

Social Network Analysis

**Project Report by
Team PEAK**

Team Members:

Paridhi Reddy (18UCS042)

Kushagra Chaturvedy (18UCS229)

Eshaan Rathi (18UCC089)

Aman Jain (18DCS009)

May 2021

Course Instructor

Dr. Sakthi Balan


Department of Computer Science Engineering
The LNM Institute of Information Technology, Jaipur

DATASET DESCRIPTION

Dataset 1: Caenorhabditis elegans (metabolic)

The statistics for the dataset is given below:

Size	$n=453$
Volume	$m=2025$
Loop Count	$l=0$
Wedge Count	$s=79173$
Claw Count	$z=3352172$
Cross Count	$x=153983040$
Triangle Count	$t=3284$
Square Count	$q=50289$
Maximum Degree	$d_{\max}=237$
Average Degree	$d=8.94040$
Fill	$p=0.0197796$
Diameter	$\delta=7$
50- percentile effective diameter	$\delta_{0.5}=2.14350$
90- percentile effective diameter	$\delta_{0.9}=3.16935$
Median distance	$\delta_M=3$

Mean distance	$\delta_m=2.67627$
Gini Coefficient	$G=0.494803$
Balanced Inequality Ration	$P=0.321481$
Relative edge distribution entropy	$H_{er}=0.898264$
Power law exponent	$Y=1.56569$
Tail power-law exponent	$Y_t=2.63100$
Tail power-law exponent with p	$Y_3=2.63100$
p-value	$p=0.0470000$
Spectral norm	$\alpha=26.3085$
Algebraic connectivity 	$a=0.258000$
Non-bipartivity	$b_A=0.429332$
Normalized non-bipartivity	$b_N=0.200551$
Algebraic non-bipartivity	$X=0.293557$
Spectral bipartite frustration	$b_K=0.00820872$
Controllability	$C=7$
Relative controllability	$C_r=0.0154525$

Graph (visual) for Dataset 1:

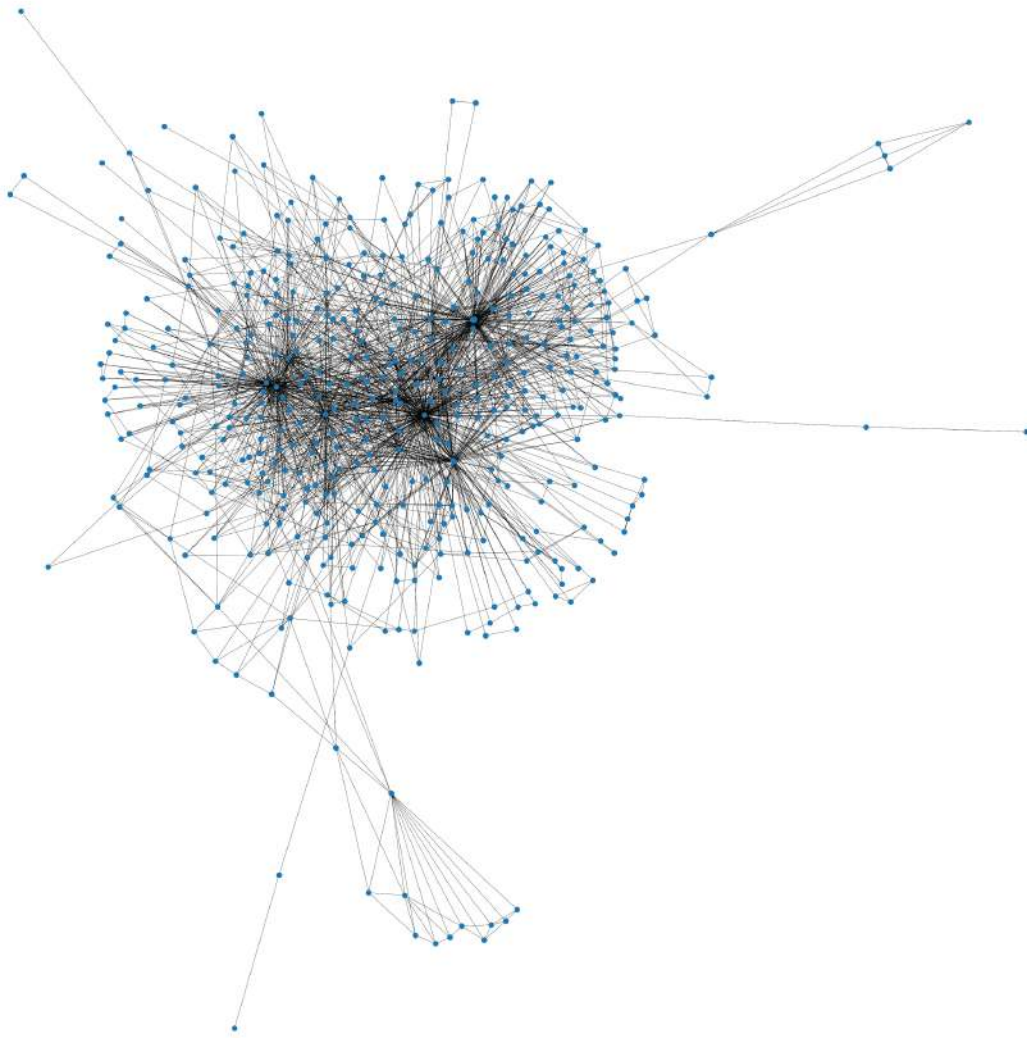


Figure 1.1 : Caenorhabditis Network

Dataset 2: Jazz Musicians

The statistics for the dataset is given below:

Size	$n=198$
Volume	$m=2742$
Loop Count	$l=0$
Wedge Count	$s=103212$
Claw Count	$z=1583352$
Cross Count	$x=21666963$
Triangle Count	$t=17899$
Square Count	$q=406441$
Maximum Degree	$d_{\max}=100$
Average Degree	$d=27.6970$
Fill	$p=0.140594$
Diameter	$\delta=6$
50- percentile effective diameter	$\delta_{0.5}=1.64700$
90- percentile effective diameter	$\delta_{0.9}=2.79355$
Median distance	$\delta_M=2$
Mean distance	$\delta_m=2.20604$
Gini Coefficient	$G=0.345989$

Balanced Inequality Ration	$P=0.373450$
Relative edge distribution entropy	$H_{er}=0.961549$
Power law exponent	$Y=1.32928$
Tail power-law exponent	$Y_t=5.27100$
Tail power-law exponent with p	$Y_3=5.27100$
p-value	$p=0.610000$
Spectral norm	$\alpha=40.0274$
Algebraic connectivity	$a=0.571994$
Non-bipartivity	$b_A=0.782583$
Normalized non-bipartivity	$b_N=0.460437$
Algebraic non-bipartivity	$X=0.692773$
Spectral bipartite frustration	$b_K=0.00625314$
Controllability	$C=1$
Relative controllability	$C_r=0.00505051$

Graph (visual) for Dataset 2:

