VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Social Network Analysis LAB

Assessment - 2 REPORT

Faculty Name: Durgesh Kumar

Student Name: Sarthak Tripathi
Reg. No.: 22MCB0030
E-mail: sarthak.tripathi2022@vitstudent.ac.in
Mobile No.: 8630054046

Table of Contents

Introduction

While humans are very good at detecting distinct or repetitive patterns among a few components, the nature of large interconnected networks makes it practically impossible to perform such basic tasks manually. Groups of densely connected nodes are easy to spot visually, but more sophisticated methods are needed to perform these tasks programmatically. Community detection algorithms are used to find such groups of densely connected components in various networks.

Many of you are familiar with networks, right? You might be using social media sites such as Facebook, Instagram, Twitter, etc. They are social networks. You might be dealing with stock exchanges. Either you might be buying new stocks, selling what you already have, etc. They are networks. Not only in the technological field but also in our day to day social life, we deal with many networks. Communities are a property of many networks in which a particular network may have multiple communities such that nodes inside a community are densely connected. Nodes in multiple communities can overlap. Think of your Facebook or Instagram account and consider who you interact with daily. You might be heavily interacting with your friends, colleagues, family members and a few other important people in your life. They form a very dense community inside your social network.

M. Girvan and M. E. J. Newman are two popular researchers in the domain of community detection. In one of their research, they have highlighted the community structure-property using social networks and biological networks. According to them, network nodes are tightly connected in knit groups within communities and loosely connected between communities.

Why Community Detection?

When analysing different networks, it may be important to discover communities inside them. Community detection techniques are useful for social media algorithms to discover people with common interests and keep them tightly connected. Community detection can be used in machine learning to detect groups with similar properties and extract groups for various reasons. For example, this technique can be used to discover manipulative groups inside a social network or a stock market.

Community Detection vs Clustering

One can argue that community detection is similar to clustering. Clustering is a machine learning technique in which similar data points are grouped into the same cluster based on their attributes. Even though clustering can be applied to networks, it is a broader field in unsupervised machine learning which deals with multiple attribute types. On the other hand, community detection is specially tailored for network analysis which depends on a single

attribute type called edges. Also, clustering algorithms have a tendency to separate single peripheral nodes from the communities it should belong to. However, both clustering and community detection techniques can be applied to many network analysis problems and may raise different pros and cons depending on the domain.

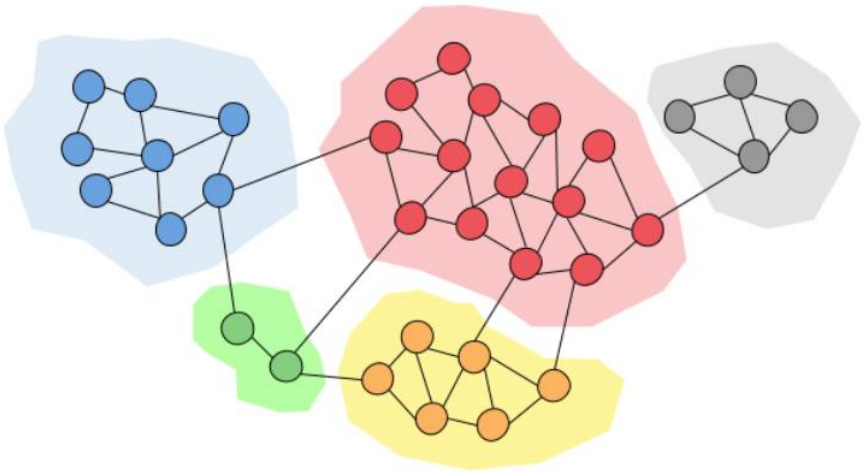


Fig. 1 Clustering vs Community Detection

Community Detection Example Applications

Because networks are an integral part of many real-world problems, community detection algorithms have found their way into various fields, ranging from social network analysis to public health initiatives.

- A known use case is the detection of terrorist groups in social networks by tracking their activities and interactions. Similarly, groups of malicious bots can be detected on online social platforms.
- Community detection can be used to study the dynamics of certain groups that are susceptible to epidemic diseases. Other types of diseases can be studied similarly to discover common links among patients.
- One of the most recent use cases, community evolution prediction, involves the prediction of changes in network structures.

The concept of community detection has emerged in network science as a method for finding groups within complex systems through represented on a graph. In contrast to more traditional decomposition methods which seek a strict block diagonal or block triangular structure, community detection methods find subnetworks with statistically significantly more links between nodes in the same group than nodes in different groups (Girvan and Newman, 2002). Central to community detection is the notion of modularity, a metric that captures this difference:

$$Q = \frac{1}{2m}\sum_{i,j} A_{\text{ij}} - \frac{k_i k_j}{2m}\delta g_i g_j$$

Here, Q is the modularity, Aij is the edge weight between nodes i and j, ki is the total weight of all edges connecting node i with all other nodes, and m is the total weight of all edges in the graph. The Kronecker delta function δ(gi,gj) will evaluate to one if nodes i and j belong to the same group, and zero otherwise. Modularity is a property of how one decides to partition a network: networks that are not partitioned and those that place every node in its own community will both have modularity equal to zero. The goal of community detection, then, is to find communities that maximize modularity. Although modularity maximization is an NP-hard integer program, many efficient algorithms exist to solve it approximately, including spectral clustering (Newman, 2006) and fast unfolding (Blondel et al., 2008). Below Figure shows a few networks with increasing maximum modularity; note how the community structure becomes increasingly apparent as this value increases. Previous efforts in our group have applied community detection to chemical plant networks by creating an equation graph of the corresponding dynamic model (Moharrir et al., 2017). By doing so, communities of state variables, inputs, and outputs can be obtained which are tightly interacting amongst themselves but weakly interacting with other communities. As such, these communities can form the basis of distributed control architectures (Jogwar and Daoutidis, 2017) which typically perform better than other distributed control architectures that one may obtain from "intuition" (Pourkargar et al., 2017). For a more comprehensive review of the use of community detection in distributed control, we refer the reader to Daoutidis et al., 2017. An alternative method for finding communities for distributed model predictive control is to apply a decomposition on the optimization problem as a whole (Tang et al., 2017).

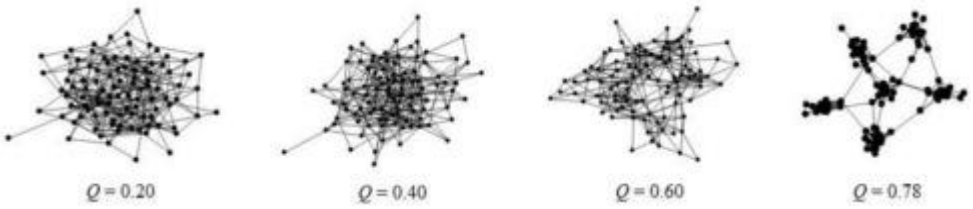


Fig. 2 increasing maximum modularity

However, community structure can exist in all optimization problems, thus it makes sense to extend this method to any generic optimization problem, and this work proposes to do so. The advantage of using community detection to find decompositions is that subproblems generated will have statistically minimal interactions, through complicating variables or constraints, and thus require minimal coordination through the decomposition solution method. The proposed method is generic, applicable to any optimization problem or decomposition solution approach, and scalable, using computationally efficient graph theory algorithms.

Graph clustering algorithms

Clustering (also known as community detection in the context of graphs) methods for graphs/networks are designed to locate communities based on the network topology, such as tightly connected groups of nodes. Before performing the community detection algorithm in graph-based clustering, the data is represented by a graph. Edge weight in the graph can be calculated using a variety of similarity measures. A community is formed when nodes in a network are of the same type. Intra-community edges are the edges that connect the nodes

within a community. Inter-community edges are nodes that connect nodes from different communities. In Fig. 3, a small graph depicts intra-community and inter-community edges between distinct communities. The following sections explain two prominent graph-based clustering algorithms.

