Introduction to social network project

Dr. Reem Essameldin Ebrahim

Eng. Fady

Social circles: Facebook

Team:

Basmala Salama	20221060328
Irinie Magued	20221440868
Elham Hamed	20221468745
Menna sead	20221371722
Menna Samy	20221031698

Introduction

The purpose of this project is to analyze the structure of a social network based on data from Facebook. This type of analysis helps us understand the dynamics of online social interactions, the key influencers within the network, and how information or connections spread through the network.

Problem Statement

In today's digital world, social networks, such as Facebook, play a crucial role in connecting individuals and enabling communication, with millions of users and complex interactions, it becomes challenging to identify key influencers, understand the network's structure, and assess how information spreads. The problem lies in effectively analyzing large-scale network data to extract meaningful insights, such as the identification of central users and the network's connectivity.

Study Overview

For this study, we use data from **the Stanford Network Analysis Project (SNAP),** which provides a dataset representing the Facebook ego network. This dataset contains the social network of a single user (ego) and their friends (alter). The goal is to analyze the relationships between the ego and their friends, as well as the connections between friends themselves.

Data Description

The dataset, which can be accessed from the Stanford SNAP website, is structured as an undirected graph where each node represents a user, and the edges represent friendships or connections between users. The dataset includes:

- **Nodes**: Each node represents a user on Facebook.
- **Edges**: An edge between two nodes indicates a friendship between those users.
- **Attributes**: Additional information about each user, such as the number of friends, is included in the dataset.

You can find the dataset here: SNAP Facebook Dataset.

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

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

Agenda (What We Cover!)

1. Introduction to Social Network Project

- You are working with the **Ego Facebook dataset** provided by Stanford, which contains social circles (friendship relationships).

2. Graph Properties and Analysis

- You calculated the **degree** of each node (i.e., the number of friends each user has) and identified the **node with the highest and lowest degrees**.
- You computed the **average degree** of the network, indicating that on average, each user has 43 relationships.

3. Connectivity and Visualization

- You checked whether the graph is **connected** (i.e., if all users are reachable from any other user).

4. Degree Distribution

- You calculated the **degree distribution** of the network and normalized it to visualize how nodes are distributed across different degree values (number of friends).

5. Adjacency and Edge List

- You explored the **adjacency matrix** and **edge list** to analyze the structure of the network.

6. Graph Density

- You calculated the **graph density**, which measures how close the graph is to being fully connected. You found the graph to be **sparse** with a density of 0.0108.

7. Clustering Coefficients

- You calculated the **local clustering coefficients** of nodes, which measure the extent to which a node's neighbors are also connected.
- You computed the **global clustering coefficient (transitivity)** of the graph, which is 0.519, suggesting that the network has clusters but is not fully cohesive.

8. Shortest Path (Diameter)

- You calculated the **diameter** of the graph (the longest shortest path between any two nodes) and analyzed **shortest path lengths** from a specific node (Node 687) to others in the network.

9. Community Detection

- You introduced the concept of **modularity**, which measures how well a network is divided into communities.

10. Centrality Measures

- Degree Centrality
- Betweenness Centrality
- Closeness Centrality
- Eigenvector Centrality

About the network/graph G(N,E):

- A central user (the **ego**).
- The **ego's friends** (called **alters**).
- The **relationships between the alters** (friendships among the ego's friends).

```
:
#load data
file_path = "facebook_combined.txt"

G = Graph.Read_Edgelist(file_path, directed=False)

#About graph G(N,E):
print(f"Graph has {G.vcount()} nodes and {G.ecount()} edges")

```

Graph has 4039 nodes (fb users) and 88234 edges.

- **Node Degree:** The number of edges (connections) each node (user) has.

```
# Degree of each node
degrees = G.degree()

# Format the output as a list of (node, degree) pairs
degree_output = [(i, degrees[i]) for i in range(len(degrees))]
print("Node Degrees:")
for node, degree in degree_output:
    print(f"Node {node}: Degree {degree}")

```

```
Node Degrees:
Node 0: Degree 347
Node 1: Degree 17
Node 2: Degree 10
Node 3: Degree 17
Node 4: Degree 10
Node 5: Degree 13
Node 6: Degree 6
Node 7: Degree 20
Node 8: Degree 8
```

Find the node with the highest & lowest degrees

```
# Find the node with the highest degree
max_degree = max(degrees)
max_degree_node = degrees.index(max_degree)

# Find the node with the lowest degree
min_degree = min(degrees)
min_degree_node = degrees.index(min_degree)

# Print the results
print(f"Node with the highest degree: Node {max_degree_node} with degree {ma
print(f"Node with the lowest degree: Node {min_degree_node} with degree {min

```

```
Node with the highest degree: Node 107 with degree 1045
Node with the lowest degree: Node 11 with degree 1
```

The most connected nodes in the network. Nodes with the highest degrees are potentially influential users or hubs. Node 107 has many connections (1045), it may be a popular user(actor, influencer for example). Lowest degree is 1 (can't be 0 since it is connected)

- **Average Degree:** The average number of edges per node**.**

```
avg_degree = sum(degrees) / len(degrees)
print("Average Degree:", avg_degree)
```

```
Average Degree: 43.69101262688784
```

Being in this network means you will have on average 43 relationships (friends).