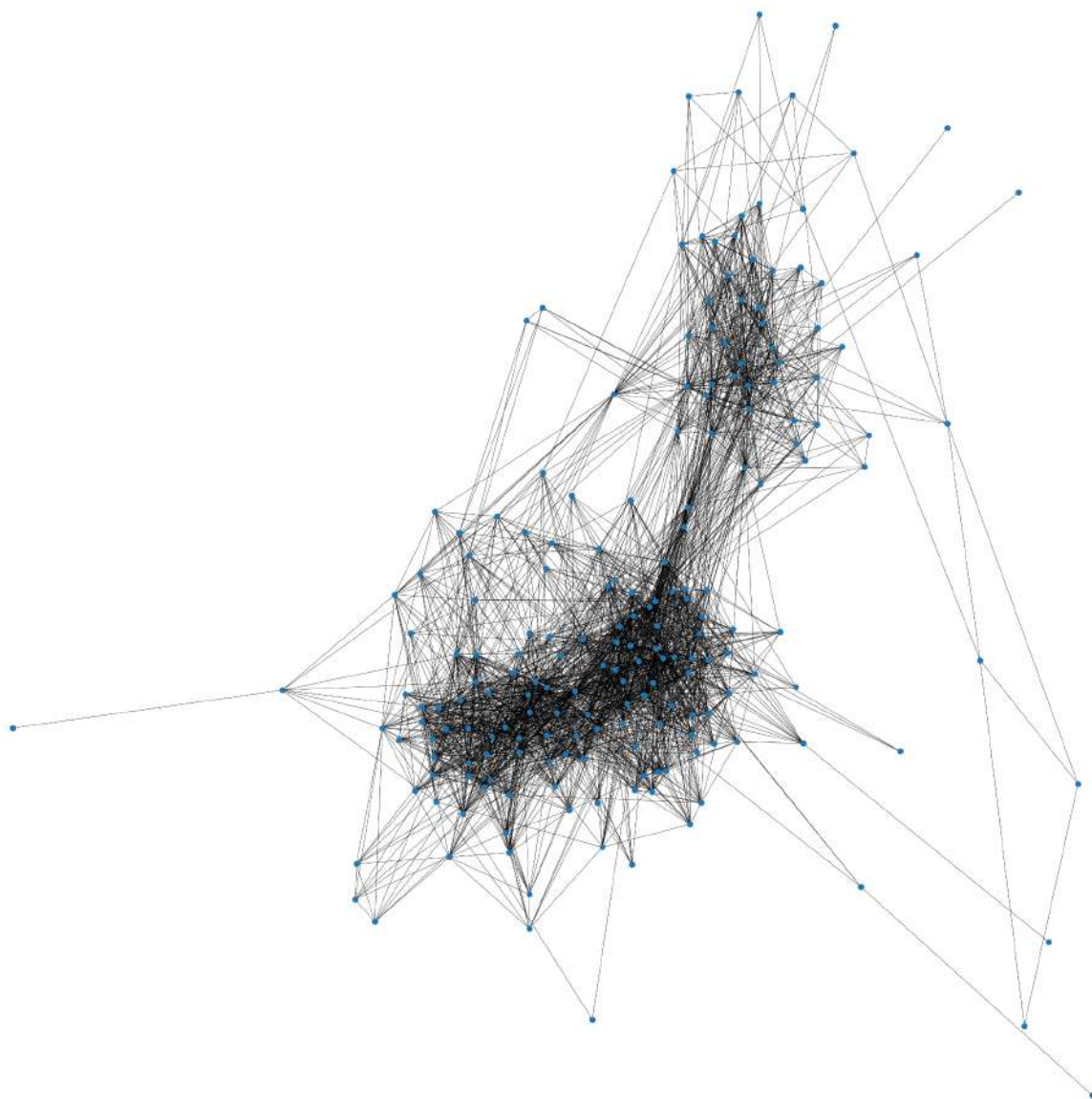


Figure 1.2 : Jazz Network

NETWORK DESCRIPTION

Dataset 1: Caenorhabditis elegans (metabolic)

- This metabolic network represents the metabolic pathways in 43 organisms.
- The nodes represent the substrate and the edge between two nodes represents the existence of a metabolic reaction between them.
- The network is unipartite and edges are undirected.
- The edges are not weighted.
- The network has no multiple edges.
- The given network has no loops.

Dataset 2: Jazz Musicians

- This network is a collaboration network between Jazz musicians.
- The nodes represent the musicians and the edge between two nodes represents that the two musicians have played together in a band.
- The network is unipartite and edges are undirected.
- The edges are not weighted.
- The network has no multiple edges.
- The given network has no loops.

PROBLEM 1:

Find all centrality measures, clustering coefficients (both local and global), and reciprocity and transitivity that we have studied in the class using appropriate algorithms for the selected datasets.

→ [Code Notebook for Dataset 1](#)

→ [Code Notebook for Dataset 2](#)

CENTRALITY MEASURES STUDIED

Degree Centrality:

Degree Centrality measures the importance of a node concerning its degree. It is a good centrality measure when the goal is to find connected, popular individuals or those who are probably going to hold more information in a network.

- [Degree Centrality of Dataset 1](#)
- [Degree Centrality of Dataset 2](#)

Eigenvector Centrality:

Eigenvector centrality generalizes degree centrality. The general idea is: Having more important friends (incoming edge in case of a directed graph) also makes me important. Eigenvector centrality is a good measure for understanding human social networks.

- [Eigenvector Centrality of Dataset 1](#)
- [Eigenvector Centrality of Dataset 2](#)

Katz Centrality:

The eigenvector centrality proves to be a bad centrality measure in the special case of a directed acyclic graph where the centrality becomes zero even though the node may have many edges. We add a bias term β to all the nodes which is added to the centrality measure of all nodes irrespective of how they are placed in a network.

- [Katz Centrality of Dataset 1](#)
- [Katz Centrality of Dataset 2](#)

Pagerank Centrality:

Pagerank centrality measure is a well-known and widely used measure. The major issue with Katz Centrality is that once a node becomes central, it passes its centrality along all of its outlinks, but this is not ideal in the real world sometimes as not everyone known by a popular individual is also popular. This centrality measure can be very helpful to study citations and authority as it takes the edge direction and connection weight into account.

- [Pagerank Centrality of Dataset 1](#)
- [Pagerank Centrality of Dataset 2](#)

Betweenness Centrality:

Till now all the centrality measures we discussed, talked about node centrality with respect to the structure of the network but betweenness centrality measures the importance of a node in terms of connecting other nodes via path connections. This measure can be very helpful to study which individuals influence flow in a system.

- [Betweenness Centrality of Dataset 1](#)
- [Betweenness Centrality of Dataset 2](#)

Closeness Centrality:

According to this centrality measure, important nodes are the ones from where we can reach other nodes in the network more quickly, that is, the nodes with the smallest average path length to other nodes.

- [Closeness Centrality of Dataset 1](#)
- [Closeness Centrality of Dataset 2](#)

CLUSTERING COEFFICIENT OF DATASETS

Clustering Coefficient is a measure of transitivity. It also gives us an idea about whether the network is sparse or dense.

Local Clustering Coefficient:

Local clustering Coefficient measures a given nodes' neighbors' level of connectivity. It varies in the range between 0 and 1. It represents transitivity for a given node.

- [Local Clustering Coefficient of Dataset 1](#)
- [Local Clustering Coefficient of Dataset 2](#)

Global Clustering Coefficient:

It measures the transitivity of the entire network.

- *Global Clustering Coefficient of Dataset 1:*
0.12443636088060324
- *Global Clustering Coefficient of dataset 2:*
0.5202592721776538

RELEVANT CENTRALITY MEASURES

Dataset 1: Caenorhabditis elegans (metabolic)

- **Degree Centrality:** As the dataset contains metabolic reactions in between multiple substrates, degree centrality would help us to understand the reactive nature of the substrate. Immediate neighbors are of more importance rather than neighbors of neighbors.

Dataset 2: Jazz Musicians

- **Degree Centrality:** In collaborative networks such as this Jazz Network, degree centrality is relevant as the nodes with high degree centrality will help us find famous and popular Jazz musicians that have collaborated with the most number of musicians.
- **Closeness Centrality:** The closeness centrality of the Jazz network tells us how close a musician is to other musicians in the network. This gives us information about the independence and autonomy of the musician.

INFERENCES

Dataset 1: Caenorhabditis elegans (metabolic)

- ➔ It is observed that the node designated as '3' has the highest Degree Centrality (0.52433). This shows it is the most reactive substrate.
- ➔ It is observed that the node designated as '3' has the highest Eigenvector Centrality ($=0.3799$). This shows that it is connected to many substrates who are themselves reactive.
- ➔ It is observed that the nodes designated as '7' and '8' have the highest Katz Centrality ($=0.2097$). This shows that these nodes have the highest influence over other nodes in the graph.
- ➔ It is observed that the node designated as '3' has the highest Pagerank Centrality ($=0.0550$). This shows that it is important because it is connected to the other important nodes.
- ➔ It is observed that the node designated as '3' has the highest Betweenness Centrality ($=0.4783$). This shows that this substrate comes in between the maximum number of metabolic pathways between any other two nodes.

- It is observed that the node designated as '3' has the highest Closeness Centrality ($=0.6541$). This shows that it is closest to every other node, i.e. has the least sum of metabolic paths to every other node.
- There are 109 nodes with local clustering coefficients equal to 1. This means that all their neighbors can react to each other.
- Transitivity determines how close the networks are to being complete. We measure it with clustering coefficients. The global clustering coefficient came out to be 0.124436, which shows that the network is more on the incomplete side, rather than complete.
- The average clustering coefficient is 0.6464, which shows that either majority of the nodes have a high clustering coefficient, or some have extremely high clustering coefficients.
- The reciprocity is found to be 0. This is to be expected as there are no closed cycles of length 2 in the network since it is undirected.

Dataset 2: Jazz Musicians

- It is observed that the node designated as '67' has the highest value of degree centrality and betweenness centrality among all nodes in the graph. This means that node '67' has the highest degree in the graph.

- Since the betweenness centrality is also highest among all nodes, this means that node 67 is well connected and lies on the shortest path of the most number of node pairs.
- The highest betweenness centrality is just 0.151. Thus we can infer that there are not many bridge-like nodes in this graph.
- Node 7 has the highest eigenvector centrality among all nodes, implying that it is connected to the most central nodes in the graph.
- Node 158 has a local clustering coefficient of 1. This means that all of the neighbors of node 158 are completely connected and have an edge between each other.
- The reciprocity is 0, which is to be expected as there are no closed cycles of length 2 in our graph as the graph is undirected.
- Node 67 has the highest closeness centrality. This means that node 67 has the shortest distance to all of the other nodes.
- The average clustering coefficient is greater than 0.5. This means that there are more nodes with high clustering coefficients or that some nodes have extremely high clustering coefficients.
- Since transitivity can be defined as the probability that the neighbors of a node are interconnected and given the transitivity of our graph is 0.52, we can say that 52% of the nodes in the graph will have interconnected neighbors.