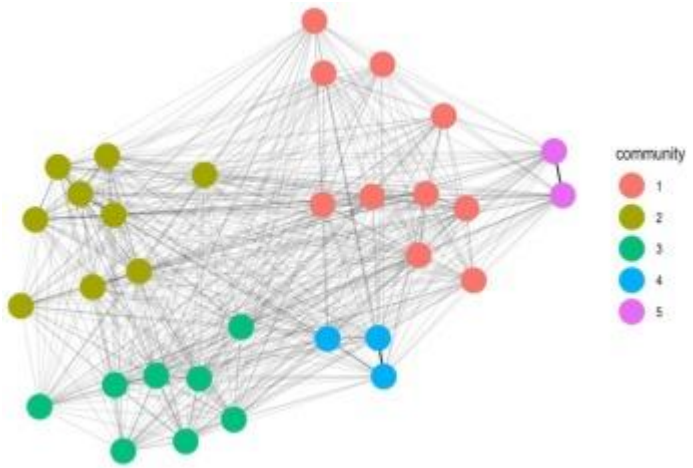


Fig. 3 Community structure in a graph showing intra-community edges and inter-community edges.

The researchers utilized a variety of community discovery tools, including the Louvain algorithm and Infomap.

1) Louvain method: Blondel, Lambiotte, and Lefebvre (2008) of the University of Louvain created the Louvain technique for community detection, which is a method for detecting functioning communities from massive networks. The Louvain community detection method is made up of two processes that repeat themselves. The first phase is a "greedy" task of assigning nodes to communities, with a preference for neighborhood improvements. The second stage is a coarse-grained reorganization of the networks discovered in the first step. These two stages are performed until there are no more measured quality expanding network reassignments possible.

   The Louvain strategy optimizes for "modularity" in the same way as previous methods, but in a fraction of the time, allowing for the analysis of far bigger systems. In this method, modularity is defined as a ratio of the density of edges inside communities to the density of edges outside communities in the range of [-1,1].

   Heuristic techniques are employed to maximize this modularity value so that the best possible groupings of nodes in a network can theoretically be produced. Small communities are first discovered by locally maximizing the modularity value, after which each microcommunity is grouped into a single node and the process is repeated.

2) Infomap method: Rosvall and Bergstrom's Infomap is another famous community discovery algorithm (2008). It improves the map equation, which achieves a balance between the difficulty of data compression and the problem of recognizing and retrieving essential patterns or structures within the data. The Louvain community

finding approach is closely followed by this algorithm. Each node is assigned to its own module at first. The nodes are then transferred to neighboring modules in a random sequential manner, resulting in the greatest reduction of the map equation. If no move leads to a further reduction of the map equation, the node remains in its original place. The method is done until the map equation can no longer be decreased in a different random sequential order each time. Finally, the network is rebuilt, with the lowest-level modules producing nodes at that level and nodes being joined into modules in the same way that previous-level nodes were. This hierarchical network rebuilding is continued until the map equation can no longer be lowered. Both undirected and directed graphs can be used with Infomap. Infomap returns a list of non-overlapping communities (node-set partitions) as well as a decimal value that represents the total flow in that node.

Thus, there are a wide variety of clustering algorithms, as is evident from the discussion above. It is a natural question as to which algorithm performs the best on a certain dataset. To answer this question, first some measures for cluster validation/evaluation need to be decided. The next section discusses some commonly used cluster validation measures.

Community Detection Techniques

Community detection methods can be broadly categorized into two types; **Agglomerative Methods and Divisive Methods**. In Agglomerative methods, edges are added one by one to a graph which only contains nodes. Edges are added from the stronger edge to the weaker edge. Divisive methods follow the opposite of agglomerative methods. In there, edges are removed one by one from a complete graph.

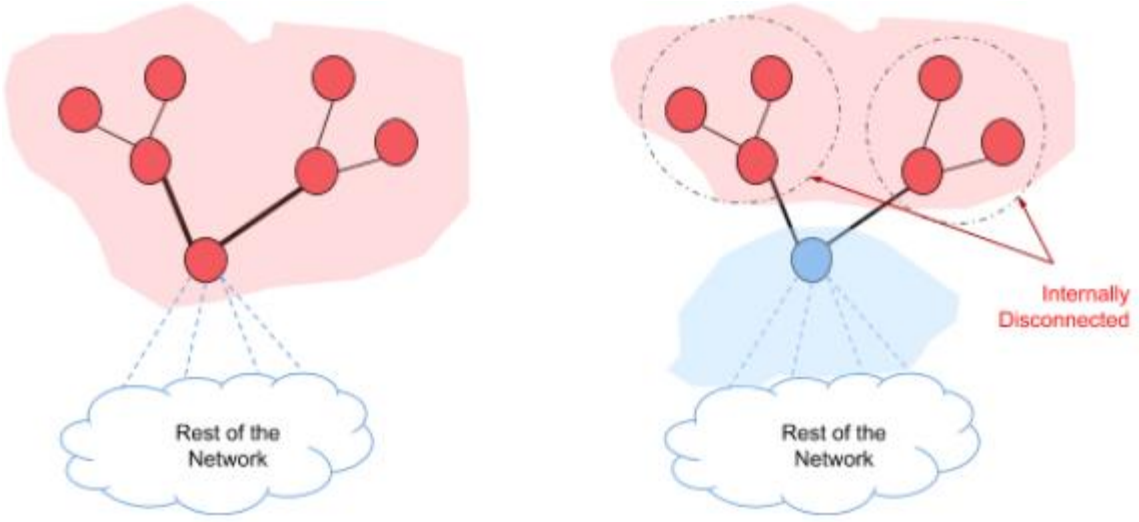
There can be any number of communities in a given network and they can be of varying sizes. These characteristics make the detection procedure of communities very hard. However, there are many different techniques proposed in the domain of community detection. Four popular community detection algorithms are explained below. All of these listed algorithms can be found in the python cdlib library.

1) **Agglomerative methods** generally start with a network that contains only nodes of the original graph. The edges are added one-by-one to the graph, while stronger edges are prioritized over weaker ones. The strength of an edge, or weight, is calculated differently depending on the specific algorithm implementation.

2) On the other hand, **divisive methods** rely on the process of removing edges from the original graph iteratively. Stronger edges are removed before weaker ones. At every step, the edge-weight calculation is repeated, since the weight of the remaining edges changes after an edge is removed. After a certain number of steps, we get clusters of densely connected nodes, a.k.a. communities.

Leiden Community Detection

In later research (2019), V.A. Traag et al. showed that Louvain community detection has a tendency to discover communities that are internally disconnected (badly connected communities). In the Louvain algorithm, moving a node which has acted as a bridge between two components in a community to a new community may disconnect the old community. This won't be a problem if the old community is being further split. But according to Traag et al., this won't be the case. Other nodes in the old community allow it to remain as a single community due to their strong connections. Also according to them, Louvain has a tendency to discover very weekly connected communities. Therefore, they have proposed the much faster Leiden algorithm which guarantees that communities are well connected.



Additionally to the phases used in Louvain algorithm, Leiden uses one more phase which tries to refine the discovered partitions. The three-phases in Leiden algorithm are:

- Local moving of nodes
- Refinement of the partitions
- Aggregation of the network based on refined partitions

In the refinement phase, the algorithm tries to identify refined partitions from the partitions proposed by the first phase. Communities proposed by the first phase may further split into multiple partitions in the second phase. The refinement phase does not follow a greedy approach and may merge a node with a randomly chosen community which increases the quality function. This randomness allows discovering the partition space more broadly. Also in the first phase, Leiden follows a different approach to the Louvain. Instead of visiting all the nodes in the network after the first visit to all nodes has been completed, Leiden only visits those nodes whose neighbourhood has changed.

Louvain's Algorithm

What is Louvain's Algorithm?

Louvain's algorithm, named after the University of Louvain by professor Vincent Blondel et al. in 2008. The algorithm originated from their paper "Fast unfolding of communities in large networks" where they introduced a greedy method which would generate communities in O(n*log(n)) time where n is the number of nodes in the original network.