Network is sparse, 88234 links and being in this network you will have about 43 link.

- **Connectivity of Graph:** Determines whether all nodes are reachable.

```
# Check connectivity
is_connected = G.is_connected()
print("Graph Connected:", is_connected)
```

```
Graph Connected: True
```

The network/graph is connected, meaning all users are reachable from any other user

which means that any user can be reached from any other user via a series of friendships.

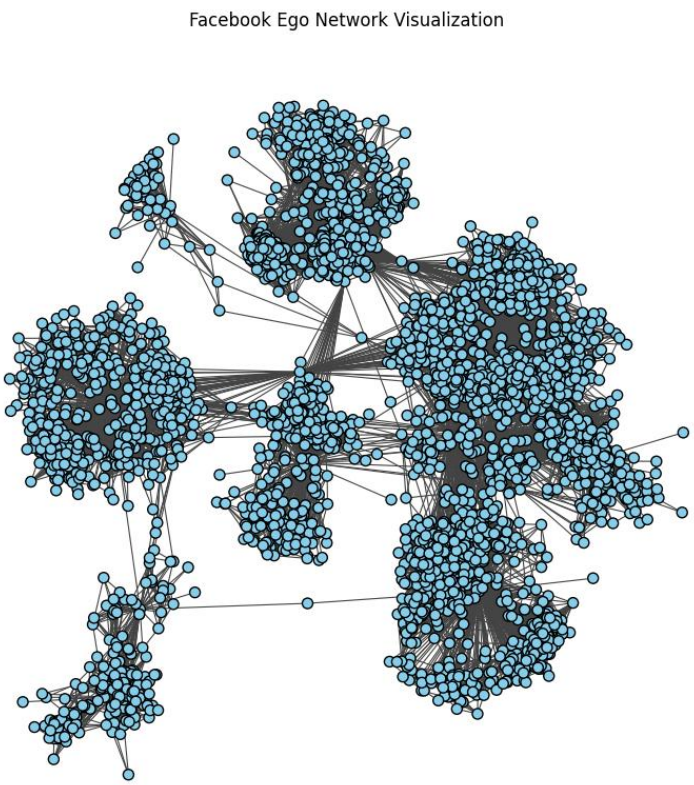
If it wasn't connected we would see the connected components:

```python
# Find the connected components
components = G.connected_components()
num_components = len(components)
print(f"Number of connected components: {num_components}")

# Sizes of connected components
component_sizes = components.sizes()
print(f"Sizes of connected components: {component_sizes}")

#visualize the largest component
largest_component = components.giant()
print(f"Number of nodes in the largest connected component: {len(largest_component.vs)}")

```

```
Number of connected components: 1
Sizes of connected components: [4039]
Number of nodes in the largest connected component: 4039
```