PROBLEM 2:

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

→ [Code Notebook](#)

THEORY

If the probability (P) is equal to 0 then there will be no connected component and the value of N_g/N will be zero, that said if P is 1 the value of N_g/N will be 1. In our case, the probability P is between 0 and 1 exclusive.

The graph of N_g/N v.s $\langle k \rangle$ is divided into 4 regions:

→ *Subcritical Regime:*

- ◆ $0 \leq \langle k \rangle < 1$: There are small tree-like components when $\langle k \rangle$ is greater than 0.

→ *Critical Point:*

- ◆ $k = 1$: This separates the regime when there is not yet a giant component to where there is one.

→ *Supercritical regime:*

- ◆ $1 < \langle k \rangle < \ln(N)$: The N_g/N value keeps increasing.

→ *Connected Regime:*

- ◆ $\langle k \rangle > \ln(N)$. The entire network is connected.

```
import networkx as nx
from networkx.generators.degree_seq import expected_degree_graph
from matplotlib import pyplot as plt
```

The imported libraries are:

- **NetworkX:** NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.
- **Matplotlib:** Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

```
N=1000 # number of nodes
k=0
degree=[] #list of degree
NgbyN=[] # list of Ng / N for corresponding k
```

- We are taking the number of nodes, $N = 1000$ and the initial average degree, $k = 0$.
- We are using two lists to store the average degree k , labeled *degree* and relative size of the giant component, labeled *NgbyN*.

```
while k <= 5:
    degree.append(round(k,1))
    w = [k for i in range(N)]
    G = expected_degree_graph(w)
    connected_components = (G.subgraph(c) for c in nx.connected_components(G))
    giant = max(connected_components, key=len)
    Ng = len(giant)
    ngbyn = Ng/N
    print("Degree = " + str(round(k,1)) + ", Ng = " + str(Ng) + ", Ng/N = " + str(ngbyn))
    NgbyN.append(ngbyn)
    k += .1
```

- The while loop increments k (average degree) at an interval of 0.1 from 0 to 5.
- Inside the while loop, we append the same degree k to all nodes and pass it through the function *expected_degree_graph*, which takes a list as a parameter; we pass the degree of all nodes in this case. It gives a network G as output which assigns edges between nodes u and v based on the normalized product of their degrees.
- For finding the connected components, we use the inbuilt function *connected_components*, which gives us all the connected components in the network G . Here, *giant* is the largest of all the connected components wrt the size of connected components.
- Ng is the length of *giant*, that is, the number of nodes in the giant component. With this we find Ng/N .
- Each value of Ng/N is appended to the list *NgbyN*.

```
plt.plot(degree, NgbyN)  
plt.xlabel("Average degree <k>")  
plt.ylabel("Ng / N")  
plt.xlim(0, 5)  
plt.show()
```

The plot of Ng/N vs the average degree $\langle k \rangle$ of the network:

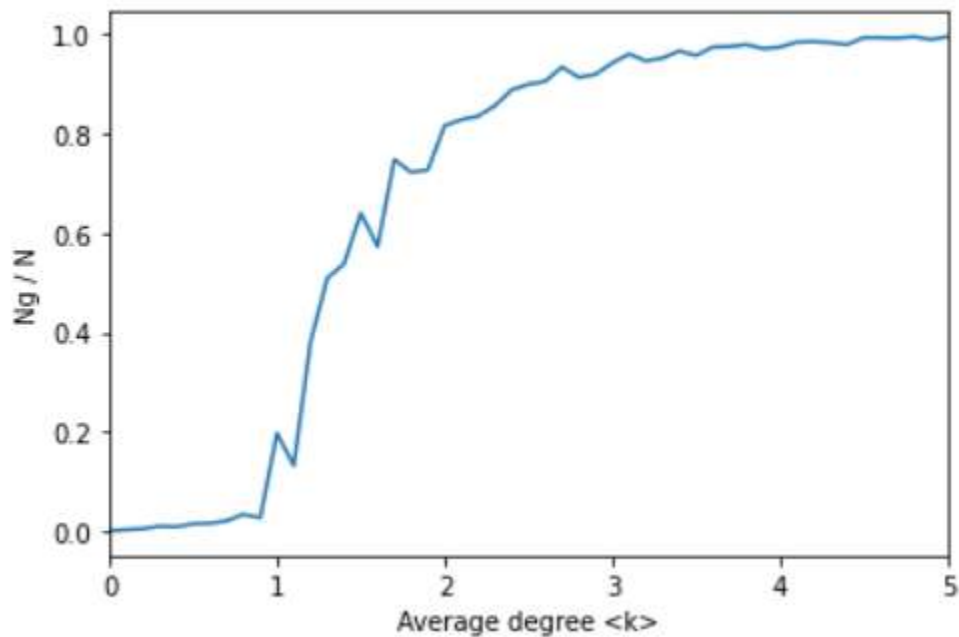


Figure 2.1 : Graph Ng/N v.s $\langle k \rangle$

INFERENCES

- From the graph (*Figure 2.1*), we can infer that the value of N_g/N remains close to zero till $\langle k \rangle$ is 1, following that we need at least one link per node to see a giant component. Also, N_g/N increases linearly in the subcritical region. This is due to the existence of small connected components having a tree-like structure.
- The graph shows a sudden rise in the relative size of the giant component of the network, that is, N_g/N at around $k = 1$, and keeps on increasing and reaches saturation around $k = 4.5$.
- *Figure 2.1* visualizes the expected behavior at the critical point and in the subcritical region.
- The network exists in the supercritical regime as it reaches the fully connected state in the supercritical region ($\langle k \rangle < \ln(N) \approx 7$).
- The value of N here is small (1000) for our computation. To generalize it for larger datasets and real-life datasets that have a billion nodes, we can say that, even when the average degree is 2, that is, a node is connected to only two other nodes at random, the giant component occupies 75% of the network (inference from *Figure 2.1*).

PROBLEM 3:

Find the giant component G in the network.

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 for the selected datasets.

→ [Code Notebook](#)

Dataset 1: *Caenorhabditis elegans* (metabolic)

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

The required libraries are imported.

- NetworkX
- Matplotlib

```
celegans_graph=nx.Graph()
f = open("/content/out.dimacs10-celegans_metabolic", "r")
lines = f.readlines()
lines.pop(0)
for line in lines:
    l = line.split()
    celegans_graph.add_edge(l[0],l[1])
```

The *Graph* method of the NetworkX library creates an empty graph with no nodes and no edges, labeled as *celegans_graph*. The dataset file is read line by line using the for loop. Inside the loop, edges are added to the network *celegans_graph* using the *add_edge* method of the NetworkX library.

```
fig=plt.figure(figsize=(50,50))  
nx.draw(celegans_graph)  
plt.show()
```

The *draw* method of the NetworkX library draws a visual representation of the network *celegans_graph*. The *show* method of the Matplotlib library displays the produced drawing of the network.