Modularity

Louvain's algorithm aims at optimizing modularity. Modularity is a score between -0.5 and 1 which indicates the density of edges within communities with respect to edges outside communities. The closer the modularity is to -0.5 implies non modular clustering and the closer it is to 1 implies fully modular clustering. Optimizing this score yields the best possible groups of nodes given a network. Intuitively, you can interpret it as maximizing the difference between the actual number of edges and the expected number of edges. Modularity can be defined by the following formula:

$$Q = \frac{1}{2m} \sum_{ij} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j)$$

- Ai,j represents the edges between nodes i and j
- m is the sum of all edge weights in the network
- delta is the Kronecker delta function
  - delta = 1 if i =j
  - delta = 0 otherwise
- Ci and Cj are the communities of the nodes
- Ki and Kj is the sum of weights connecting nodes i and j

Louvain's Algorithm

To maximize the modularity, Louvain's algorithm has two iterative phases. The first phase assigns each node in the network to its own community. Then it tries to maximize modularity gain by merging communities together. Node i would be potentially merged with node j (the neighbour of i) to determine if there would be a positive increase in the modularity. If there is no modular increase then the node remains in its current community, otherwise the communities are merged. This merging of communities can be represented by the following formula:

$$\Delta Q = \left[\frac{\Sigma_{in} + 2k_{i,in}}{2m} - \left(\frac{\Sigma_{tot} + k_i}{2m}\right)^2\right] - \left[\frac{\Sigma_{in}}{2m} - \left(\frac{\Sigma_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right]$$

Where Sigma_in is sum of all the weights of the links inside the community i is moving into, Sigma_tot is the sum of all the weights of the links to nodes in the community i is moving into, k_i is the weighted degree of i, k_i,in is the sum of the weights of the links between i and other nodes in the community that i is moving into, and m is the sum of the weights of all links in the network.

Girvan-Newman Algorithm

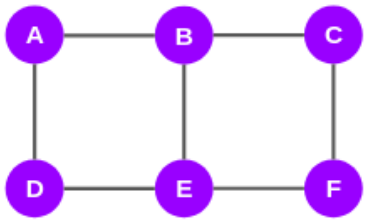
Under the Girvan-Newman algorithm, the communities in a graph are discovered by iteratively removing the edges of the graph, based on the edge betweenness centrality value.

The edge with the highest edge betweenness is removed first. We will cover this algorithm later in the article, but first, let's understand the concept of "edge betweenness centrality".

Edge Betweenness Centrality (EBC)

The edge betweenness centrality (EBC) can be defined as the number of shortest paths that pass through an edge in a network. Each and every edge is given an EBC score based on the shortest paths among all the nodes in the graph.

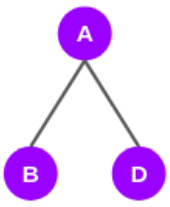
With respect to graphs and networks, the shortest path means the path between any two nodes covering the least amount of distance. Let's take an example to find how EBC scores are calculated. Consider this graph below:



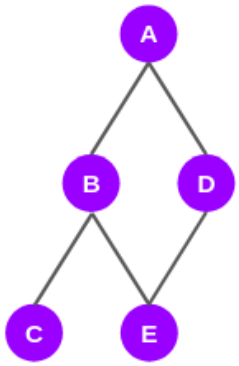
It has 6 nodes and 7 edges, right? Now, we will try to find the EBC scores for all the edges in this graph. Note that it is an iterative process and I've given an outline of it here:

- We will take one node at a time and plot the shortest paths to the other nodes from the selected node
- Based on the shortest paths, we will compute the EBC scores for all the edges
- We need to repeat this process for every node in the graph. As you can see, we have 6 nodes in the graph above. Therefore, there will be 6 iterations of this process
- This means every edge will get 6 scores. These scores will be added edge-wise
- Finally, the total score of each edge will be divided by 2 to get the EBC score

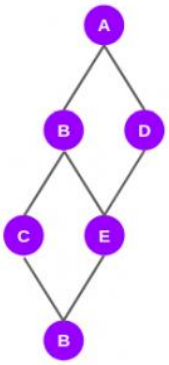
Now, let's start with node A. The directly connected nodes to node A are nodes B and D. So, the shortest paths to B and D from A are AB and AD respectively:



It turns out that the shortest paths to nodes C and E from A go through B and D:



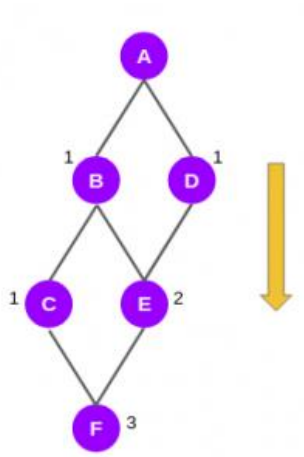
The shortest paths to the last node F from node A, pass through nodes B, D, C, and E:



The graph above depicts only the shortest paths from node A to all the other nodes. Now we will see how edges are scored.

Before giving scores to the edges, we will assign a score to the nodes in the shortest-path-graph. To assign these scores, we will have to traverse the graph from the root node, i.e., node A to the last node (node F).

Assigning Scores to Nodes



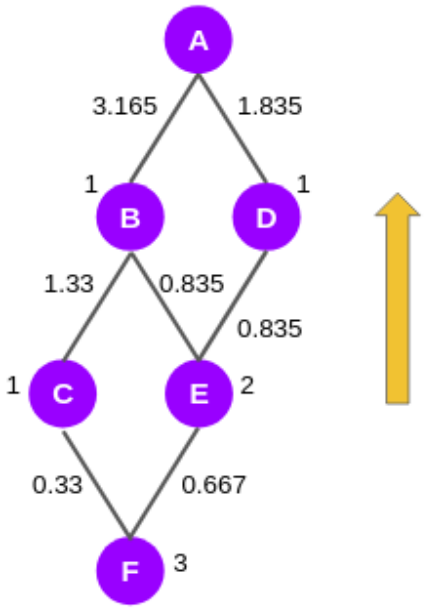
As you can see in the graph above, nodes B and D have been given a score of 1 each. This is because the shortest path to either node from node A is only one. For the very same reason, node C has been given a score of 1 as there is only one shortest path from node A to node C.

Moving on to node E. It is connected to node A through two shortest paths, ABE and ADE. Hence, it gets a score of 2.

The last node F is connected to A through three shortest paths — ABCF, ABEF, and ADEF. So, it gets a score of 3.

Computing Scores for Edges

Next, we will proceed with computing scores for the edges. Here we will move in the backward direction, from node F to node A:



FC = ⅓ = 0.33
FE = ⅔ = 0.667

CB = 1 + 0.33 = 1.33
EB = (1 + 0.667)/2 = 0.835
ED = (1 + 0.667)/2 = 0.835

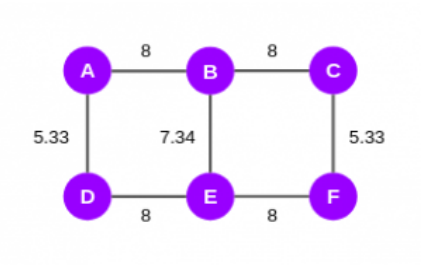
BA = (1 + 1.33 + 0.835)/1 = 3.165
DA = (1 + 0.835)/1 = 1.835

We first compute the score for the edges FC and FE. As you can see, the edge score for edge FC is the ratio of the node scores of C and F, i.e. 1/3 or 0.33. Similarly, for FE the edge score is 2/3.

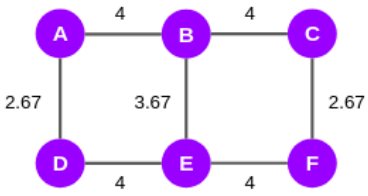
Now we have to calculate the edge score for the edges CB, EB, and ED. According to the Girvan-Newman algorithm, from this level onwards, every node will have a default value of 1 and the edge scores computed in the previous step will be added to this value.

So, the edge score of CB is (1 + 0.33)/1. Similarly, edge score EB or ED is (1 + 0.667)/2. Then we move to the next level to calculate the edge scores for BA and DA.