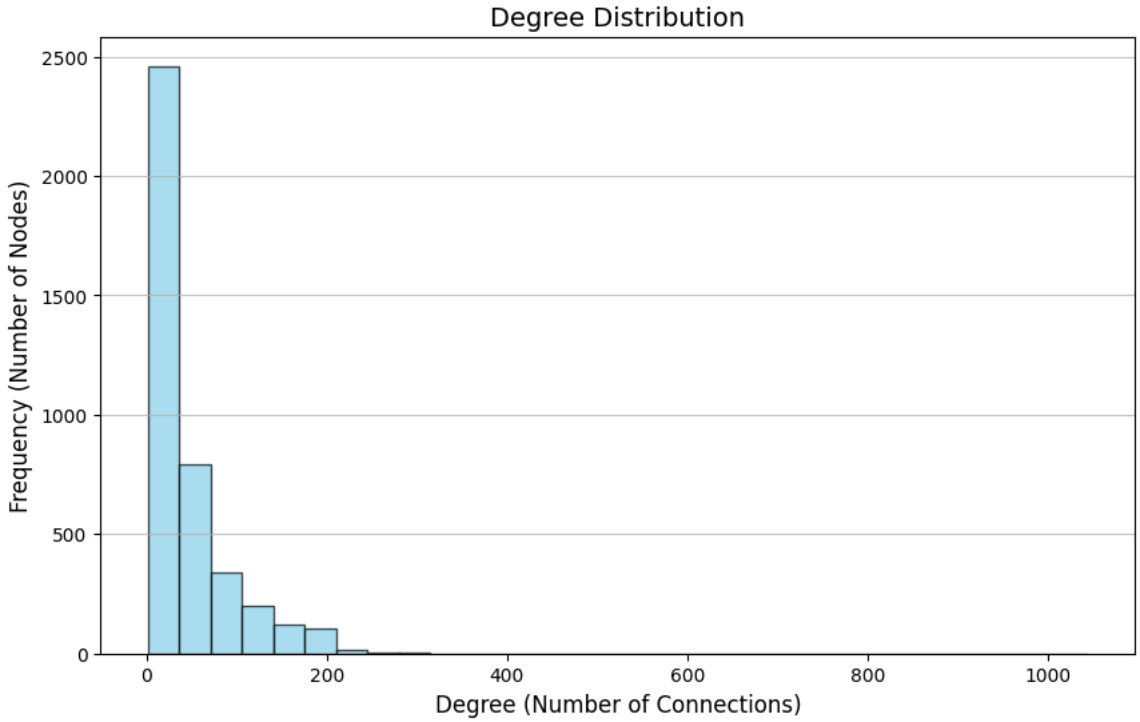
Plot the Graph:

```python
# Layout for the graph
layout = G.layout("fr")

# Set up the plot size
plt.figure(figsize=(10, 10))

# Plot the graph
plot(G, layout=layout, target=plt.gca(), vertex_size=10, vertex_color='skybl
     edge_width=0.8, bbox=(0, 0, 1, 1), margin=50)

# Set title of the plot then display it
plt.title("Facebook Ego Network Visualization")
plt.show()

```



Degree Distribution: (first graph is frequency, second scalled 0-1)

```
:
1  # Plot the degree distribution
2  plt.figure(figsize=(10, 6))
3  plt.hist(degrees, bins=30, color='skyblue', edgecolor='black', alpha=0.7)
4  plt.title("Degree Distribution", fontsize=14)
5  plt.xlabel("Degree (Number of Connections)", fontsize=12)
6  plt.ylabel("Frequency (Number of Nodes)", fontsize=12)
7  plt.grid(axis='y', alpha=0.75)
8  plt.show()
```



The **x-axis** shows the degree (number of connections for nodes).
The **y-axis** shows the number of nodes with that degree.

Some insights:

Skewness:
Social networks typically have a **skewed distribution**, with many nodes having a low degree and a few nodes (hubs) having a very high degree.

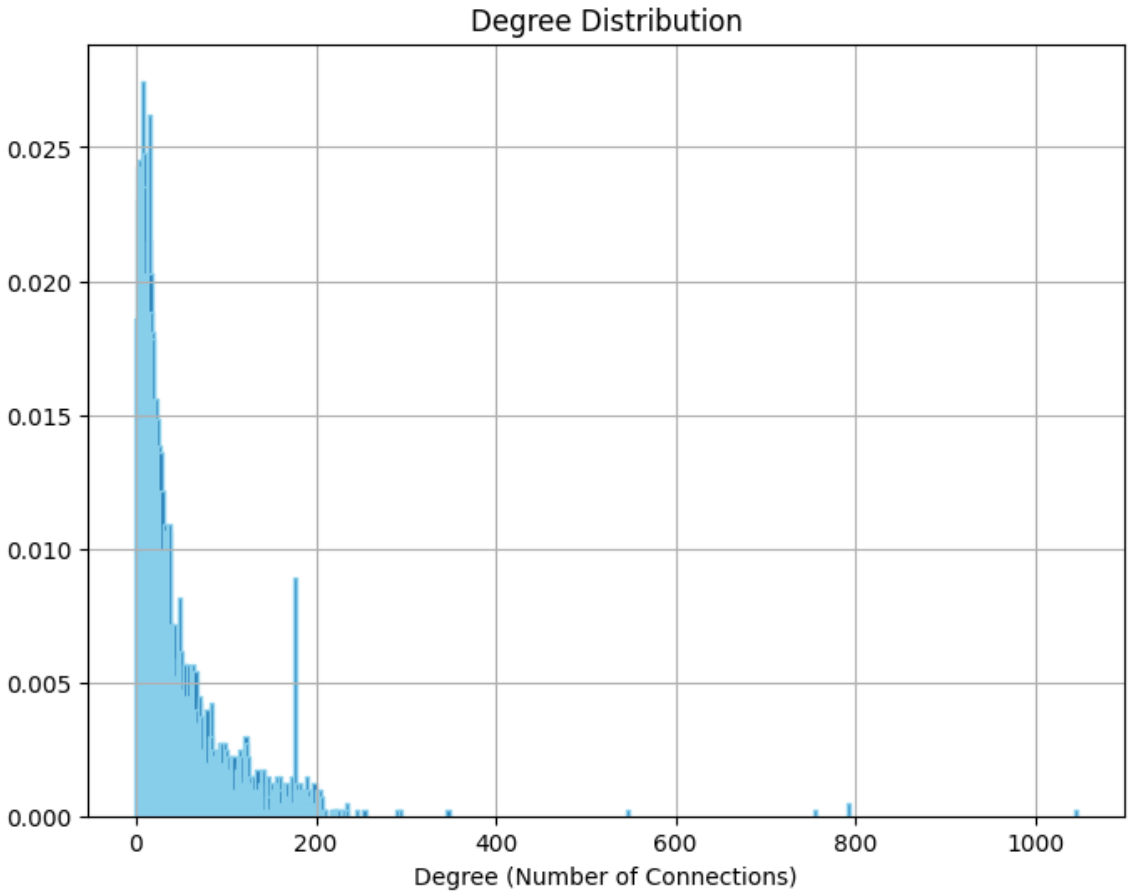
Outliers:
Nodes with significantly higher degrees (hubs) are visible as separate peaks or outliers.

