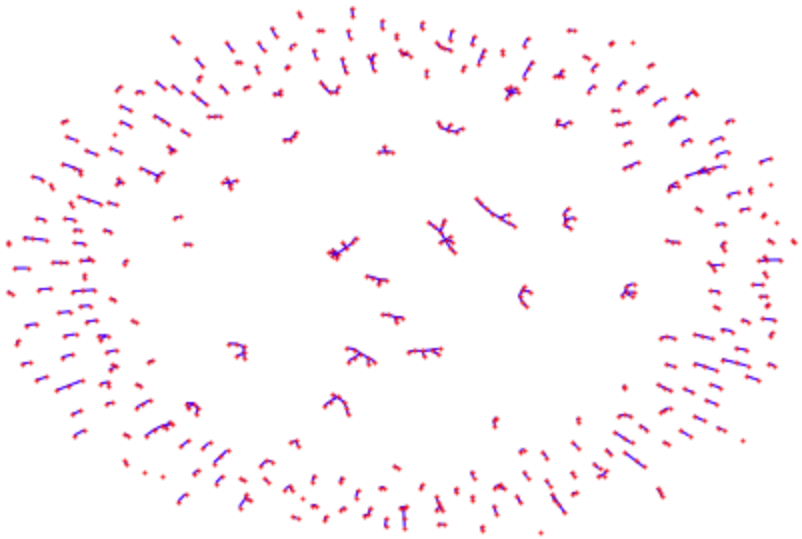
Code Snippet for Applying Girvan :

```
# Iteration 5
print("Iteration 5:")
#Step1:
print("Initial Value of Connected Componenets:"+str(nx.number_connected_components(G)))
b = max(nx.edge_betweenness_centrality(G));
print("Value of highest edge betweenness: "+str(b))

#Step2:
print("Highest Edge Betweenness Removed:")
G.remove_edge(*b)

#Step3:
print("Value of Connected Componenets after Removal of Edge:"+str(nx.number_connected_components(G)))
nx.draw(G,pos=None, node_color="r",edge_color="b", node_size=1)

#All the Communities
S = [G.subgraph(c).copy() for c in nx.connected_components(G)]
for i in S:
    print(i.nodes())
    print("**************************************************")
```

Output:

```
Iteration 5:
Initial Value of Connected Componenets:246
Value of highest edge betweenness: (8141, 6443)
Highest Edge Betweenness Removed:
Value of Connected Componenets after Removal of Edge:247
[2972, 1799]
**************************************************
[1859, 11, 722, 54, 439, 2294]
**************************************************
[122, 3180, 2958]
**************************************************
[2565, 936, 8, 4713, 7373, 538, 7386, 29, 4031]
**************************************************
[416, 226, 223]
**************************************************
[5176, 4276]
**************************************************
[72, 813, 3284, 2264, 5695]
**************************************************
```

Graph:



Part b.2) Ravasaz Algorithm Communities after each step till 5 Rounds

Ravasaz is an Agglomerative Algorithm.

Step 1: Calculation of Similarity Matrix
Step 2: Group Similarity
Step 3: Hierarchical Clustering
Step 4: Build Dendrogram

Group Similarity Criteria

After joining the most similar nodes, clusters need to be compared with the remaining elements of the network (nodes/clusters). Three clustering approaches can be used:

1. Single Linkage: similarity between two groups is equal to the similarity between the most similar elements.

2. Complete Linkage: analogous to the previous measure but using as reference the most dissimilar nodes from each cluster.

3. Average Linkage: considers the average distance of every possible pair combination in the 2 clusters.

Hierarchical Clustering Procedure

Having defined a similarity matrix and a similarity criterion to compare clusters, the following steps are executed:

1. Assign a similarity value to every pair of nodes in the network;
2. Identify the most similar community/node pair and join both. The similarity matrix is updated based on group similarity criteria;
3. The second step is repeated until all nodes are in the same community.

Dendrogram Cut

At the end of the execution, a single tree joining all nodes is obtained — dendrogram. Although it is possible to identify the most similar nodes, it does not return the best partition of the network. In fact, the dendrogram can be cut in one out of several levels. To solve this, modularity is calculated for each partition and the one with the highest value is chosen.