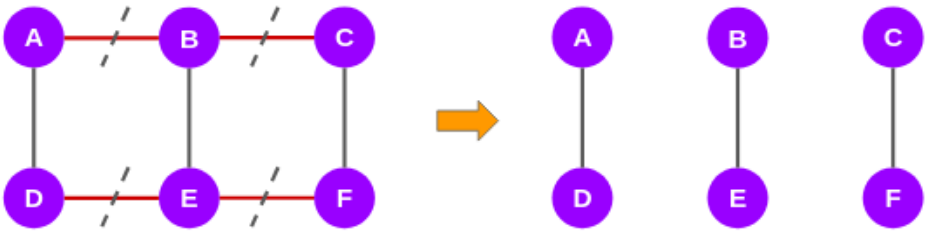
So far, we have computed the edge scores of the shortest paths with respect to node A. We will have to repeat the same steps again from the other remaining five nodes. In the end, we will get a set of six scores for all the edges in the network. We will add these scores and assign them to the original graph as shown below:



Since it is an undirected graph, we will divide these scores by two and finally, we will get the EBC scores:



According to the Girvan-Newman algorithm, after computing the EBC scores, the edges with the highest scores will be taken off till the point the graph splits into two. So, in the graph above, we can see that the edges AB, BC, DE, and EF have the highest score, i.e., 4. We will strike off these edges and it gives us 3 subgraphs that we can call communities:



Implementation of Louvain Algorithm

Importing necessary libraries

```
[1]  import random
     import networkx as nx
     import numpy as np
     from community import community_louvain
     import matplotlib.pyplot as plt
```

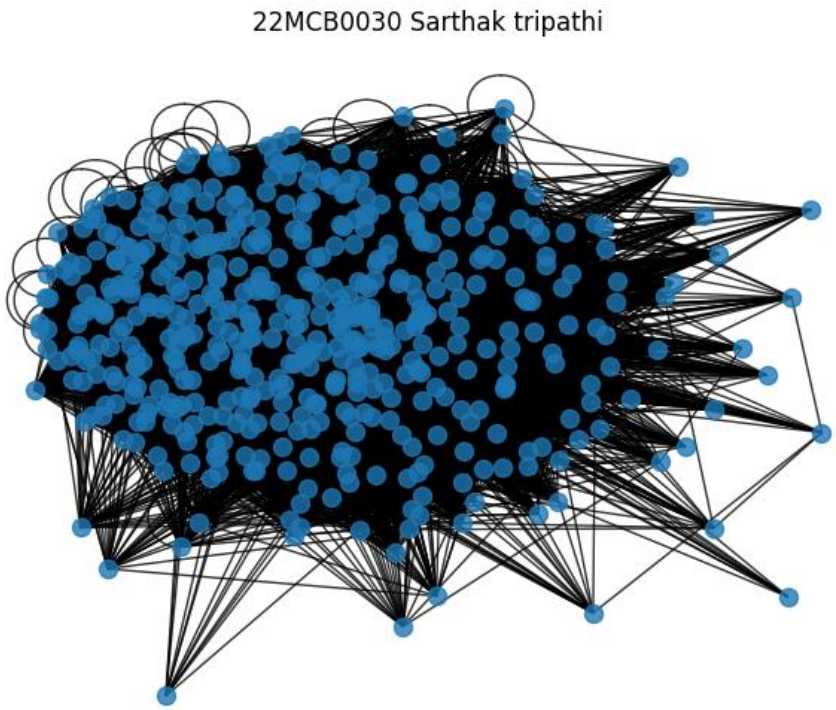
```
def generate_network(n=10):
    '''
    This function will generate a random weighted network associated to the user specifed
    number of nodes.

    params:
        n (Integer) : The number of nodes you want in your network

    returns:
        A networkX multi-graph

    example:
        G = generate_network(n)
    '''
    # initialize dictionary with nodes
    graph_dct = {node:[] for node in range(n)}
    nodes = list(range(n))

    # generate edges
    for n,edge_list in graph_dct.items():
        edge_c = random.randint(min(nodes), int(max(nodes) / 2))
        el = random.sample(nodes, edge_c)
        graph_dct[n] = el

    # create networkx multi-edge graph
    G = nx.MultiGraph(graph_dct)
    return G

n = 500
G = generate_network(n)
print("Number of nodes:", G.number_of_nodes())
print("Number of edges:", G.number_of_edges())

# visualize graph
pos = nx.spring_layout(G)
nx.draw(G, pos, node_size = 75, alpha = 0.8)
plt.show()
```

The following function above generates a network associated with a randomly generated degree distribution. The user can specify the number of nodes they would like in the associated

network, for the purposes of this tutorial I selected 50. The following is the network statistics from the sample network I created.

```
Number of nodes: 500
Number of edges: 32059
```

The following is a visual representation of how the network looks.



Apply Louvain's Algorithm

We can simply apply Louvain's algorithm through the Python-Louvain module.

```
comms = community_louvain.best_partition(G)
```

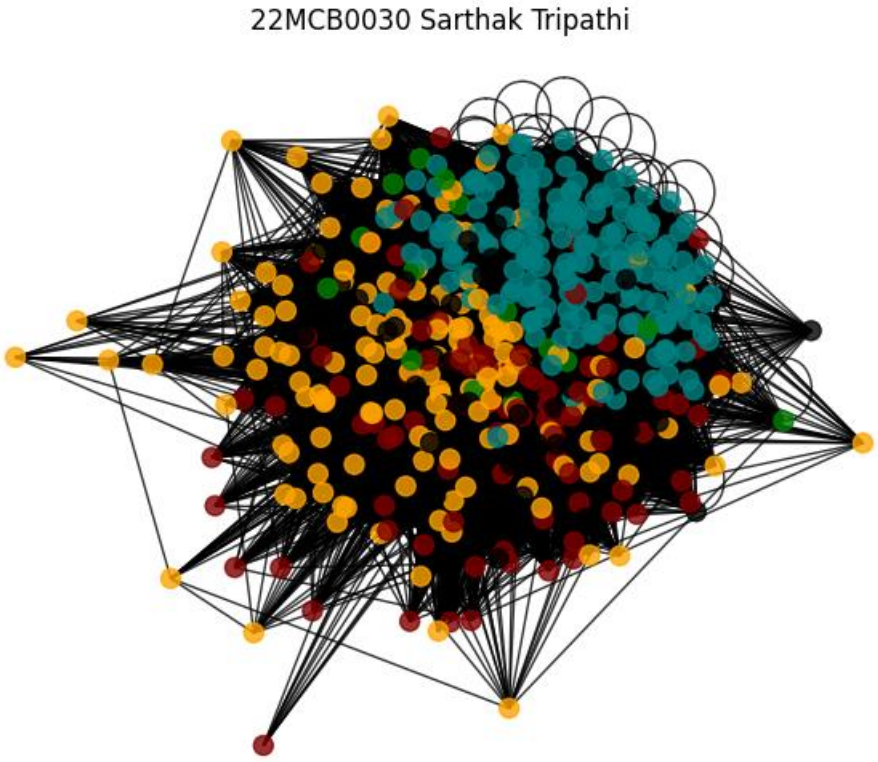
This should return the associated communities detected from G in the form of a dictionary. The keys of the dictionary are the nodes and the values correspond to the community in which that node belongs to.

Visualize Communities

Be aware that this visualization is only possible because I selected a small number of nodes (which in turn generated a small number of edges). Rendering the image associated with a large

network is quite a heavy task for any computer. Be aware prior to running the next component of code on your network.

```
[16] unique_coms = np.unique(list(comms.values()))
     cmap = {
         0 : 'maroon',
         1 : 'teal',
         2 : 'black',
         3 : 'orange',
         4 : 'green',
         5 : 'yellow'
     }

     node_cmap = [cmap[v] for _,v in comms.items()]

     pos = nx.spring_layout(G)
     nx.draw(G, pos, node_size = 75, alpha = 0.8, node_color=node_cmap)
     plt.title("22MCB0030 Sarthak Tripathi")
     plt.show()
```



Implementing Girvan-Newman algorithm

We will use Zachary's karate club Graph to demonstrate how you can perform community detection using the Girvan-Newman Algorithm.