Network Density:
If the histogram shows that most nodes have a degree close to the average degree, the network is relatively dense.

degree distribution

```
]:
```

```python
# Calculate the degrees of all nodes
degrees = G.degree()

# Normalize the degree distribution by dividing frequency by the total number of nodes
total_nodes = G.vcount()  # Correct way to get the number of nodes in igraph
degree_counts = [degrees.count(degree) for degree in set(degrees)]  # Count frequencies of each degree
unique_degrees = sorted(set(degrees))  # Unique degree values

# Normalize the frequencies
normalized_counts = [count / total_nodes for count in degree_counts]

#bar_width = 1.0 / len(unique_degrees)  # Dynamic width based on number of unique degrees

# Plot the degree distribution (normalized)
plt.figure(figsize=(8, 6))
plt.bar(unique_degrees, normalized_counts, width=4, edgecolor='skyblue', alpha=1)
plt.title("Degree Distribution")
plt.xlabel("Degree (Number of Connections)")
plt.grid(True)
plt.show()

```



5- Adjacency and Edge List

An **adjacency list** is a representation of a graph where each node (vertex) is associated with a list of its connected neighbors. This structure is efficient for sparse graphs as it only stores existing edges, saving memory compared to an adjacency matrix. It allows for quick access to the neighbors of a node.

Method used: G.get_adjlist(mode='ALL')

Then sort the list based on the number of neighbors

output to print the top 10 nodes having max num of neighbors

```
# Create adjacency list directly using igraph's built-in method
adjacency_list = G.get_adjlist(mode='ALL')

# Sort the adjacency list by the number of neighbors (degree of nodes)
sorted_adjacency_list = sorted(enumerate(adjacency_list), key=lambda x: len(x[1]), reverse=True)

# Get the top 10 nodes with the most neighbors (degree)
top_10_nodes = sorted_adjacency_list[:10]

# Convert to DataFrame for easy visualization
df = pd.DataFrame(top_10_nodes, columns=["Node", "Neighbors"])
df["Degree"] = df["Neighbors"].apply(len)  # Add a column for the degree (number of neighbors)

# Print the table
df
```

✓ 0.0s

	Node	Neighbors	Degree
0	107	[0, 58, 171, 348, 353, 363, 366, 376, 389, 414...	1045
1	1684	[58, 107, 171, 860, 990, 1171, 1405, 1419, 145...	792
2	1912	[58, 136, 428, 563, 1465, 1577, 1718, 1913, 19...	755

An edge list is a representation of a graph where each edge is explicitly listed as a pair (or tuple) of nodes connected by that edge. The output you've provided shows the edges of a graph G in the form of pairs of node indices.

Method used: G.get_edgelist() retrieves the list of edges in Graph G

- The output [(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), ...] indicates that:
- Node 0 is connected to nodes 1, 2, 3, 4, 5, and 6,..,etc.

6- Graph Density:

Graph density represents the ratio between the edges present in a graph and the maximum number of edges that the graph can contain.

The graph said to be dense graph if the number of edges is close to the maximum possible number of edges.