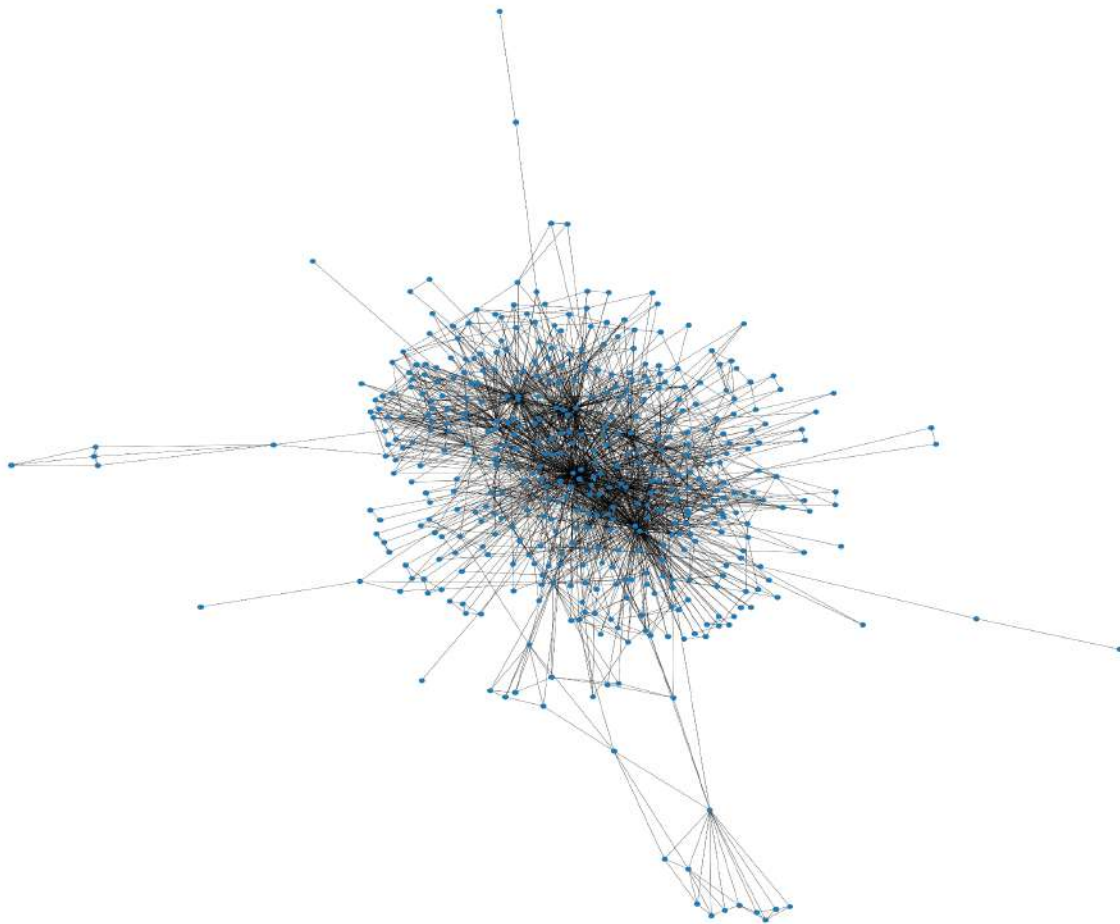


Figure 3.1 : Caenorhabditis Network

```
giant = max(nx.connected_components(celegans_graph), key=len)
Ng = len(giant)
print("The len of giant component is:" + str(len(giant)))
print("The len of celegans graph is:" + str(len(celegans_graph)))
N = len(celegans_graph)
ngbyn = Ng/N
print(ngbyn)
```

The *connected_components* method of the NetworkX library generates the connected components of the network *celegans_graph*. The *max* function gives the largest component of the network wrt the size of the connected components. *Ng* is the size of the largest connected component *giant* and *N* is the size of the network *celegans_graph*. The ratio of the size of the largest component and the size of the network is represented by *ngbyn*.

INFERENCE

- The relative size of the giant component of the *celegans_graph* network is 1; that is, the size of the giant component is equal to the size of the network *celegans_graph*, which means 453.
- We infer that the network *celegans_graph* is connected; thus, we have a path between every pair of nodes.

Dataset 2: Jazz Musicians

```
jazz_graph=nx.Graph()
with open('out.arenas-jazz','r') as f:
    lines=f.readlines()
lines.pop(0)
for line in lines:
    line=line.strip()
    lis=line.split('\t')
    if lis != '% sym unweighted':
        jazz_graph.add_edge(lis[0],lis[1])
```

The *Graph* method of the NetworkX library creates an empty graph with no nodes and no edges, labeled as *jazz_graph*. The dataset file 'out.arenas-jazz' is read line by line using the for loop and inside the loop, edges are added to the network *jazz_graph* using the *add_edge* method of the NetworkX library.

```
fig=plt.figure(figsize=(50,50))
nx.draw(jazz_graph)
plt.show()
```

The *draw* method of the NetworkX library draws a visual representation of the network *jazz_graph*. The *show* method of the Matplotlib library displays the produced drawing of the network.

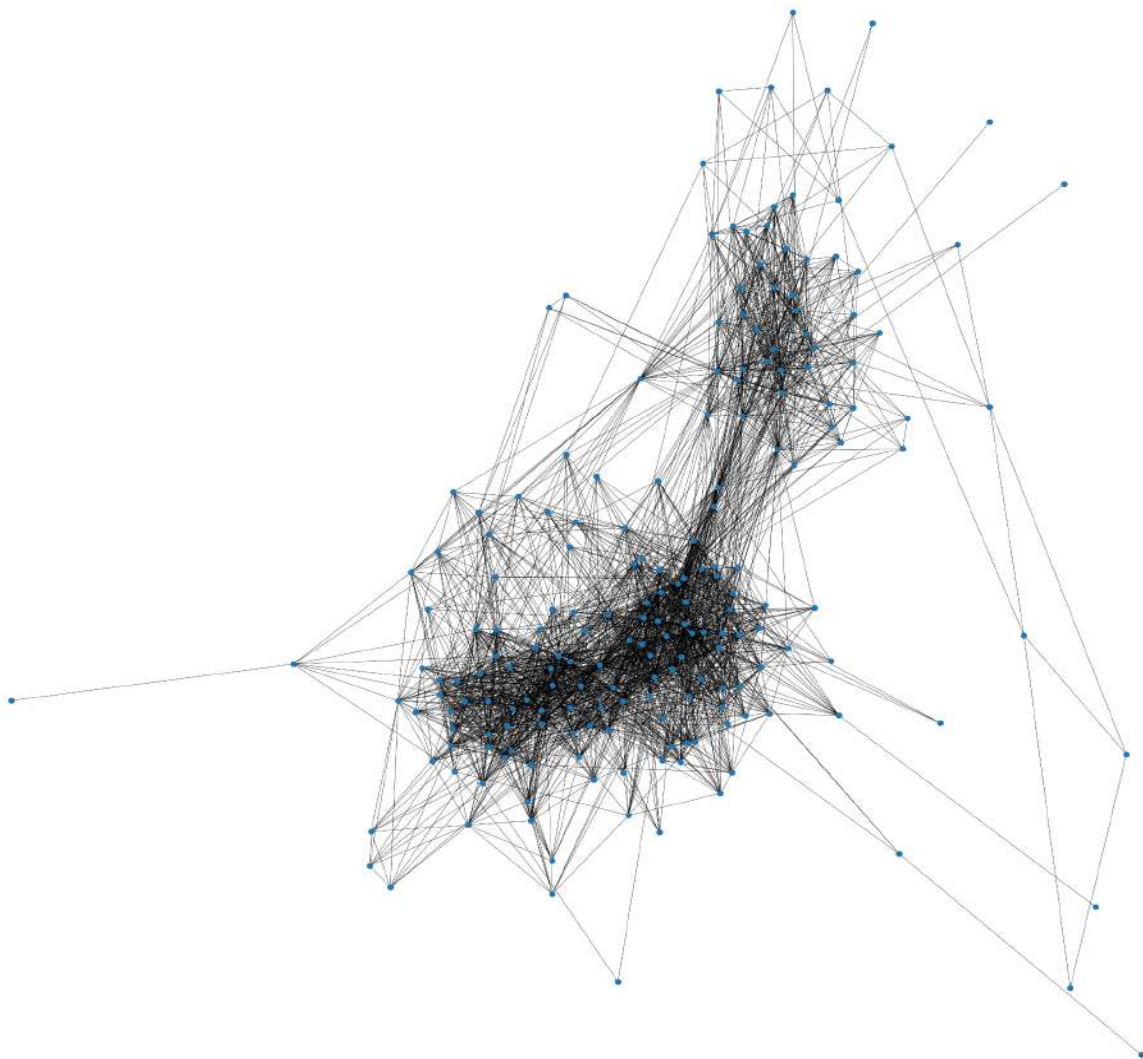


Figure 3.2 : Jazz Network


```
giant = max(nx.connected_components(jazz_graph), key=len)
Ng = len(giant)
print("The len of giant component is:" + str(len(giant)))
print("The len of Jazz Musicial graph is:" + str(len(jazz_graph)))
N = len(jazz_graph)
ngbyn = Ng/N
print(ngbyn)
```

The *connected_components* method of the NetworkX library generates the connected components of the network *jazz_graph*. The *max* function gives the largest component of the network wrt the size of the connected components. *Ng* is the size of the largest connected component *giant* and *N* is the size of the network *jazz_graph*. The ratio of the size of the largest component and the size of the network is represented by *ngbyn*.

INFERENCE

- The relative size of the giant component of the *jazz_graph* network is 1; that is, the size of the giant component is equal to the size of the network *jazz_graph*, which means 198.
- We infer that the network *jazz_graph* is connected; thus, we have a path between every pair of nodes.

PROBLEM 4:

Apply Girvan Newman algorithm and Ravasz algorithm to find the communities step by step and illustrate at 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. Looking at the output given by the two algorithms compare and contrast the two algorithms.

→ [Code Notebook: GN](#)

→ [Code Notebook: Ravasz](#)

GIRVAN NEWMAN CLUSTERING ALGORITHM

It is a divisive hierarchical clustering algorithm and assumes the entire network to be a single community initially. To split communities, a centrality measure needs to be defined. Link betweenness and random walk betweenness are the most common centrality measures used.

- **Link betweenness:** Number of shortest paths between all node pairs that run along a link. If there is a large betweenness value, that means it joins edges from two different communities.
- **Random-walk betweenness:** A pair of nodes m and n chosen at random. A walker starts at m , following each adjacent link with equal probability until it reaches n . Random-walk betweenness x_{ij} is the probability that the link $i \rightarrow j$ was crossed by a walker after averaging over all possible choices for the starting nodes m and n . If this is high, that means the walker crossed the edge many times, which means there are lesser options to go another way to reach n . So, this link acts as a bridge and thus connects different communities.