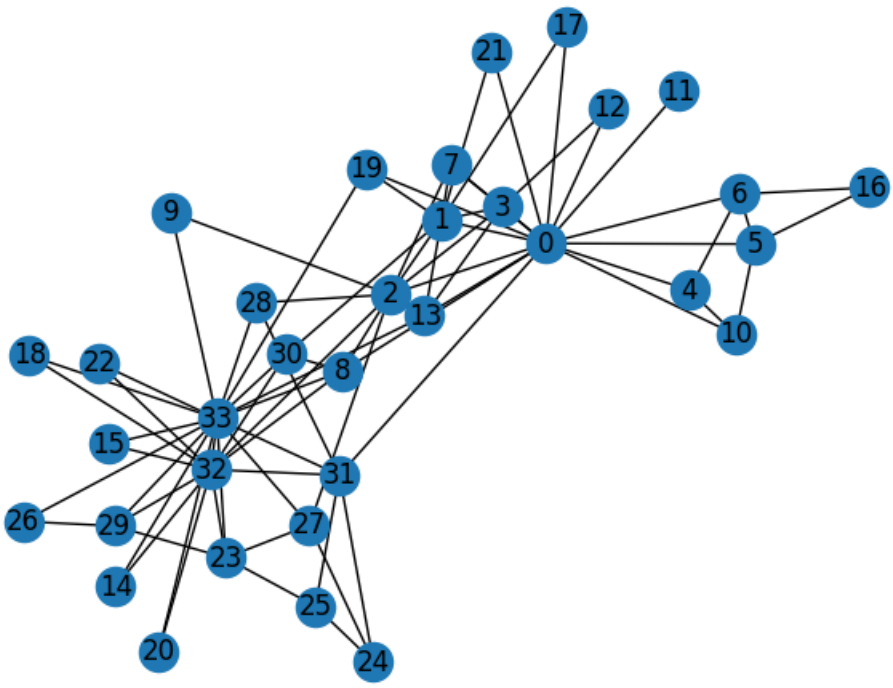
Zachary's karate club is a social network of a university karate club. The network became a popular example of community structure in networks after its use by Michelle Girvan and Mark Newman in 2002.

We will start off with loading the required libraries: networkx and matplotlib:

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

%matplotlib inline
```

The Karate Club graph comes pre-installed with the networkx library. So, we can easily import it:

```
[18] # load the graph
     G = nx.karate_club_graph()

     # visualize the graph
     nx.draw(G, with_labels = True)
     plt.title("22MCB0030 Sarthak tripathi")
     plt.show()
```



```
[19] len(G.nodes), len(G.edges)

(34, 78)
```

It turns out that the graph has 34 nodes and 78 edges. This is a pretty small-sized graph. Usually, real-world graphs could easily be a thousand times bigger than this graph. But we'll go ahead with this graph for the purpose of this tutorial.

Now, we know that in the Girvan-Newman method, the edges of the graph are eliminated one-by-one based on the EBC score. So, the first task is to find the EBC values for all the edges and then take off the edge with the largest value. The below function performs the exact task:

```
def edge_to_remove(graph):
  G_dict = nx.edge_betweenness_centrality(graph)
  edge = ()

  # extract the edge with highest edge betweenness centrality score
  for key, value in sorted(G_dict.items(), key=lambda item: item[1], reverse = True):
      edge = key
      break

  return edge
```

Now the function below will use the above function to partition the graph into multiple communities:

```
[10] def girvan_newman(graph):
       # find number of connected components
       sg = nx.connected_components(graph)
       sg_count = nx.number_connected_components(graph)

       while(sg_count == 1):
         graph.remove_edge(edge_to_remove(graph)[0], edge_to_remove(graph)[1])
         sg = nx.connected_components(graph)
         sg_count = nx.number_connected_components(graph)

       return sg
```

Next, let's pass the Karate Club graph to the above function:

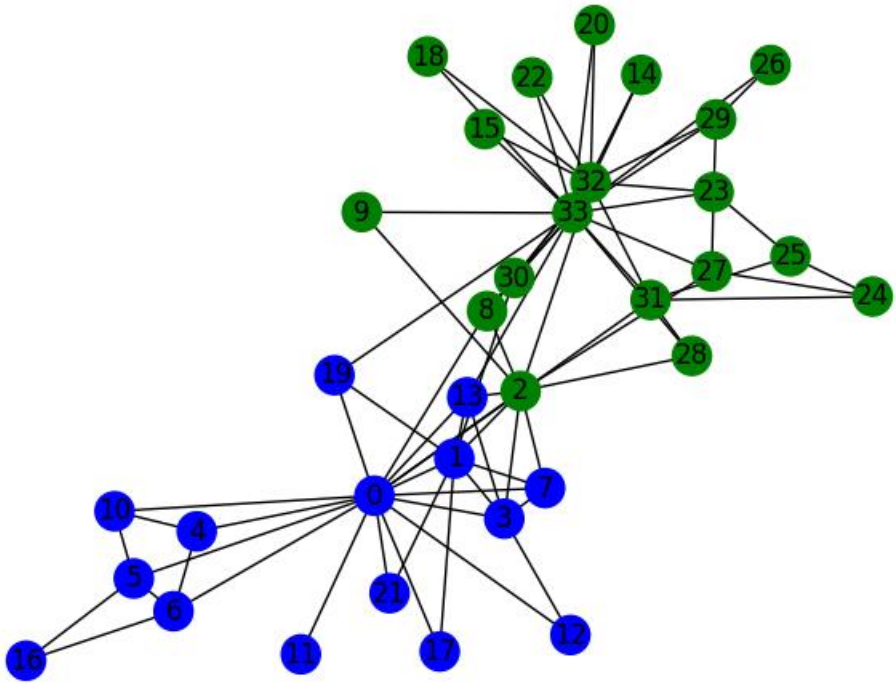
```
[11] # find communities in the graph
     c = girvan_newman(G.copy())

     # find the nodes forming the communities
     node_groups = []

     for i in c:
       node_groups.append(list(i))
```

```
[12] print(node_groups)

[[0, 1, 3, 4, 5, 6, 7, 10, 11, 12, 13, 16, 17, 19, 21], [2, 8, 9, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33]]
```

Output: [[0, 1, 3, 4, 5, 6, 7, 10, 11, 12, 13, 16, 17, 19, 21], [32, 33, 2, 8, 9, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]]

These are the node sets belonging to two communities in the Karate Club graph according to the Girvan-Newman algorithm. Let's use these node sets to visualize the communities discovered in the graph:

```
[20] # plot the communities
     color_map = []
     for node in G:
         if node in node_groups[0]:
             color_map.append('blue')
         else:
             color_map.append('green')

     nx.draw(G, node_color=color_map, with_labels=True)
     plt.title("22MCB0030 Sarthak Tripathi")
     plt.show()
```



Dataset Used

The Marvel Universe Social Network

Marvel Comics, originally called Timely Comics Inc., has been publishing comic books for several decades. "The Golden Age of Comics" name that was given due to the popularity of the books during the first years, was later followed by a period of decline of interest in superhero stories due to World War ref. In 1961, Marvel relaunched its superhero comic books publishing line. This new era started what has been known as the Marvel Age of Comics. Characters created during this period such as Spider-Man, the Hulk, the Fantastic Four, and the X-Men, together with those created during the Golden Age such as Captain America, are known worldwide and have become cultural icons during the last decades. Later, Marvel's characters popularity has been revitalized even more due to the release of several recent movies which recreate the comic books using spectacular modern special effects. Nowadays, it is possible to access the content of the comic books via a digital platform created by Marvel, where it is possible to subscribe monthly or yearly to get access to the comics. More information about the Marvel Universe can be found here.

Content

The dataset contains heroes and comics, and the relationship between them. The dataset is divided into three files:

- nodes.csv: Contains two columns (node, type), indicating the name and the type (comic, hero) of the nodes.
- edges.csv: Contains two columns (hero, comic), indicating in which comics the heroes appear.hero-edge.csv: Contains the network of heroes which appear together in the comics. This file was originally taken from http://syntagmatic.github.io/exposedata/marvel/

Political Blogs

The dataset utilized in the paper titled "The political blogosphere and the 2004 US Election" by L. A. Adamic and N. Glance (2005). It's a directed network of hyperlinks between weblogs on US politics.

Word Co-occurrence Network

In this example, we look at the word co-occurrence dataset used in the paper titled "News Framing of Population and Family Planning Issues via Syntactic Network Analysis" by E.F. Legara et al. Using a community detection algorithm, we looked for word patterns (how the words were 'arranged' or connected in the sentences of each news article) to evaluate how the media framed the population issue in the Philippines through the use of the different labels to refer to it in public discourse.

CODING

First Step toward our implementation of Community Detection Algorithm for Social network is to import libraries:

22MCB0030 Sarthak Tripathi

Implement Community detection Algorithim on Social network

```
#22mcb0030 Sarthak Tripathi
#Social Network Analysis
#Assessment - 2

try:
    ## For Network Analysis and Visualization
    import networkx as nx
    import numpy as np
    import matplotlib.pyplot as plt
    from collections import defaultdict
    import operator
    import warnings
    import pandas as pd

    ## For Hierarchical Clustering
    from scipy.cluster import hierarchy
    from scipy.spatial import distance

    ## For Community Detection (Louvain Method)
    import community

except:
    import traceback
    traceback.print_exc()
    raise ImportError('Something failed, see above.')

warnings.filterwarnings("ignore")
%matplotlib inline
```

Second step is to load the network:

Here I am using the karate club network.

The karate_club_graph is a well-known social network that represents the friendships between members of a karate club at a university in the 1970s. The graph has 34 nodes, representing the members of the karate club, and 78 edges, representing the friendships between the members.