The density value of 0.0108 that obtained indicates that the graph is very sparse.

```
# Calculate the density of the graph
density = G.density()

# Print the density of the graph
print(f"Graph Density = {density}")
```

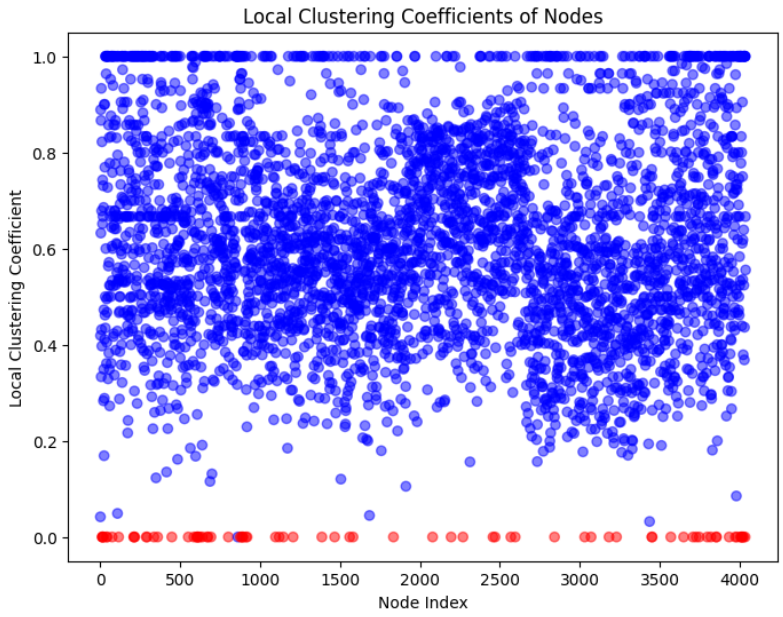
✓ 0.0s

```
Graph Density = 0.010819963503439287
```

7- Clustering Coefficients

1. **Local clustering coefficients**
   Measures the clustering around a specific node.

Insights:



1. most of nodes have clustering coefficients between [0.4: 0.8]
2. Red color identify that nodes have one neighbor (degree =1)
3. So clustering coefficients will be = 2*0/1(1-1) = inf or (0)
4. Number of nodes with a clustering coefficient of 1: 267
5. Number of nodes with a clustering coefficient of 0: 76
6. Node 860 has a clustering coefficient of 0 and degree of 2

then calculate the AVG clustering coefficient = 0.6

```
# Calculate the average local clustering coefficient
average_local_clustering_coeff = sum(local_clustering_coeffs) / G.vcount()

# Print the average local clustering coefficient
print(f"Average Local Clustering Coefficient = {average_local_clustering_coeff}")
```

✓ 0.0s

```
Average Local Clustering Coefficient = 0.6055467186200871
```

2-Global clustering coefficients

Measures the overall clustering of the graph.

```
# Calculate the global clustering coefficient (transitivity)
global_clustering = G.transitivity_undirected()

print(f"Global Clustering Coefficient (Transitivity): {global_clustering}")
```

✓ 0.0s

```
Global Clustering Coefficient (Transitivity): 0.5191742775433075
```

A Global Clustering Coefficient (Transitivity) of 0.519 suggests that the network consists of clusters or tightly-knit groups (cliques or communities), but the entire network is not a single unified community.

8- Shortest path

1- Diameter:

We found that path from 687 to 3981 is the diameter.

Longest Shortest Path (Diameter)

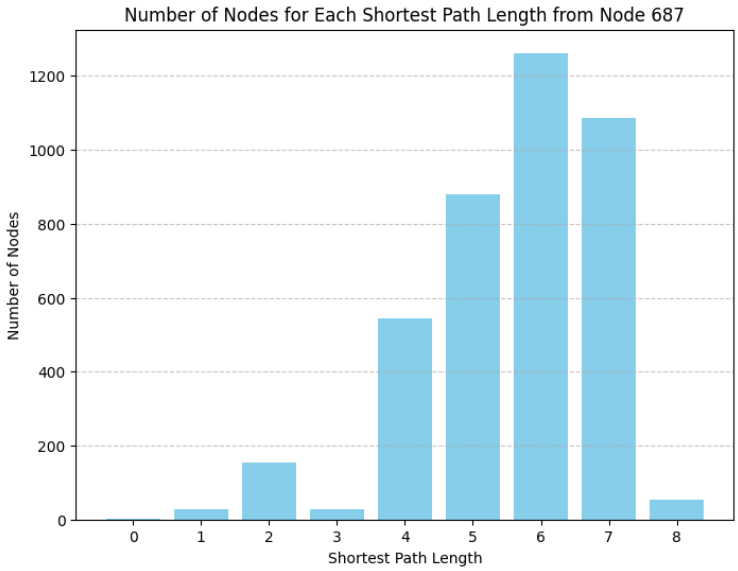
```
# Calculate the diameter
diameter_length = G.diameter(directed=False)
print(f"The diameter of the graph is: {diameter_length}")

# Get the nodes involved in the longest shortest path
diameter_path = G.get_diameter(directed=False)
print(f"The longest shortest path (diameter path) is: {diameter_path}")
```