Combining the four steps, the complexity of the algorithm is estimated:

- Step 1: similarity between every pair of nodes is calculated. Complexity should be , being the number of elements in the network;
- Step 2: each community is compared against the others. This requires calculations;
- Steps 3 and 4: using a convenient structure to represent data, in the worst-case scenario, the dendrogram can be built in steps.

```python
import scipy.cluster.hierarchy as hierarchy
from collections import defaultdict
import numpy as np
def ravasaz(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(v)]=0

    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
```

DATASET 1: Facebook

```python
# # DATSET 1 : FaceBook
import random
s = random.sample(G1.edges(),100)
G = nx.Graph()
G.add_edges_from(s)
print(nx.info(G))
```

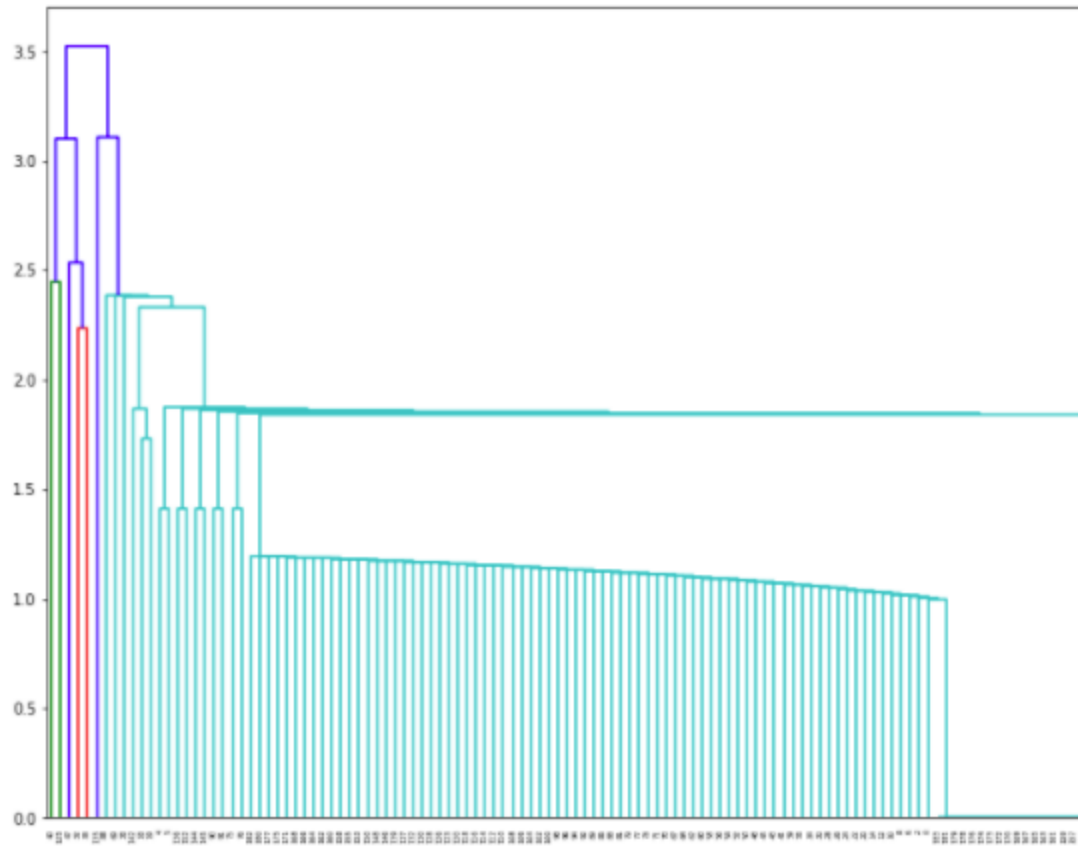
```
Name:
Type: Graph
Number of nodes: 188
Number of edges: 100
Average degree:   1.0638
```

```
Z, membership, partition = ravasaz(G,t=1.15)
print(partition.items())
```

```
dict_items([(3, [NodeView((2629, 2351, 983, 1845, 2754, 2919, 2118, 2495, 2134, 1987, 2327, 2516, 1997, 2340, 2046, 2104,
2584, 2611, 2218, 68, 175, 1985, 2506, 372, 543, 2115, 2142, 698, 871, 2871, 2875, 2342, 2644, 107, 966, 2336, 2572, 248
4, 2625, 1535, 1811, 2302, 2491, 2011, 2627, 2317, 2402, 1427, 1545, 1573, 1628, 1912, 2271, 2018, 2494, 3555, 3803, 143
1, 1737, 1014, 1231, 2053, 2332, 1629, 1733, 86, 227, 1680, 1966, 1989, 1025, 1844, 578, 659, 2669, 2874, 1290, 1730, 135
2, 1861, 1209, 1528, 428, 517, 2282, 2555, 2073, 2369, 2308, 2254, 2598, 2370, 2430, 2224, 2616, 1472, 1482, 2460, 2624,
1361, 1391, 2960, 3399, 2247, 1920, 726, 815, 925, 1689, 2030, 2140, 1158, 1626, 1674, 1082, 503, 539, 703, 828, 2949, 29