To find communities, the centrality value of all links is computed and the link(s) with maximum centrality is removed. This is repeated till no links are left.

Dataset 1: *Caenorhabditis elegans* (metabolic)

```
import networkx as nx
import math
import csv
import random as rand
import sys
```

The required libraries are imported.

- NetworkX
- Math
- CSV
- Random
- System

```
# This method reads the graph structure from the input file
def buildG(G, file_, delimiter_):
    reader = csv.reader(open(file_), delimiter=delimiter_)
    for line in reader:
        if len(line) > 2:
            if float(line[2]) != 0.0:
                #line format: u,v,w
                G.add_edge(int(line[0]),int(line[1]),weight=float(line[2]))
        else:
            #line format: u,v
            G.add_edge(int(line[0]),int(line[1]),weight=1.0)
```

This function reads the dataset file line by line using the for loop. Inside the loop, weighted edges of weight 1.0 are added to the network G using the `add_edge` method of the NetworkX library.

```
def CmtyGirvanNewmanStep(G):  
    if _DEBUG_:  
        print("Running CmtyGirvanNewmanStep method ...")  
    init_ncomp = nx.number_connected_components(G)    #number of components  
    ncomp = init_ncomp  
    while ncomp <= init_ncomp:  
        bw = nx.edge_betweenness centrality(G, weight='weight')    #edge betweenness for G  
        #find the edge with max centrality  
        max_ = max(bw.values())  
        #find the edge with the highest centrality and remove all of them if there is more than one!  
        for k, v in bw.items():  
            if float(v) == max_:  
                G.remove_edge(k[0],k[1])    #remove the central edge  
        ncomp = nx.number_connected_components(G)    #recalculate the no of components
```

- This function keeps removing edges from network G until one of the connected components of the network splits into two using a while loop.
- It does so by computing the edge betweenness of all the edges of the network G , then finding the maximum betweenness centrality value.
- Once it finds the maximum betweenness centrality, edges with the highest betweenness are removed.
- It recalculates the number of connected components labeled $ncomp$ in each iteration of the while loop and the while loop stops when the number of connected components labeled $ncomp$ attains a value more than the initial number of connected components $init_ncomp$.

```
def _GirvanNewmanGetModularity(G, deg_, m_):
    New_A = nx.adj_matrix(G)
    New_deg = {}
    New_deg = UpdateDeg(New_A, G.nodes())
    #Let's compute the Q
    comps = nx.connected_components(G)    #list of components
    print('No of communities in decomposed G: {}'.format(nx.number_connected_components(G)))
    Mod = 0    #Modularity of a given partitioning
    for c in comps:
        EWC = 0    #no of edges within a community
        RE = 0    #no of random edges
        for u in c:
            EWC += New_deg[u]
            RE += deg_[u]    #count the probability of a random edge
        Mod += ( float(EWC) - float(RE*RE)/float(2*m_) )
    Mod = Mod/float(2*m_)
    if _DEBUG_:
        print("Modularity: {}".format(Mod))
    return Mod
```

- This function computes the modularity of the most recent split.
- It takes a list of components as the input and using the normal modularity formulae, edges between communities and subtract that from the probability of a random edge.

```
def runGirvanNewman(G, Orig_deg, m_):
    BestQ = 0.0
    Q = 0.0
    for i in range(0,5):
        print("Step : " + str(i+1))
        CmtyGirvanNewmanStep(G)
        Q = _GirvanNewmanGetModularity(G, Orig_deg, m_);
        print("Modularity of decomposed G: {}".format(Q))
        if Q > BestQ:
            BestQ = Q
            Bestcomps = list(nx.connected_components(G))    #Best Split
            print("Identified components: {}".format(Bestcomps))
        if G.number_of_edges() == 0:
            break
```

- This function runs the Girvan Newman algorithm for 5 steps.
- If we remove the for loop and replace it with an infinite while loop it will stop when we have reached maximum modularity.
- Here Q is the modularity value for the most recent split and $BestQ$ is the optimal modularity value.

```
celegans_graph=nx.Graph()
f = open("/content/out.dimacs10-celegans_metabolic", "r")
lines = f.readlines()
lines.pop(0)
for line in lines:
    l = line.split()
    celegans_graph.add_edge(l[0],l[1])
G = celegans_graph
```

The *Graph* method of the NetworkX library creates an empty graph with no nodes and no edges, labeled as *celegans_graph*. The dataset file is read line by line using the for loop. Inside the loop, edges are added to the graph *celegans_graph* using the *add_edge* method of the NetworkX library.


```
def main(argv):
    if len(argv) < 2:
        sys.stderr.write("Usage: %s <input graph>\n" % (argv[0],))
        return 1
    graph_fn = argv[1]

    if _DEBUG_:
        print('G nodes: {} & G no of nodes: {}'.format(G.nodes(), G.number_of_nodes()))

    n = G.number_of_nodes()    #|V|
    A = nx.adj_matrix(G)       #adjacencnt matrix

    m_ = 0.0    #the weighted version for number of edges
    for i in range(0,n):
        for j in range(0,n):
            m_ += A[i,j]
    m_ = m_/2.0
    if _DEBUG_:
        print("m: {}".format(m_))

    #calculate the weighted degree for each node
    Orig_deg = {}
    Orig_deg = UpdateDeg(A, G.nodes())

    #run Newman alg
    runGirvanNewman(G, Orig_deg, m_)

if __name__ == "__main__":
    sys.exit(main(sys.argv))
```

Here we are creating the adjacency matrix labeled A for our network G . Then creating a weighted network where the weights are the activation probability.

Dataset 2: Jazz Musicians

```
jazz_graph=nx.Graph()

with open('out.arenas-jazz','r') as f:
    lines=f.readlines()
    lines.pop(0)
    for line in lines:
        line=line.strip()
        lis=line.split('\t')
        if lis != '% sym unweighted':
            jazz_graph.add_edge(lis[0],lis[1])
G = jazz_graph
```

The *Graph* method of the NetworkX library creates an empty graph with no nodes and no edges, labeled as *celegrans_graph*. The dataset file is read line by line using the for loop. Inside the loop, edges are added to the graph *celegrans_graph* using the *add_edge* method of the NetworkX library.

```
def main(argv):
    if len(argv) < 2:
        sys.stderr.write("Usage: %s <input graph>\n" % (argv[0],))
        return 1
    graph_fn = argv[1]
    if _DEBUG_:
        print('G nodes: {} & G no of nodes: {}'.format(G.nodes(), G.number_of_nodes()))

    n = G.number_of_nodes()    #|V|
    A = nx.adj_matrix(G)       #adjacencnt matrix

    m_ = 0.0    #the weighted version for number of edges
    for i in range(0,n):
        for j in range(0,n):
            m_ += A[i,j]
    m_ = m_/2.0
    if _DEBUG_:
        print("m: {}".format(m_))

    Orig_deg = {}
    Orig_deg = UpdateDeg(A, G.nodes())

    #run Newman alg
    runGirvanNewman(G, Orig_deg, m_)

if __name__ == "__main__":
    sys.exit(main(sys.argv))
```

Here we are creating the adjacency matrix A for our network G . Then creating a weighted network where the weights are the activation probability.

RAVASZ CLUSTERING ALGORITHM

It is an agglomerative hierarchical clustering algorithm and assumes each node to be a single community initially. It merges nodes and communities with high similarity, so a similarity matrix needs to be defined which is determined from the adjacency matrix.

Similarity Matrix: It determines how similar two nodes are. Similarity matrix entries are high for nodes that probably belong to the same community. Nodes that are connected to each other or share many common neighbors likely belong to the same community.

Group Similarity: Clusters are merged depending on their mutual similarity which can be through single, complete or average cluster linkage.

To find communities, the similarity value for all community pairs is computed and the pair with maximum similarity is merged. This is repeated till only a single community is left.

In our case, we have to run the algorithm for only 5 steps.

Dataset 1: *Caenorhabditis elegans* (metabolic)

```
import matplotlib.pyplot as plt #for plotting
import numpy as np #array manipulations
import networkx as nx #for graph making
import pandas as pd #storing dataframe
```

The required libraries are imported.

- Numpy
- NetworkX
- Pandas
- Matplotlib

```
celegans_g = nx.Graph()

f = open("/content/out.dimacs10-celegans_metabolic", "r")
lines = f.readlines()
lines.pop(0)
for line in lines:
    l = line.split()
    celegans_g.add_edge(int(l[0]), int(l[1]))
    #celegans.append([int(l[0]), int(l[1])])
celegans_mat = nx.adjacency_matrix(celegans_g).todense()
celegans = nx.to_pandas_adjacency(celegans_g, dtype=int)
print("adjacency matrix for celegans data\n")
print(celegans)
```

- The *Graph* method of the NetworkX library creates an empty graph with no nodes and no edges, labeled as *celegans_g*. The dataset file is read line by line using the for loop. Inside the loop,

edges are added to the network *celegrans_g* using the *add_edge* method of the NetworkX library.