✓ 1.4s

```
The diameter of the graph is: 8
The longest shortest path (diameter path) is: [687, 686, 698, 3437, 567, 414, 594, 3980, 3981]
```

2- Node shortest path:

Node 687 shortest path form most of nodes is 6



3.AVG shortest path:

AVG shortest path

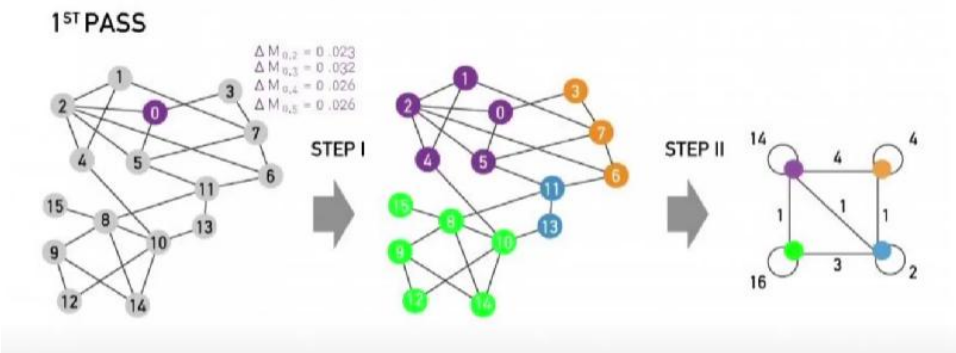
```
# Calculate the shortest path lengths between all pairs of nodes (distance matrix)
path_lengths = G.distances()

# Flatten the matrix
finite_lengths = [length for row in path_lengths for length in row]

E_max = G.vcount() * (G.vcount()-1) // 2

# Compute the average shortest path length
avg_shortest_path_length =( 1 /(2*E_max)) * (sum(finite_lengths))

print("Average Shortest Path Length:", avg_shortest_path_length)
```

```
Average Shortest Path Length: 3.692506849696391
```

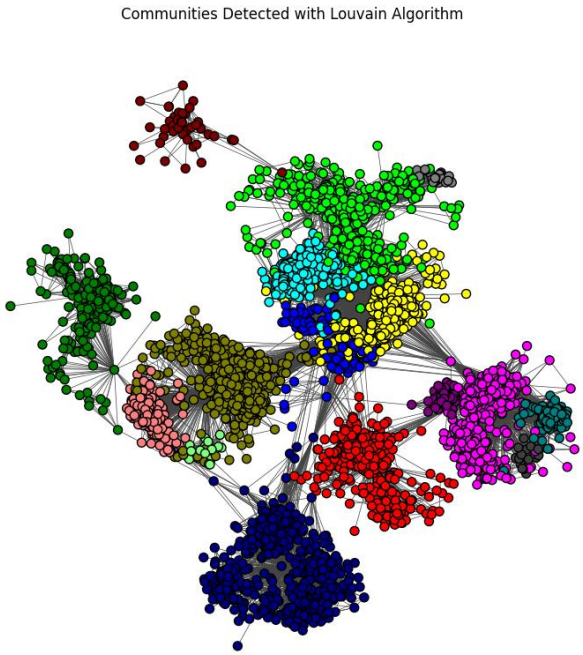
calculating the average shortest path using rule: $\frac{1}{2E\_max}\sum_{i\neq j} hij$

9- Community Detection

Using The Louvain algorithm is a widely used method for detecting communities in networks. It maximizes modularity, identifying groups of nodes that are more densely connected to each other than to the rest of the network. The algorithm operates in iterative phases to optimize community structure.

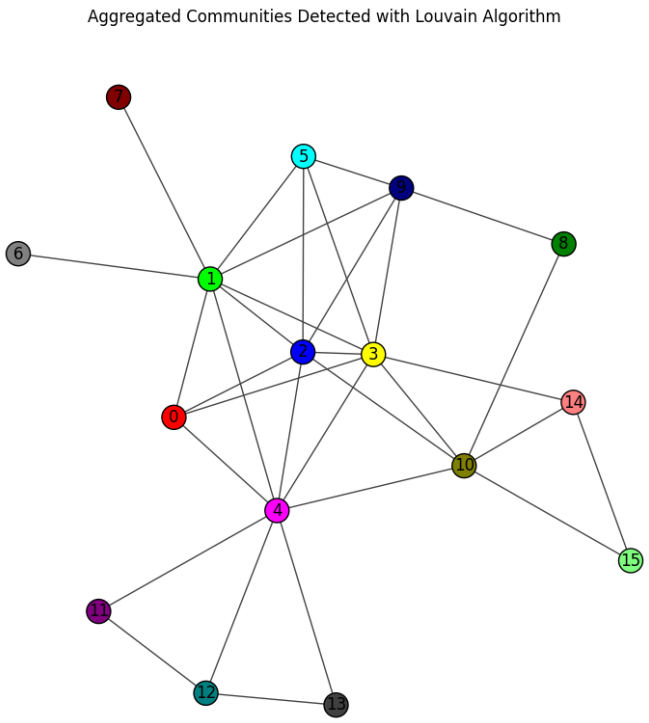


Louvain algorithm identify the network into 16 Community [0-15]



We can aggregate the communities into one node to understand the relationship between communities better.