97, 2925, 3040, 978, 1491, 640, 675, 2220, 2526, 2075, 2222, 1288, 1542, 2869, 2849, 897, 1458, 627, 643, 455, 500, 3508,
3850, 1971, 2276, 2309, 2554, 636, 653, 3509, 3593, 3943, 3800, 1211, 1597, 2428, 2045, 2615, 1376, 1810, 688, 800, 995,
1238, 1603, 1800, 2744, 2762, 2068, 2144, 1813, 718, 834, 1429, 1502, 2992, 3397, 3283, 3314, 1066, 1316, 1584, 1826, 109
9, 1595, 2384, 2052, 2138, 2567, 1132, 1289)), NodeView((2629, 2351, 983, 1845, 2754, 2919, 2118, 2495, 2134, 1987, 2327,
2516, 1997, 2340, 2046, 2104, 2584, 2611, 2218, 68, 175, 1985, 2506, 372, 543, 2115, 2142, 698, 871, 2871, 2875, 2342, 26
44, 107, 966, 2336, 2572, 2484, 2625, 1535, 1811, 2302, 2491, 2011, 2627, 2317, 2402, 1427, 1545, 1573, 1628, 1912, 2271,
2018, 2494, 3555, 3803, 1431, 1737, 1014, 1231, 2053, 2332, 1629, 1733, 86, 227, 1680, 1966, 1989, 1025, 1844, 578, 659,
2669, 2874, 1290, 1730, 1352, 1861, 1209, 1528, 428, 517, 2282, 2555, 2073, 2369, 2308, 2254, 2598, 2370, 2430, 2224, 261
6, 1472, 1482, 2460, 2624, 1361, 1391, 2960, 3399, 2247, 1920, 726, 815, 925, 1689, 2030, 2140, 1158, 1626, 1674, 1082, 5
03, 539, 703, 828, 2949, 2997, 2925, 3040, 978, 1491, 640, 675, 2220, 2526, 2075, 2222, 1288, 1542, 2869, 2849, 897, 145
8, 627, 643, 455, 500, 3508, 3850, 1971, 2276, 2309, 2554, 636, 653, 3509, 3593, 3943, 3800, 1211, 1597, 2428, 2045, 261
5, 1376, 1810, 688, 800, 995, 1238, 1603, 1800, 2744, 2762, 2068, 2144, 1813, 718, 834, 1429, 1502, 2992, 3397, 3283, 331
4, 1066, 1316, 1584, 1826, 1099, 1595, 2384, 2052, 2138, 2567, 1132, 1289)), NodeView((2629, 2351, 983, 1845, 2754, 2919,
2118, 2495, 2134, 1987, 2327, 2516, 1997, 2340, 2046, 2104, 2584, 2611, 2218, 68, 175, 1985, 2506, 372, 543, 2115, 2142,
```