```
[2]  K = nx.karate_club_graph()
```

Now to get more information about this network we can find out the Network Structure and Properties:

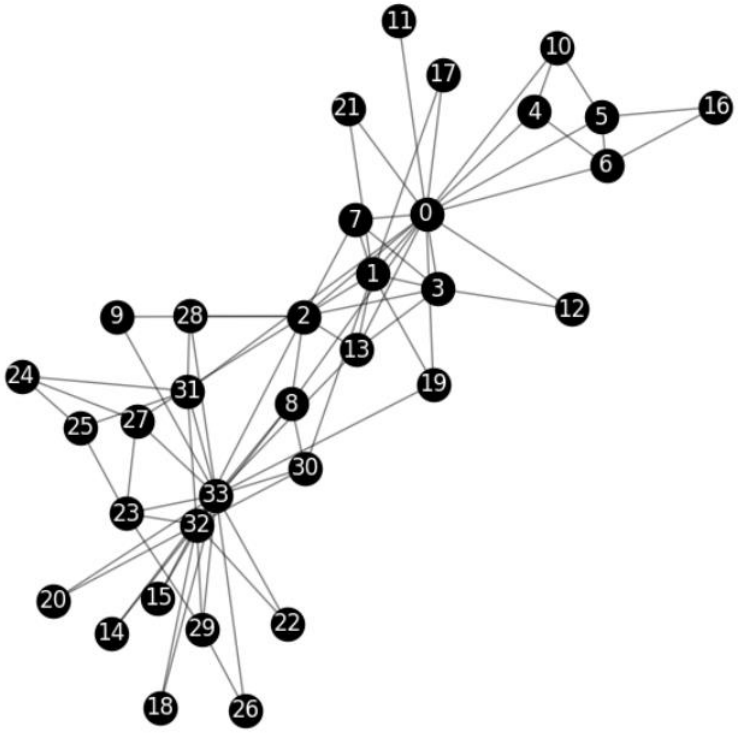
```
print ("Number of nodes: ", K.size())
print ("Number of edges: ", K.order())
```

```
Number of nodes:  78
Number of edges:  34
```

```
[4]  pos = nx.fruchterman_reingold_layout(K);
```

```
[6]  plt.figure(figsize=(8,8));
     plt.axis("off");
     nx.draw_networkx_nodes(K, pos, node_size=300, node_color="black");
     nx.draw_networkx_edges(K, pos, alpha=0.500);
     nx.draw_networkx_labels(K, pos, font_color="white");
     plt.title("22MCB0030 Sarthak Tripathi")
     plt.show();
```

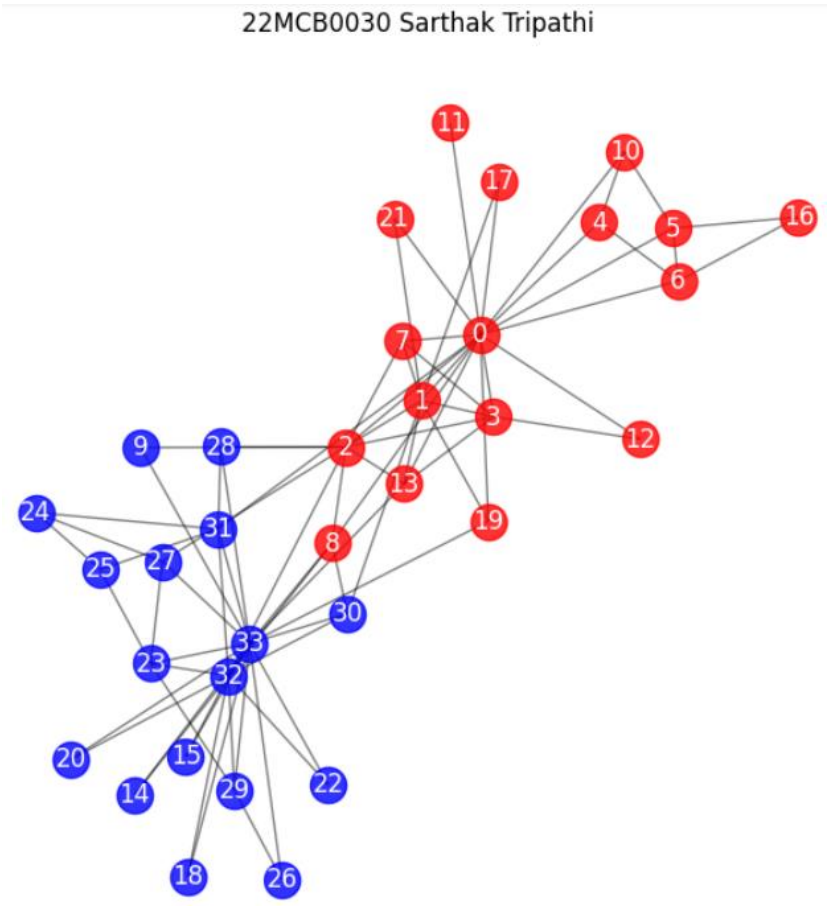
22MCB0030 Sarthak Tripathi



Now we are performing real-world partition to understand more about our network:

```
[8]  sarthak = [0,1,2,3,4,5,6,7,8,10,11,12,13,16,17,19,21]
     tripathi = [9,14,15,18,20,22,23,24,25,26,27,28,29,30,31,32,33]

     for mem in K.nodes():
         if mem in sarthak:
             K.nodes()[mem]["group"] = "sarthak"
         else:
             K.nodes()[mem]["group"] = "tripathi"
```

```
plt.figure(figsize=(8,8));
plt.axis("off");
nx.draw_networkx_nodes(K,pos,
                       nodelist=sarthak,
                       node_color='r',
                       node_size=300,
                   alpha=0.8);
nx.draw_networkx_nodes(K,pos,
                       nodelist=tripathi,
                       node_color='b',
                       node_size=300,
                   alpha=0.8);
nx.draw_networkx_edges(K, pos, alpha=0.5);
nx.draw_networkx_labels(K, pos, font_color="white");
plt.title("22MCB0030 Sarthak Tripathi")
plt.show()
```



Finding the Cliques

```
Cliques

[11] print (list(nx.find_cliques(K)))

[[0, 1, 17], [0, 1, 2, 3, 13], [0, 1, 2, 3, 7], [0, 1, 19], [0, 1, 21], [0, 4, 10], [0, 4, 6], [0, 5, 10], [0, 5, 6], [0, 8, 2], [0, 11], [0, 12, 3], [0, 31], [1, 30],
```

Divisive Method: The Girvan-Newman Algorithm

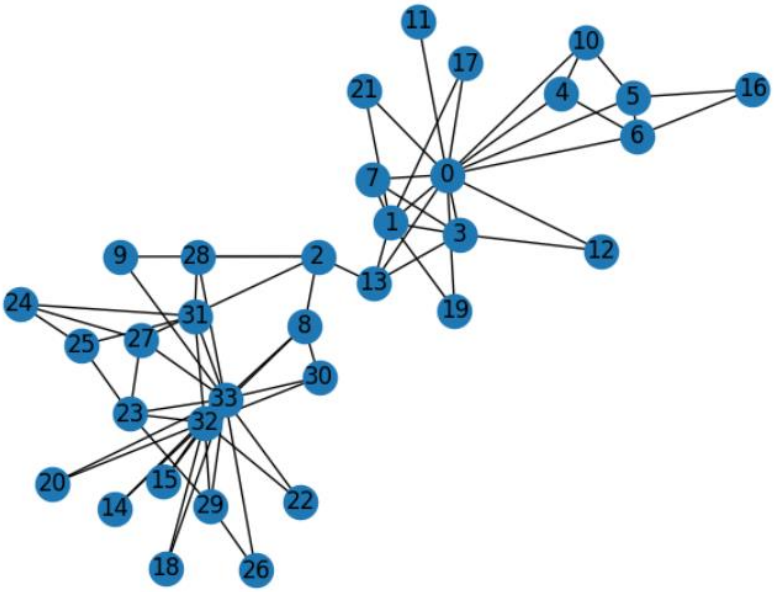
```
[12] #order the dictionary by value, which is the edge betweeness of two nodes
     ebet = nx.edge_betweenness_centrality(K)
     sorted_ebet = sorted(ebet.items(), key=operator.itemgetter(1), reverse=True)
     sorted_ebet[0:5]

     [((0, 31), 0.1272599949070537),
      ((0, 6), 0.07813428401663695),
      ((0, 5), 0.07813428401663694),
      ((0, 2), 0.0777876807288572),
      ((0, 8), 0.07423959482783014)]
```

Iteratively remove edges with the highest edge betweenness. For purpose of illustration, we first make a copy of the karate club network K . We then perform i=10 iterations of edge removal.