- We notice that node 4 high have Betweenness (39). Removing it will make the graph disconnected
- Node 6, 7 connected only with node 1
- Node 3,4 have highest closeness



10 - Centrality Measures

1. Degree Centrality

Measures the number of connections (edges) a node has.

1. Centrality Measures

```
# Calculate degree centrality
degree_centrality = G.degree()

# Create a list of (node, degree centrality) pairs
degree_centrality_pairs = [(node, degree) for node, degree in enumerate(degree_centrality)]

# Sort nodes by degree centrality in descending order
sorted_degree_centrality = sorted(degree_centrality_pairs, key=lambda x: x[1], reverse=True)

# Print the top 4 nodes with the highest degree centrality
print("Top 4 Nodes by Degree Centrality:")
for i in range(4):
    node, centrality = sorted_degree_centrality[i]
    print(f"Node {node}: Degree Centrality {centrality}")
```

```
Top 4 Nodes by Degree Centrality:
Node 107: Degree Centrality 1045
Node 1684: Degree Centrality 792
Node 1912: Degree Centrality 755
Node 3437: Degree Centrality 547
```

The degree of each node is calculated, sorted in descending order, and the top 4 nodes with the highest degrees are reported.

(Node 107 has the highest degree centrality of 1045, indicating it is the most directly connected node in the network.)

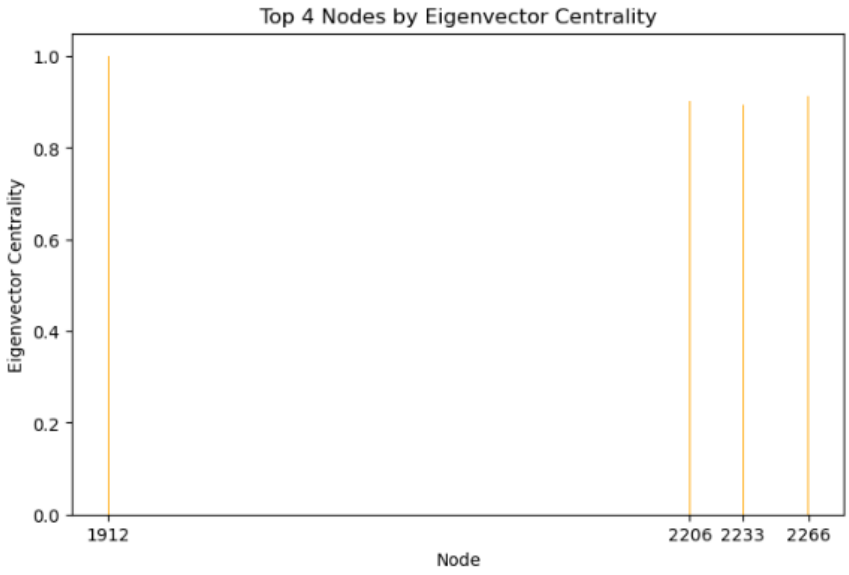
2. Eigenvector Centrality

2. Eigenvector Centrality

```
# Calculate eigenvector centrality
eigenvector_centrality = G.eigenvector_centrality()

# Create a list of (node, eigenvector centrality) pairs
eigenvector_centrality_pairs = [(node, eig) for node, eig in enumerate(eigenvector_centrality)]

# Sort nodes by eigenvector centrality in descending order
sorted_eigenvector_centrality = sorted(eigenvector_centrality_pairs, key=lambda x: x[1], reverse=True)

# Print the top 4 nodes with the highest eigenvector centrality
print("Top 4 Nodes by Eigenvector Centrality:")
for i in range(4):
    node, centrality = sorted_eigenvector_centrality[i]
    print(f"Node {node}: Eigenvector Centrality {centrality}")
```

```
Top 4 Nodes by Eigenvector Centrality:
Node 1912: Eigenvector Centrality 1.0
Node 2266: Eigenvector Centrality 0.9117190175727701
Node 2206: Eigenvector Centrality 0.9019626322184237
Node 2233: Eigenvector Centrality 0.8927488203018584
```

Eigenvector centrality is computed, sorted, and the top 4 nodes are presented.