Dendrogram:

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



DATASET 2: Wiki

```
# # DATSET 2 : Wiki
import random
s = random.sample(G2.edges(),100)
G = nx.Graph()
G.add_edges_from(s)
print(nx.info(G))
```

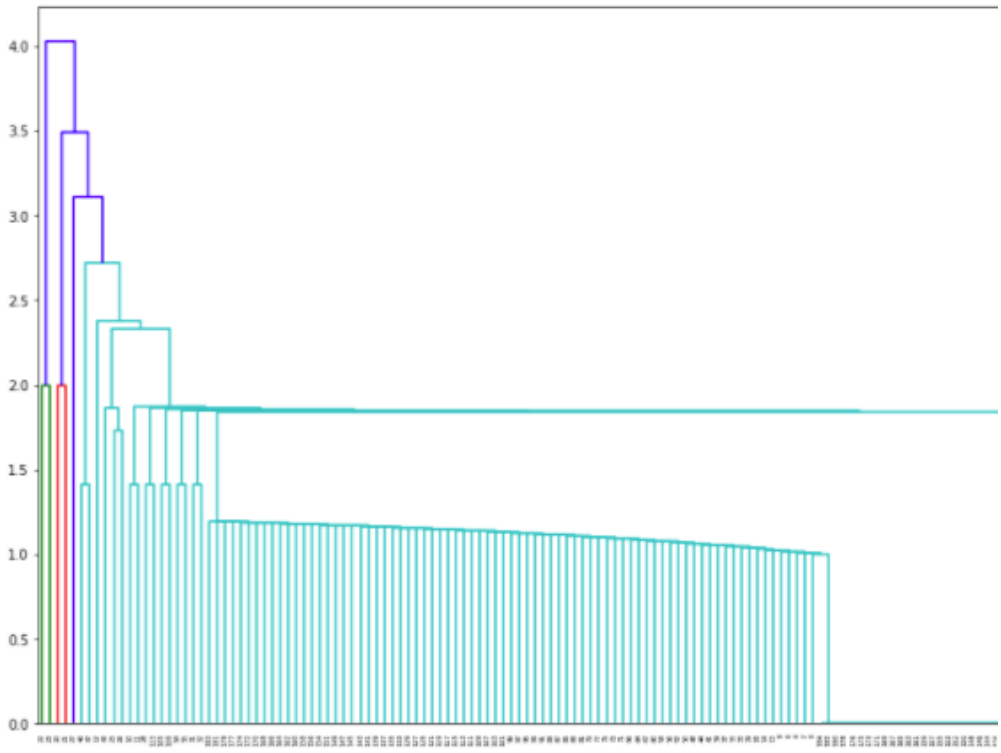
```
Name:
Type: Graph
Number of nodes: 184
Number of edges: 100
Average degree:   1.0870
```

```
Z, membership, partition = ravasaz(G,t=1.15)
print(partition.items())
```

```
98, 664, 432, 882, 620, 2144, 3461, 5563, 6320, 4875, 6114, 6715, 5189, 3456, 5008, 1151, 2592, 813, 1239, 4666, 5850, 24
3, 245, 5100, 4948, 3520, 2740, 2062, 2339, 749, 6347, 1253, 1525, 3020, 3288, 2696, 2118, 8, 132, 407, 1566, 3253, 1098,
4263, 2972, 707, 1020, 3144, 3850, 319, 2371, 1473, 2966, 3479, 852, 4999, 1487, 4588, 2326, 2016, 865, 3787, 737, 1297,
2499, 600, 6833, 3238, 6941, 28, 80, 993, 1980, 5404, 6094, 1549, 1646, 3021, 2654, 3056, 3752, 7598, 3800, 3978, 5887, 7
449, 5064, 4476, 4037, 4428, 465, 1307, 3276, 879, 4713, 4967, 4231, 4138, 1514, 1306, 878, 94, 56, 8294, 3918, 840, 580
0, 2323, 1820)), NodeView((6665, 6032, 2822, 3402, 7047, 5466, 1973, 1166, 3646, 3777, 5775, 5771, 643, 838, 1501, 1788,
1374, 2763, 5123, 7691, 3847, 2967, 1550, 2129, 1752, 2565, 3660, 6464, 457, 2297, 546, 11, 5144, 3028, 3127, 7052, 3459,
4632, 4798, 2485, 2375, 1111, 1621, 3455, 2251, 2948, 2544, 3541, 1734, 7961, 2811, 3453, 4310, 4988, 1291, 2474, 6720, 6
082, 23, 972, 1654, 1653, 1428, 1466, 7225, 1037, 6006, 5172, 2736, 7115, 2940, 1966, 633, 739, 513, 5790, 4598, 664, 43
2, 882, 620, 2144, 3461, 5563, 6320, 4875, 6114, 6715, 5189, 3456, 5008, 1151, 2592, 813, 1239, 4666, 5850, 243, 245, 510
0, 4948, 3520, 2740, 2062, 2339, 749, 6347, 1253, 1525, 3020, 3288, 2696, 2118, 8, 132, 407, 1566, 3253, 1098, 4263, 297
2, 707, 1020, 3144, 3850, 319, 2371, 1473, 2966, 3479, 852, 4999, 1487, 4588, 2326, 2016, 865, 3787, 737, 1297, 2499, 60
0, 6833, 3238, 6941, 28, 80, 993, 1980, 5404, 6094, 1549, 1646, 3021, 2654, 3056, 3752, 7598, 3800, 3978, 5887, 7449, 506
4, 4476, 4037, 4428, 465, 1307, 3276, 879, 4713, 4967, 4231, 4138, 1514, 1306, 878, 94, 56, 8294, 3918, 840, 5800, 2323,
1820)), NodeView((6665, 6032, 2822, 3402, 7047, 5466, 1973, 1166, 3646, 3777, 5775, 5771, 643, 838, 1501, 1788, 1374, 276
3, 5123, 7691, 3847, 2967, 1550, 2129, 1752, 2565, 3660, 6464, 457, 2297, 546, 11, 5144, 3028, 3127, 7052, 3459, 4632, 47
98, 2485, 2375, 1111, 1621, 3455, 2251, 2948, 2544, 3541, 1734, 7961, 2811, 3453, 4310, 4988, 1291, 2474, 6720, 6082, 23,
972, 1654, 1653, 1428, 1466, 7225, 1037, 6006, 5172, 2736, 7115, 2940, 1966, 633, 739, 513, 5790, 4598, 664, 432, 882, 62
0, 2144, 3461, 5563, 6320, 4875, 6114, 6715, 5189, 3456, 5008, 1151, 2592, 813, 1239, 4666, 5850, 243, 245, 5100, 4948, 3
520, 2740, 2062, 2339, 749, 6347, 1253, 1525, 3020, 3288, 2696, 2118, 8, 132, 407, 1566, 3253, 1098, 4263, 2972, 707, 102
```