- *celegrans_mat* is the adjacency matrix representation of the network *celegrans_g*.
- The *to_pandas_adjacency* method converts the *celegrans_mat* to a dataframe labeled *celegrans*.

```
class length_find_class(object):  
    def __init__(self):  
        pass
```

A class labeled *length_find_class* to hold all the methods relating to distance between nodes and communities.

```
def compute_length(self, communities):  
    """  
    matrix creation for distances between individual nodes and merged communities.  
    """  
  
    length_mat = np.zeros((len(communities), len(communities)))  
    for i in range(length_mat.shape[0]):  
        for j in range(length_mat.shape[0]):  
            if i!=j:  
                length_mat[i,j] = float(self.length_calculate(communities[i], communities[j]))  
            else:  
                length_mat[i,j] = 10**4  
    return length_mat
```

A method of the class *length_find_class*; creates a matrix whose entries correspond to the distance between nodes and merged

communities. It creates a matrix labeled *length_mat* with rows and columns equal to the number of communities.

It iterates through each element of the matrix *length_mat* and passes control to the *length_calculate* method to compute the distance between the row number *i* corresponding to community *i* and column number *j* corresponding to community *j* if the row and column number do not correspond to the same community, that is, $i \neq j$.

```
def length_calculate(self,community1,community2):  
    "for finding distance between 2 individual nodes or 2 communities"  
    dist = []  
    for i in range(len(community1)):  
        for j in range(len(community2)):  
            try:  
                dist.append(np.linalg.norm(np.array(community1[i])-np.array(community2[j])))  
            except:  
                dist.append(self.internodelength(community1[i],community2[j]))  
    return min(dist)
```

A method of the class *length_find_class*; takes as input, two communities. The try block computes the euclidean distance between the components of the two inputs and appends it to the list labeled *dist*, if the components of both the inputs are individual nodes, otherwise passes control to the except block, wherein the control is further passed to the *internodelength* method and its output is also

appended to the list *dist*. The output of this method is the minimum distance stored in the list *dist*.

```
def internodelength(self,s1,s2):
    """
    for finding distance between a node and a community.
    """
    if str(type(s2[0]))!='<class \'list\'>':
        s2=[s2]
    if str(type(s1[0]))!='<class \'list\'>':
        s1=[s1]
    m = len(s1)
    n = len(s2)
    dist = []
    if n>=m:
        for i in range(n):
            for j in range(m):
                if (len(s2[i])>=len(s1[j])) and str(type(s2[i][0]))!='<class \'list\'>':
                    dist.append(self.intercommlength(s2[i],s1[j]))
                else:
                    dist.append(np.linalg.norm(np.array(s2[i])-np.array(s1[j])))
    else:
        for i in range(m):
            for j in range(n):
                if (len(s1[i])>=len(s2[j])) and str(type(s1[i][0]))!='<class \'list\'>':
                    dist.append(self.intercommlength(s1[i],s2[j]))
                else:
                    dist.append(np.linalg.norm(np.array(s1[i])-np.array(s2[j])))
    return min(dist)
```

- A method of the class *length_find_class*, which takes as input, components *s1* and *s2* of two communities.
- This method deals with two cases:
 - *Either of s1 or s2 is an individual node*
 - Assuming *s1* is an individual node and *s2* is a community, it appends the computed Euclidean

distance between $s1$ and the component of $s2$ to a list labeled $dist$.

- ***Both $s1$ or $s2$ are two communities***
 - Passes control to *intercommlength* method and its output is also appended to the list $dist$.
- The output of this method is the minimum distance stored in the list $dist$.

```
def intercommlength(self,c1,community):  
    if community[0]!='<class \'list\'>':  
        community = [community]  
    dist = []  
    for i in range(len(c1)):  
        for j in range(len(community)):  
            dist.append(np.linalg.norm(np.array(c1[i])-np.array(community[j])))  
    return min(dist)
```

- A method of the class *length_find_class*; takes as input two communities $c1$ and $community$.
- It finds the distance between two different communities. Since both of the entities are communities, euclidean distance is used to find the distance between their i th and j th components for all i and j , and this distance is appended to a list labeled $dist$. The output of this method is the minimum distance stored in the list $dist$.


```
prog_data = [[i] for i in range(celegans_mat.shape[0])]
communities = [[list(celegans_mat[i])] for i in range(celegans_mat.shape[0])]
m = len(communities)
distcal = length_find_class()
```

- A 2-D list *prog_data* is initialized; it contains all the nodes with each of the nodes of the network *celegans_g* being their own community initially.
- An object labeled *distcal* of the class *length_find_class* is created to keep track of the distances between nodes and communities. Here, *m* is the number of nodes in the network *celegans_g*.

```
for _ in range(1,6): #5 steps
    print("Step :- ", _)
    print('#communities before clustering :- ',m)
    length_mat = distcal.compute_length(communities)
    community_ind_needed = np.where(length_mat==length_mat.min())[0]
    value_to_add = communities.pop(community_ind_needed[1])
    communities[community_ind_needed[0]].append(value_to_add)

    print('1st node to be merged in this step :- ',prog_data[community_ind_needed[0]])
    print('2nd node to be merged in this step :- ',prog_data[community_ind_needed[1]])

    prog_data[community_ind_needed[0]].append(prog_data[community_ind_needed[1]])
    prog_data[community_ind_needed[0]] = [prog_data[community_ind_needed[0]]]
    v = prog_data.pop(community_ind_needed[1])
    m = len(communities)

    print('All communities present :- ',prog_data)
    print('Communities merged in this step :- ',prog_data[community_ind_needed[0]])
    print('#Communities before clustering :- ',m)
    print('\n')
```

- The for loop is to perform agglomerative clustering for 5 steps.

- The object *distcal* and the method *compute_length* are used to find a matrix labeled *length_mat* whose entries correspond to the distance between community pairs.
- Next, the node or community pair with the least distance between them is computed and put in the same list in the *communities* matrix as well as the *prog_data* matrix.
- Finally, the present state of clustering is shown by printing the *prog_data* matrix, the communities merged in the current step using the indices of the node or community pair with the least distance between them, and the number of communities after the merging in the current step.

Dataset 2: Jazz Musicians

```
jazz_g = nx.Graph()

f = open("/content/out.arenas-jazz", "r")
lines = f.readlines()
lines.pop(0)
for line in lines:
    l = line.split()
    jazz_g.add_edge(int(l[0]), int(l[1]))
jazz_mat = nx.adjacency_matrix(jazz_g).todense()
jazz = nx.to_pandas_adjacency(jazz_g, dtype=int)
print("adjacency matrix for jazz data\n")
print(jazz)
```

- The *Graph* method of the NetworkX library creates an empty graph with no nodes and no edges, labeled as *jazz_g*. The dataset file is read line by line using the for loop. Inside the loop, edges are added to the network *celegrans_g* using the *add_edge* method of the NetworkX library.
- *jazz_mat* is the adjacency matrix representation of the network *jazz_g*.
- The *to_pandas_adjacency* method converts the *jazz_mat* to a dataframe labeled *jazz*.

```
prog_data = [[i] for i in range(jazz_mat.shape[0])]
communities = [[list(jazz_mat[i])] for i in range(jazz_mat.shape[0])]
m = len(communities)
distcal = length_find_class()
```

- A 2-D list *prog_data* is initialized; it contains all the nodes with each of the nodes of the network *jazz_g* being their own community initially.
- An object labeled *distcal* of the class *length_find_class* is created to keep track of the distances between nodes and communities. Here, *m* is the number of nodes in the network *jazz_g*.

```
for _ in range(1,6): #5 steps
    print("Step :- ", _)
    print('#communities before clustering      :- ',m)
    length_mat = distcal.compute_length(communities)
    community_ind_needed = np.where(length_mat==length_mat.min())[0]
    value_to_add      = communities.pop(community_ind_needed[1])
    communities[community_ind_needed[0]].append(value_to_add)

    print('1st node to be merged in this step      :- ',prog_data[community_ind_needed[0]])
    print('2nd node to be merged in this step      :- ',prog_data[community_ind_needed[1]])

    prog_data[community_ind_needed[0]].append(prog_data[community_ind_needed[1]])
    prog_data[community_ind_needed[0]] = [prog_data[community_ind_needed[0]]]
    v = prog_data.pop(community_ind_needed[1])
    m = len(communities)

    print('All communities present      :- ',prog_data)
    print('Communities merged in this step      :- ',prog_data[community_ind_needed[0]])
    print('#Communities before clustering      :- ',m)
    print('\n')
```

- The for loop is to perform agglomerative clustering for 5 steps.
- The object *distcal* and the method *compute_length* are used to find a matrix labeled *length_mat* whose entries correspond to the distance between community pairs. Next, the node or community pair with the least distance between them is computed and put in the same list in the *communities* matrix and the *prog_data* matrix.
- Finally, the present state of clustering is shown by printing the *prog_data* matrix, the communities merged in the current step using the indices of the node or community pair, and the number of communities after the merging in the current step.