```
K2 = K.copy()

def remove_top_ebet(K2):
    ebet = nx.edge_betweenness_centrality(K2)
    sorted_ebet = sorted(ebet.items(), key=operator.itemgetter(1), reverse=True)
    edge_to_remove = sorted_ebet[0]
    K2.remove_edge(*edge_to_remove[0])

    return K2

for i in range(10):
    K2=remove_top_ebet(K2)
```

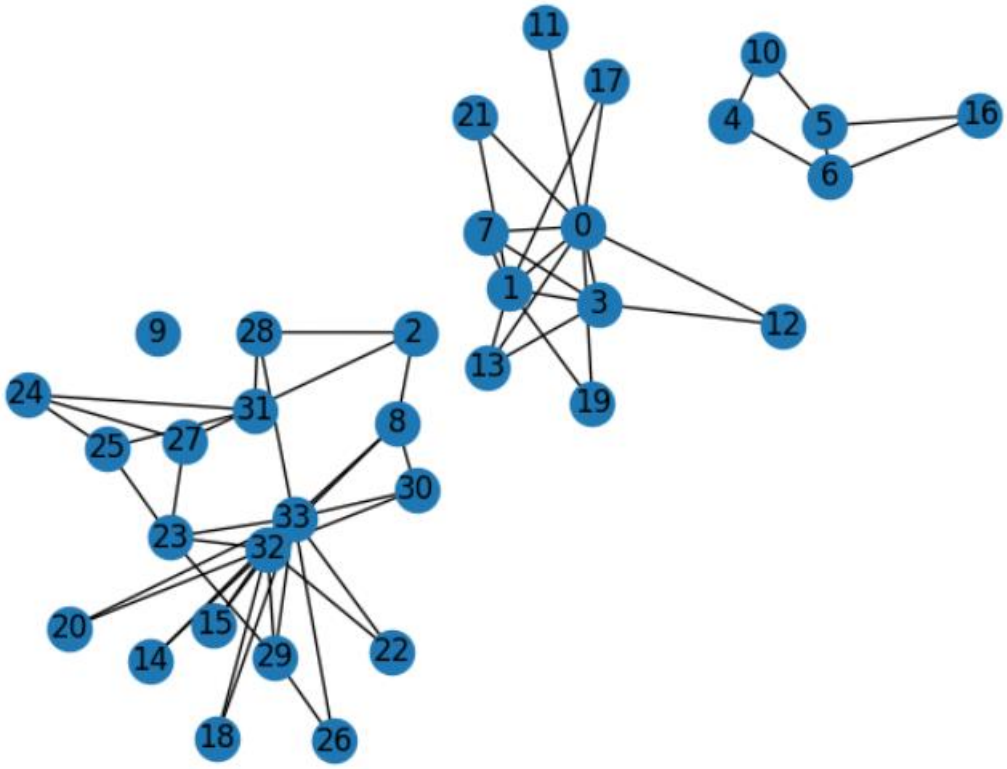
```
[14] nx.draw(K2, pos)
     nx.draw_networkx_labels(K2, pos)
     plt.title("22MCB0030 Sarthak Tripathi")
     plt.show()
```

22MCB0030 Sarthak Tripathi




```
for i in range(5):
    K2=remove_top_ebet(K2)

nx.draw(K2, pos)
nx.draw_networkx_labels(K2, pos)
plt.title("22MCB0030 Sarthak Tripathi")
plt.show()
```

22MCB0030 Sarthak Tripathi



Finding out the connected components

Connected components are subgraphs where all the nodes are connected to each via a path. Unlike cliques, they don't have to be _directly_ connected. Moreover, the Giant Component refers to the network's largest component.

```
[17] print ("Number of components: ", len(list(nx.connected_components(K2))),"\n")
     print (list(nx.connected_components(K2)))

     Number of components:  4

     [{0, 1, 3, 7, 11, 12, 13, 17, 19, 21}, {32, 33, 2, 8, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31}, {4, 5, 6, 10, 16}, {9}]
```

Agglomerative Method: Hierarchical Clustering

In this method, we start with a distance matrix, which is a matrix that provides the distance between two nodes i and j in network G. Using D, we then perform [hierarchical clustering in a recursive manner, group _similar_ nodes at each step. There are various ways to perform the similarity grouping using _agglomerative procedures_ such as single, complete, and average, to name a few. In the example below, we implement the [average/UPGMA linkage], which is the one implemented in the Ravasz Algorithm published in [Science in 2002].

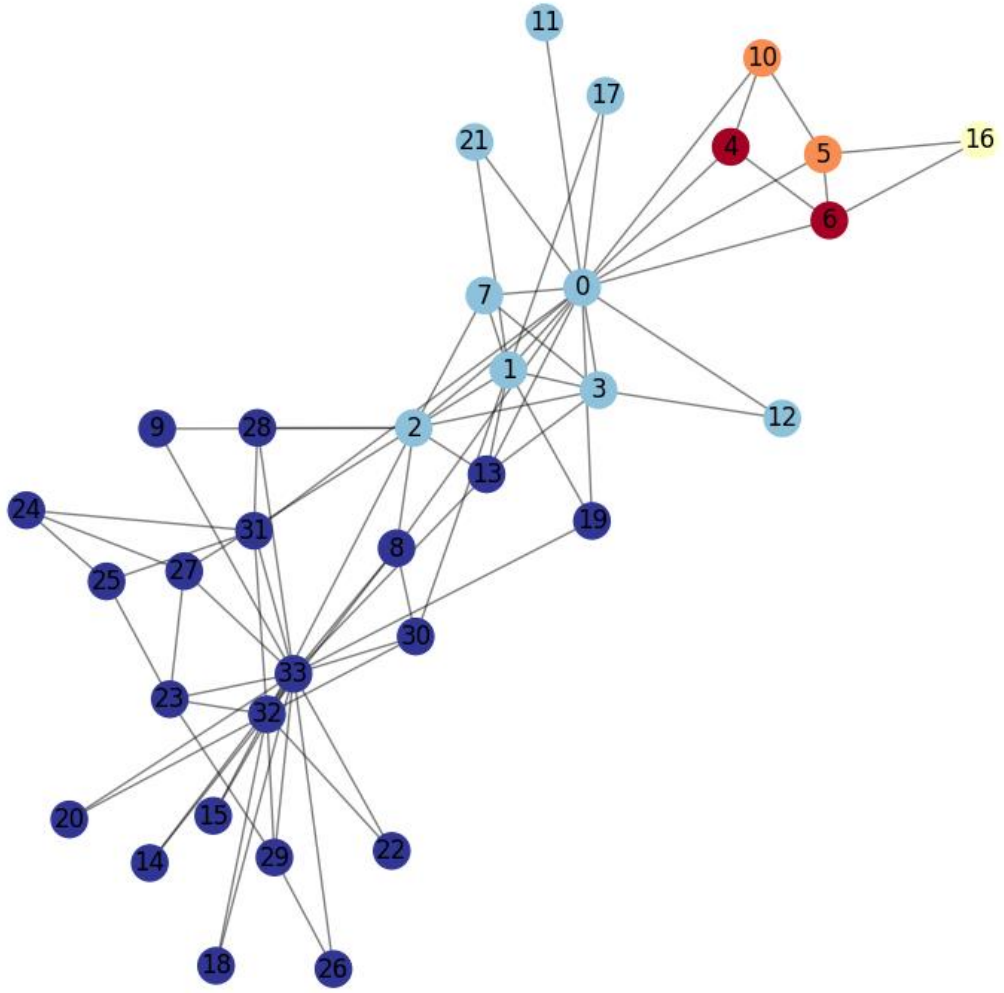
```
[21] def create_hc(G, t):
         ## Set-up the distance matrix D
         labels = G.nodes()    # keep node labels
         path_length = nx.all_pairs_shortest_path_length(G)
         distances=np.zeros((len(G),len(G)))

         for node, info in path_length:
             for other_node, l in info.items():
                 distances[node][other_node] = l
                 distances[other_node][node] = l
                 if node==other_node: distances[node][node]=0