Dendrogram:

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



Problem Statement 2:

a) Create a scale-free network using appropriate Python package (find out!).
b)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.

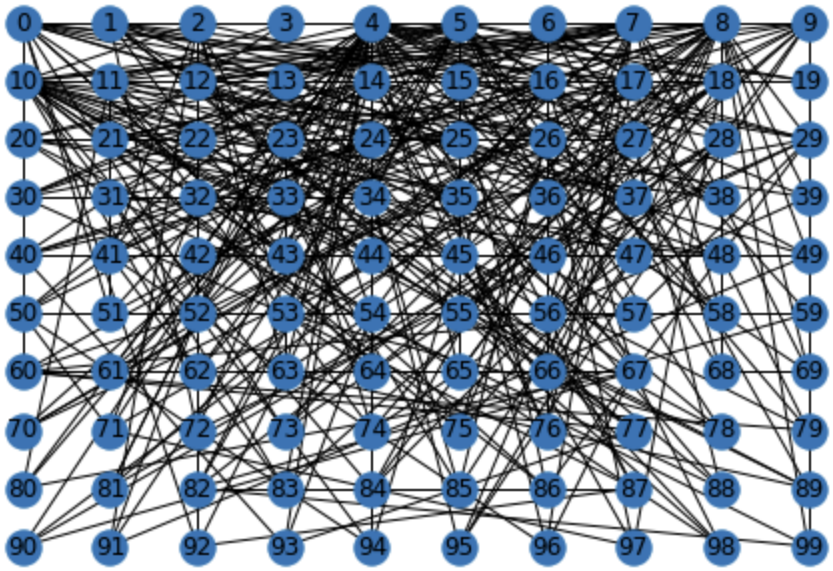
Part 1) Albert Barabasi Model

IMPLEMENTATION USING PYTHON PACKAGE:

Implemneting the Scale free Network using Albert Barabasi Model

```
import networkx as nx
import matplotlib.pyplot as plt

n = 100  # Number of nodes
m = 4  # Number of initial links
seed = 100
G = nx.barabasi_albert_graph(n, m, seed)

ncols = 10
pos = {i : (i % ncols, (n-i-1) // ncols) for i in G.nodes()}
nx.draw(G, pos, with_labels=True)
plt.show()
```

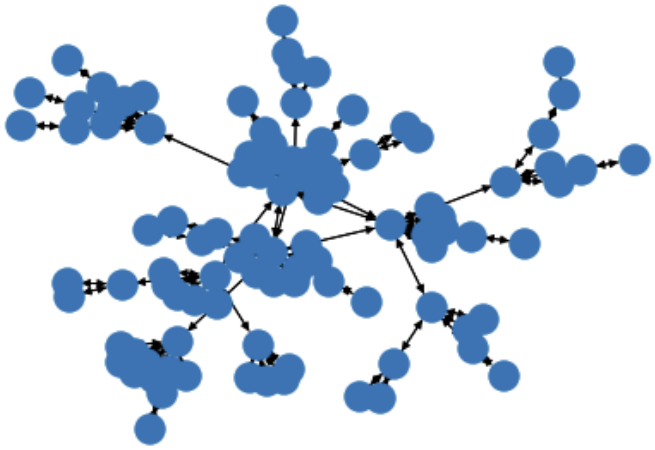
For Applying ICM we need Directed and Weighted Graph