INFERENCE

Dataset 1: Caenorhabditis elegans (metabolic)

→ Girvan Newman algorithm:

The communities after step 1:

{'80', '79', '78', '45', '41', '76', '43', '42', '75', '77', '73', '44'} and all other nodes of the network as the 2nd community.

The communities after step 2:

{'43', '41', '76', '79', '75', '44', '73', '45', '78', '80', '77', '42'}, {'279', '162', '158', '161', '160'} and all other nodes of the network as the 3rd community.

The communities after step 3:

{'43', '41', '76', '79', '75', '44', '73', '45', '78', '80', '77', '42'}, {'107', '108', '452'}, {'279', '162', '158', '161', '160'} and all other nodes of the network as the 4th community.

The communities after step 4:

{'43', '41', '76', '79', '75', '44', '73', '45', '78', '80', '77', '42'}, {'107', '108', '452'}, {'279', '162', '158', '161', '160'}, {'304', '303'} and all other nodes of the network as the 5th community.

The communities after step 5:

{'43', '41', '76', '79', '75', '44', '73', '45', '78', '80', '77', '42'}, {'74', '382', '50', '46', '47', '48'}, {'107', '108', '452'}, {'279', '162', '158', '161', '160'}, {'304', '303'} and all other nodes of the network as the 6th community.

Total communities after 5 steps:- 6

The running time was 120.40 seconds.

→ **Ravasz algorithm:**

The communities after step 1:

[[144, [146]]] and each of the remaining individual nodes of the network as a community of its own.

The communities after step 2:

[[144, [146]]], [[316, [317]]] and each of the remaining individual nodes of the network as a community of its own.

The communities after step 3:

[[144, [146]]], [[316, [317]]], [[320, [321]]] and each of the remaining individual nodes of the network as a community of its own.

The communities after step 4:

[[144, [146]]], [[316, [317]]], [[320, [321]]], [[40, [72]]] and each of the remaining individual nodes of the network as a community of its own.

The communities after step 5:

[[144, [146]]], [[316, [317]]], [[320, [321]]], [[40, [72]]], [[148, [151]]] and each of the remaining individual nodes of the network as a community of its own.

Total communities after 5 steps:- 448 (or 453 - 5)

The running time was 19.56 seconds.

→ **Differences between observations of both algorithms:**

Girvan Newman took more time to run than the Ravasz algorithm on the celegans dataset.

It was also observed for the Girvan Newman algorithm, that it worked in a top-down fashion. First parent nodes were visited and then their children. Whereas in the case of Ravasz, it worked in a bottom-up approach. First children and then the parent.

It was practically inferred from the Girvan Newman algorithm that the increment in the number of communities per step was = 1, thus after completion of 5 steps, 6 communities were formed; whereas

it was observed in the Ravasz algorithm that the decrement in the number of communities per step was = 1. And the number of communities after completion was 448.

Dataset 2: Jazz Musicians

→ Girvan Newman algorithm:

The communities after step 1:

{'30', '78', '77', '79'} and all other nodes of the network as 2nd community.

The communities after step 2 :

{'30', '78', '77', '79'}, {'49'} and all other nodes of the network as the 3rd community.

The communities after step 3:

{'30', '78', '77', '79'}, {'49'}, {'197'}, {'198'}, {'162'}, {'195'} and all other nodes of the network as the 7th community.

The communities after step 4:

{'30', '78', '77', '79'}, {'49'}, {'151'}, {'197'}, {'198'}, {'162'}, {'195'} and all other nodes of the network as the 8th community.

The communities after step 5:

{'77', '30', '79', '78'}, {'180', '182', '181', '190'}, {'49'}, {'151'}, {'197'}, {'198'}, {'162'}, {'195'} and all other nodes of the network as the 9th community.

Total communities after 5 steps:- 9

The running time was 34.11 seconds.

→ **Ravasz algorithm:**

The communities after step 1:

[[13, [18]]] and each of the remaining individual nodes of the network as a community of its own.

The communities after step 2:

[[13, [18]]], [[32, [34]]] and each of the remaining individual nodes of the network as a community of its own.

The communities after step 3:

[[13, [18]]], [[32, [34]]], [[49, [50]]] and each of the remaining individual nodes of the network as a community of its own.

The communities after step 4:

[[13, [18]]], [[32, [34]]], [[49, [50]]], [[58, [59]]] and each of the remaining individual nodes of the network as a community of its own.

The communities after step 5:

[[13, [18]]], [[32, [34]]], [[49, [50]]], [[58, [59]]], [[69, [74]]] and each of the remaining individual nodes of the network as a community of its own.

Total communities after 5 steps:- 193 (or 198 - 5)

The running time was 3.32 seconds.

→ **Differences between observations of both algorithms:**

Girvan Newman took more time to run than the Ravasz algorithm on the jazz dataset.

It was also observed for the Girvan Newman algorithm, that it worked in a top-down fashion. First parent nodes were visited and then their children. Whereas in the case of Ravasz, it worked in a bottom-up approach. First children and then the parent.

It was practically inferred in the Girvan Newman algorithm that the increment in the number of communities per step was uneven (not = 1) thus after completion of 5 steps, 9 communities were formed; whereas in the Ravasz algorithm, it was observed that the decrement in the number of communities per step was = 1. And the number of communities after completion was 193.

PROBLEM 5:

Create a scale-free network using the appropriate Python package. Apply Independent Cascade Model to find the maximum number of 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 that are 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, y . Then it should be $p(v,w)+p(v,x)+p(v,y)=1$.

→ [Code Notebook](#)

THEORY

Scale-free Network: A network whose degree distribution follows Pareto Law distribution is called a scale-free network. It is a fat tail distribution.

Two models are used to simulate a scale-free network:

→ ***Barabasi-Albert Model:*** It captures the small world phenomena but produces a very small clustering coefficient as compared to the original network. It is since, in this model, the node's growth rate is solely dependent on its degree, a phenomenon called first mover's advantage.

→ ***Bianconi-Barabasi Model:*** It simulates a network very close to a scale-free network and produces both; comparable clustering coefficient and small-world phenomena.

Independent Cascade Model (ICM): It is a sender-centric model. Each node has one chance to activate its neighbors. Active nodes are senders and the nodes that are getting activated are receivers. ICM gives a probabilistic approach to see which nodes will be activated at each step. Given a set of initially activated nodes A_0 , diffusion graph $G(V, E)$, activation probabilities $p_{v,w}$, if we run this algorithm infinitely, then it gives all the nodes that will be activated.

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

The imported libraries are:

- NetworkX
- Matplotlib
- Pandas

```
G = nx.barabasi_albert_graph(250, 10)
```

The *barabasi_albert_graph* method of the NetworkX library creates a network of 250 nodes with 10 as the number of edges to attach from a new node to existing nodes in the network.

```
plt.figure(figsize =(20, 20))
nx.draw_networkx(G, with_labels = True, node_color ='blue')
```

The *draw_networkx* method of the NetworkX library draws a visual representation of the network *G*, with *node_color* set to blue and *with_labels* parameter set to True to see the node labels for each node.