(Node 1912 has the highest eigenvector centrality (1.0), indicating it is connected to other influential nodes.)

3. **Betweenness Centrality**

Measures how often a node appears on the shortest paths between other nodes

3. Btweenness Centrality

```
]: # Calculate betweenness centrality
betweenness_centrality = G.betweenness()

# Create a list of (node, betweenness centrality) pairs
betweenness_centrality_pairs = [(node, bet) for node, bet in enumerate(betweenness_centrality)]

# Sort nodes by betweenness centrality in descending order
sorted_betweenness_centrality = sorted(betweenness_centrality_pairs, key=lambda x: x[1], reverse=True)
```

```
]: # Print the top 4 nodes with the highest betweenness centrality
print("Top 4 Nodes by Betweenness Centrality:")
top_nodes = sorted_betweenness_centrality[:4]
for node, centrality in top_nodes:
    print(f"Node {node}: Betweenness Centrality {centrality}")
```

```
Top 4 Nodes by Betweenness Centrality:
Node 107: Betweenness Centrality 3916560.1444407436
Node 1684: Betweenness Centrality 2753286.6869082903
Node 3437: Betweenness Centrality 1924506.1515714861
Node 1912: Betweenness Centrality 1868918.2122567892
```

Betweenness centrality is computed, sorted, and the top 4 nodes are displayed.

(Node 107 has the highest betweenness centrality, indicating it is crucial for connecting different parts of the network.)

4. Closeness Centrality:

Measures how quickly a node can reach all other nodes in the network.

4. Closeness Centrality

```
# Calculate closeness centrality
closeness_centrality = G.closeness()

# Create a list of (node, closeness centrality) pairs
closeness_centrality_pairs = [(node, close) for node, close in enumerate(closeness_centrality)]

# Sort nodes by closeness centrality in descending order
sorted_closeness_centrality = sorted(closeness_centrality_pairs, key=lambda x: x[1], reverse=True)

# Print the top 4 nodes with the highest closeness centrality
print("Top 4 Nodes by Closeness Centrality:")
for i in range(4):
    node, centrality = sorted_closeness_centrality[i]
    print(f"Node {node}: Closeness Centrality {centrality}")
```

```
Top 4 Nodes by Closeness Centrality:
Node 107: Closeness Centrality 0.45969945355191255
Node 58: Closeness Centrality 0.3974018305284913
Node 428: Closeness Centrality 0.3948371956585509
Node 563: Closeness Centrality 0.3939127889961955
```

Closeness centrality is calculated, sorted, and the top 4 nodes are shown.

(Node 107 has the highest closeness centrality, meaning it can efficiently interact with all other nodes.)

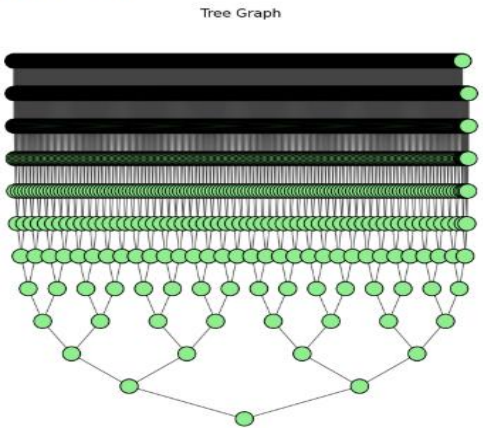
Special graphs :

1. Tree

Checking for a Tree:

- A tree must satisfy the properties of being connected and acyclic, with **edges = nodes – 1**
- If the dataset graph is not a tree, we generate a sample tree graph for comparison.
- The graph was determined **not to be a tree**, and a binary tree visualization was generated for comparison.

```
In [12]: # ----- Check for a Tree -----
         # A tree must be connected and have edges = nodes - 1
         is_tree = G.is_connected() and G.ecount() == G.vcount() - 1
         print(f"Is the graph a tree? {is_tree}")

         if not is_tree:
             # Generate a sample tree for comparison
             tree = Graph.Tree(n=G.vcount(), children=2)  # Binary tree layout
             layout = tree.layout("tree")  # Tree layout

             # Visualize the tree
         plt.figure(figsize=(8, 8))
         plot(tree, layout=layout, target=plt.gca(), vertex_size=20, vertex_color='lightgreen',
                 edge_width=0.8, bbox=(0, 0, 1, 1), margin=50)
         plt.title("Tree Graph")
         plt.show()

         Is the graph a tree? False
```



1. Cycle Check:

- Cycles naturally occur in social networks where groups of friends form loops.
- We identified 3-node cycles (triangles) in the dataset as a simple example.
- We then visualize one of the cycles.