Converting to Directed Graph

```
# change to directed graph
G = G.to_directed()
# Check
if G.is_directed():
   print("Yes, Graph is Now Directed")
print(nx.info(G))
nx.draw(G)
```

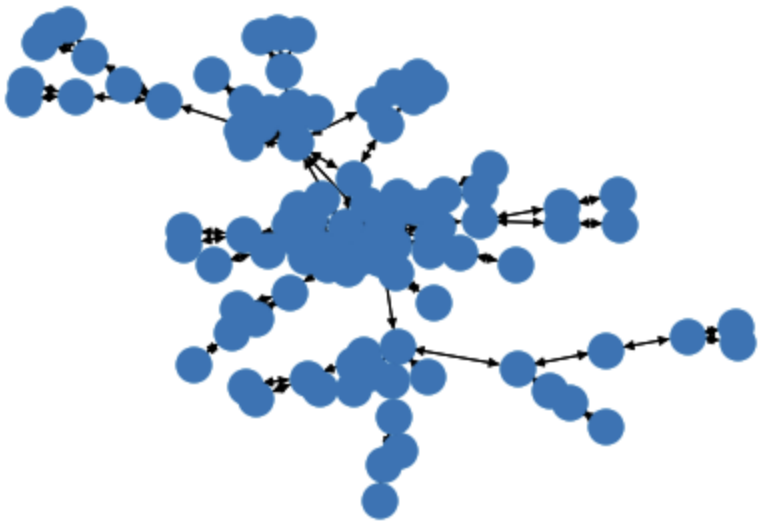
```
Yes, Graph is Now Directed
Name:
Type: DiGraph
Number of nodes: 100
Number of edges: 204
Average in degree:   2.0400
Average out degree:   2.0400
```



IMPLEMENTATION From SCRATCH and Weighted Scale free Network:

```
# Get parameters
init_nodes = 4
final_nodes = 100
m_parameter = 1

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())))
```

```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():
            add_edge()
        else:
            G.add_edge(new_node, random_proba_node)
            print("Edge added: {} {}".format(new_node + 1, random_proba_node))
```



Part b)

APPLYING ICM

Function for calculating max number of steps to get to the max number nodes using ICM. Iteration will be 5

```
def information_cascade(Networksx_Graph,times):
 mx=0
 for i in range(times):
    ans=[]
    curr=[]
    #print(i)
    if(i not in ans):
        ans.append(i)
    targets=[n for n in G.neighbors(i)]
    #print(targets)
    for k in targets:
        if(np.random.random()<G[i][k]['weight'] and k not in ans):
            ans.append(k)
            curr.append(k)
    for m in curr:
        targets=[n for n in G.neighbors(m)]
        for k in targets:
            if(k not in ans and k not in curr and np.random.random()<G[m][k]['weight']):
                ans.append(k)
                curr.append(k)

    mx=max(len(ans),mx)
    print("initial activated node:",i,end='\n')
    print("activated nodes:",ans,end='\n')
    print("count of activated nodes",len(ans),end='\n')
    return mx
```

Calling the Function:

```
print(information_cascade(G,5))
```

```
7
```

IMPLEMENTING IC MOdel:

```
Return the active nodes of each diffusion step by the independent cascade model
  Parameters
  ----------
  G : graph
    A NetworkX graph
  seeds : list of nodes
    The seed nodes for diffusion
  steps: integer
    The number of steps to diffuse.  If steps <= 0, the diffusion runs until
    no more nodes can be activated.  If steps > 0, the diffusion runs for at
    most "steps" rounds
  Returns
  -------
  layer_i_nodes : list of list of activated nodes
    layer_i_nodes[0]: the seeds
    layer_i_nodes[k]: the nodes activated at the kth diffusion step
  Notes
  -----
  When node v in G becomes active, it has a *single* chance of activating
  each currently inactive neighbor w with probability p_{vw}
```

```
r = 5 #number of cascades to be started from each node
cascades = [] #an empty list for storing cascades


for i in range(n): #generate r cascades for each node and append to the cascade list
    for j in range (r):
        cascades.append(independent_cascade(G, [i]))

print(cascades)
```

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


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
```

```python
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 independent_cascade(DG, seeds):
    A = copy.deepcopy(seeds)  # prevent side effect
    return _diffuse_all(DG, A)
```

Github Repository:

https://github.com/Raghavlakhotia/social_media_analysis_

References:

- https://networkx.org/documentation/stable/reference/algorithms/centrality.html
- https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.components.strongly_connected_components.html#networkx.algorithms.components.strongly_connected_components