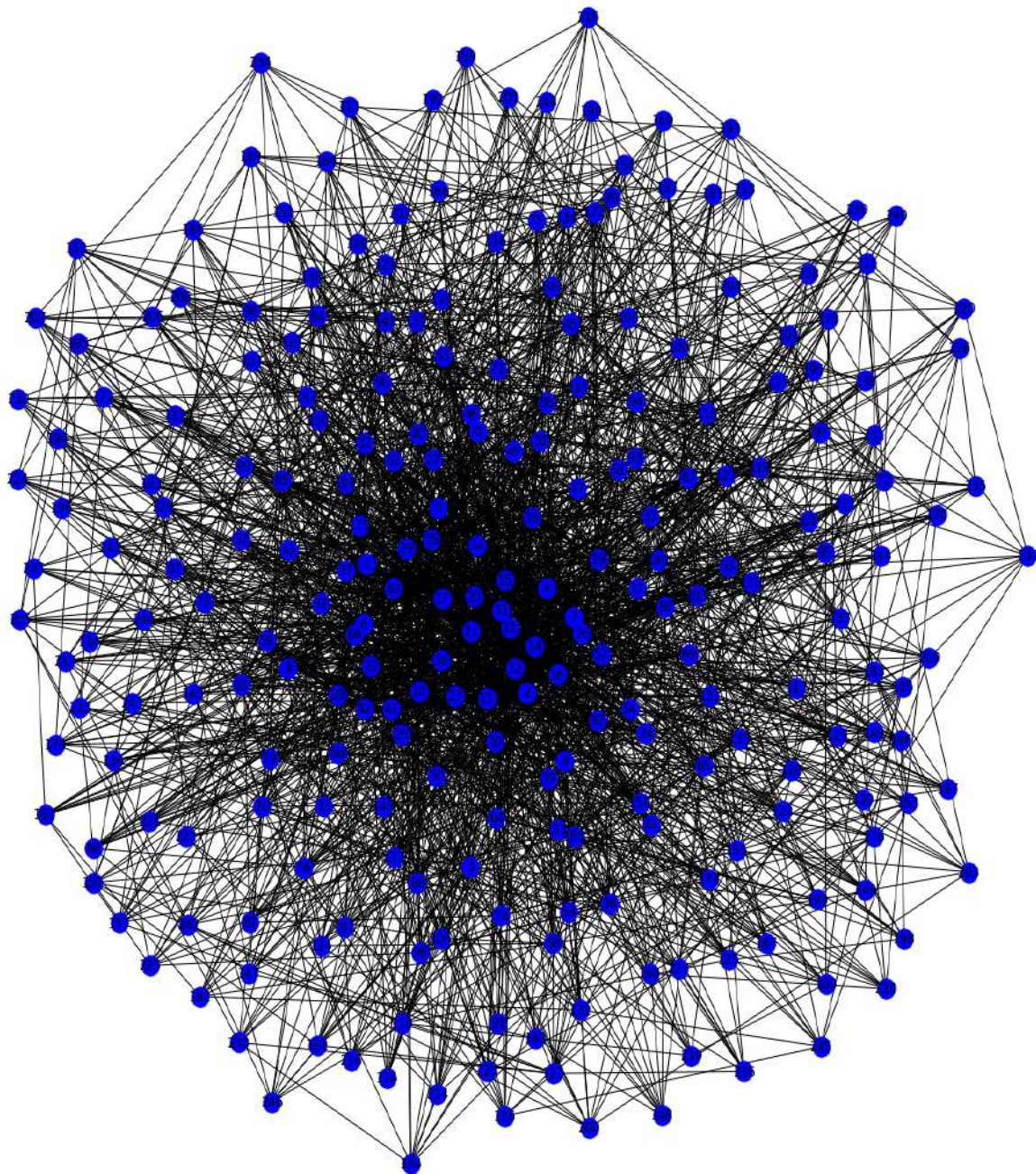


Figure 5.1 : Barabasi-Albert Model Network

```
for entry in nx.generate_adjlist(G):      # to print the adjacency list of the network created
    print(entry)
```

A for loop is used to print the entries of the adjacency list of the generated network G .

```
p_map = {}          #it is used to store the probability of outgoing edges from vertices of the network
adj_list = {}
```

- p_map is a dictionary used to store the activation probability value of edges between vertices of the network G .
- adj_list stores the adjacency list of the network G as a dictionary.

```
for entry in nx.generate_adjlist(G):
    entry = list(map(int,entry.split()))
    node = entry[0]          #node: vertex in question whose outgoing edges we will look at
    size = len(entry)-1      #count of outgoing edges/ neighbours of node in each iteration
    p_list = []
    r_sum=0

    #inner loop 1
    for i in range(size):    #for node, we are creating random numbers having count = edges connected to the node; and appending the random n
        r1 = np.random.rand()
        p_list.append(r1)
        r_sum+=r1

    #inner loop 2
    for i in range(size):    #this loop normalizes the random values stored in the list and gives the activation probability value for each con
        p_list[i]/=r_sum

    #inner loop 3
    for i in range(1,len(entry)):    #this loop is to create an adjacency list and a dictionary whose entries correspond to probability va
        p_map[(node,entry[i])]=p_list[i-1]
        if(node not in adj_list):
            adj_list[node]=[]
        adj_list[node].append(entry[i])
```

- The `generate_adjlist` method of NetworkX library returns lines of data of network G in adjacency list format.

- The outer loop reads the output of *generate_adjlist* method line by line.
- Inside the outer loop, the line (*entry*) read at each iteration of the loop is split and stored as a list, labeled *entry*.
- *node* is the vertex of the network *G* in question and we will look at the edges connected to it.
- *size* is the count of edges connected to *nodes*.
- *p_list* is a list, which we use to store the activation probability value of each edge connected to *node*.
- The first inner loop generates a random number for each edge connected to *node* and stores it in the *p_list*.
- The second inner loop normalizes the random numbers stored in the list *p_list* for each edge connected to *node* and this normalized value corresponds to the activation probability of that edge connected to *node*. The normalization maintains the property: The sum of activation probabilities of all the edges connected to a vertex is 1.
- The third inner loop creates an adjacency list stored as a dictionary *adj_list* and a dictionary *p_map* whose entries correspond to activation probability values of edges.


```
print(adj_list)
```

This prints the adjacency list denoted by *adj_list* of the network G.

```
for i in range(0,5):
    visited = [False] * 250
    queueBFS = []
    s=np.random.randint(0,250)
    queueBFS.append(s)
    visited[s] = True
    activated_nodes = []
    while queueBFS:
        s = queueBFS.pop(0)
        activated_nodes.append(s)
        if(s not in adj_list):
            continue
        for i in adj_list[s]:
            r = np.random.rand()/len(adj_list[s])
            if visited[i] == False and r < p_map[(s,i)]:
                queueBFS.append(i)
                visited[i] = True
    print(f'Activated Nodes : {activated_nodes}')
    print(f'Count of activated nodes:{len(activated_nodes)}')
```

- The loop is to repeat ICM 5 times.
- In each iteration of ICM, we choose a vertex of the network at random as a seed point and run a BFS starting from that seed point.
- In BFS, vertices are added in the queue with an added constraint. So, the modified constraint states that a vertex v in the adjacency list of currently activated vertex s is added to the queue only if its activation probability is more than the generated normalized random number r and v was not activated previously.

INFERENCE

→ The activated nodes after run 1:

[52, 66, 76, 85, 108, 144, 150, 218, 226, 81, 114, 78, 119, 124, 143, 211, 93, 146, 228, 112, 122, 151, 168, 202, 147, 189, 203, 95, 129, 140, 141, 201, 230, 133, 86, 89, 91, 192, 231, 246, 142, 155, 210, 229, 245, 148, 244, 172, 181, 126, 158, 235, 153, 233, 247, 160, 217, 236, 239, 206, 111, 123, 196, 213, 215, 248, 131, 149, 167, 220, 227, 157, 208, 135, 193, 115, 145, 177, 97, 130, 138, 209, 234, 121, 170, 175, 190, 240, 223, 214, 222, 159, 163, 169, 221, 237, 238, 205, 116, 128, 173, 183, 243, 200, 204, 139, 176, 197, 199, 212, 207, 225, 232, 185, 152, 154, 219, 182, 198, 156, 179, 186, 184, 224, 194, 171, 241, 174]

Total: 128

→ The activated nodes after run 2:

[48, 49, 104, 105, 115, 214, 243, 52, 60, 94, 141, 156, 172, 178, 205, 213, 224, 118, 184, 209, 241, 106, 130, 168, 216, 143, 145, 197, 199, 212, 236, 66, 76, 85, 103, 108, 144, 150, 217, 218, 226, 89, 95, 157, 167, 233, 159, 179, 208, 211, 225, 142, 247, 171, 223, 235, 244, 222, 248, 187, 190, 203, 210, 245, 228, 139, 189, 119, 126, 147, 181, 182, 194, 180, 185, 193, 81, 114, 134, 124, 146,

120, 200, 204, 112, 122, 151, 202, 242, 174, 93, 97, 101, 138, 234, 102, 132, 153, 196, 215, 219, 232, 149, 220, 206, 246, 239, 129, 155, 169, 221, 237, 238, 230, 231, 240, 140, 201, 133, 158, 229, 152, 148, 154, 98, 175, 207, 177, 127, 137, 165, 166, 191, 131, 227, 135, 198, 107, 110, 121, 162, 128, 195, 170, 176, 163, 123]

Total: 147

➔ The activated nodes after run 3:

[96, 121, 154, 199, 123, 152, 179, 207, 219, 244, 196, 205, 239, 248, 235, 130, 133, 144, 151, 190, 194, 198, 200, 203, 243, 234, 210, 180, 185, 135, 193, 240, 147, 189, 202, 242, 153, 229, 246, 247, 204, 206, 214, 182, 213, 237, 139, 176, 228, 160, 217, 230, 231, 236, 223, 245, 224]

Total: 57

➔ The activated nodes after run 4:

[116, 157, 182, 189, 209, 214, 190, 239, 236]

Total: 9

➔ The activated nodes after run 5:

[146, 172, 181, 222, 248, 196, 234, 240, 236]

Total: 9

- ➔ We infer that the count of activated nodes at the end of each ICM run differs depending on the seed point we initially choose.
- ➔ If a good seed point is chosen, the number of activated nodes increases tremendously as to when a poor choice of seed point is made, therefore, a wise choice of initially activated nodes is crucial to maximizing the spread of information diffusion.

-----END OF REPORT-----