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

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

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

         return Z, membership, partition
```

```
[22] Z, membership, partition = create_hc(K, t=1.15)
     partition.items()

     dict_items([(4, [0, 1, 2, 3, 7, 11, 12, 17, 21]), (1, [4, 6]), (2, [5, 10]), (5, [8, 9, 13, 14, 15, 18, 19, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33]), (3,
     [16])])
```

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

```
plt.figure(figsize=(10,10))
plt.axis('off')

#nx.draw_networkx_nodes(K, pos, cmap=plt.cm.RdYlBu, node_color=partition.values())
nx.draw_networkx_nodes(K, pos, cmap=plt.cm.RdYlBu, node_color=list(partition.values()))
nx.draw_networkx_edges(K, pos, alpha=0.5)
nx.draw_networkx_labels(K, pos)
plt.title("22MCB0030 Sarthak Tripathi")
plt.show()
```

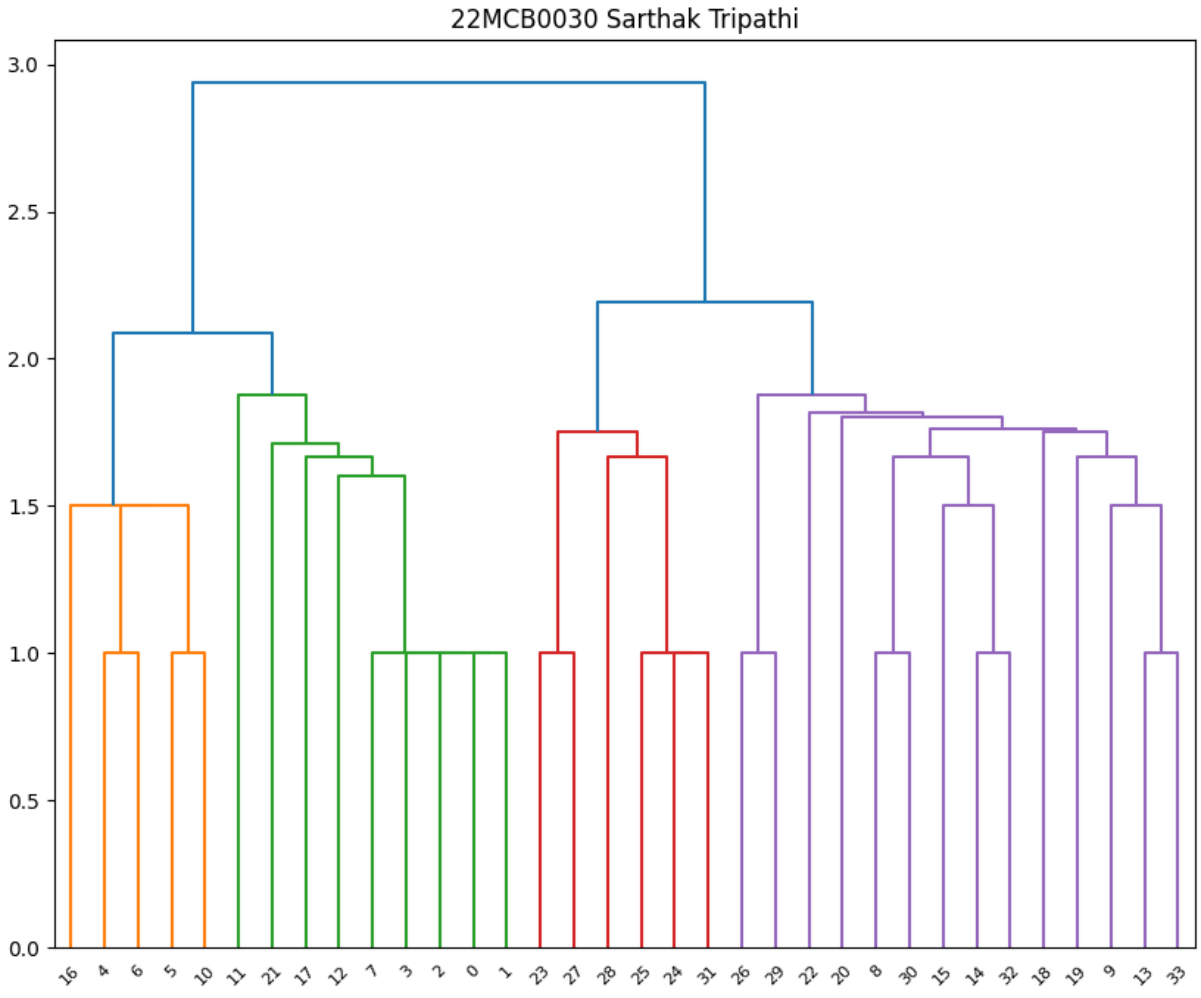
22MCB0030 Sarthak Tripathi



Dendrogram

The dendrogram allows us to visualize the order in which nodes are grouped together in the clustering process.

```
[25] plt.figure(figsize=(10,8))
     hierarchy.dendrogram(Z)
     plt.title("22MCB0030 Sarthak Tripathi")
     plt.show()
```

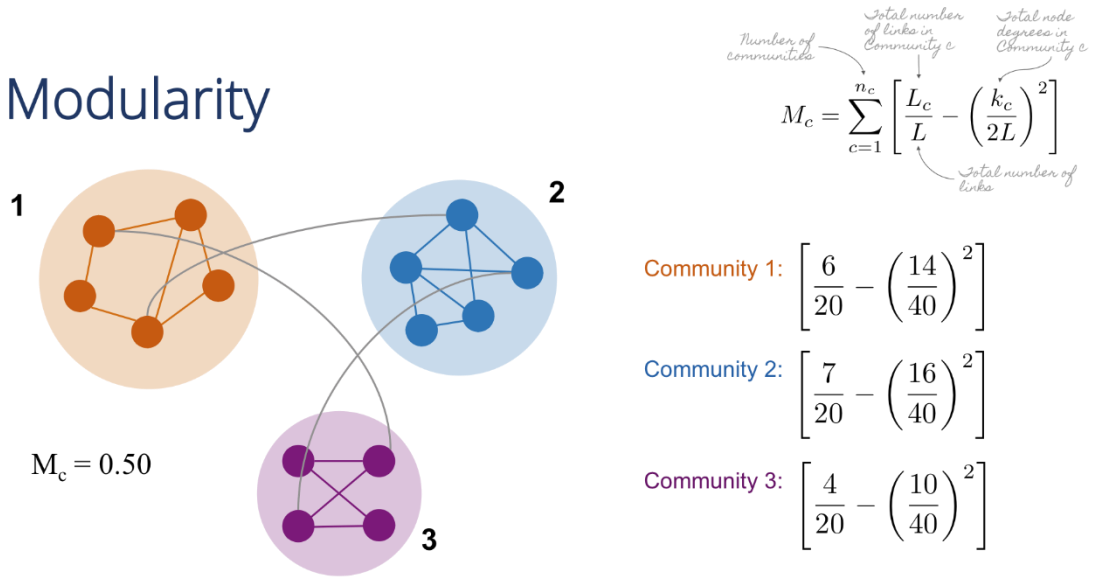


Modularity-Based Community Detection: The Louvain Method

The community detection algorithm that we implement in this notebook is a modularity-based algorithm. The modularity Mc quantifies how good a "community" or partition is, and is given by

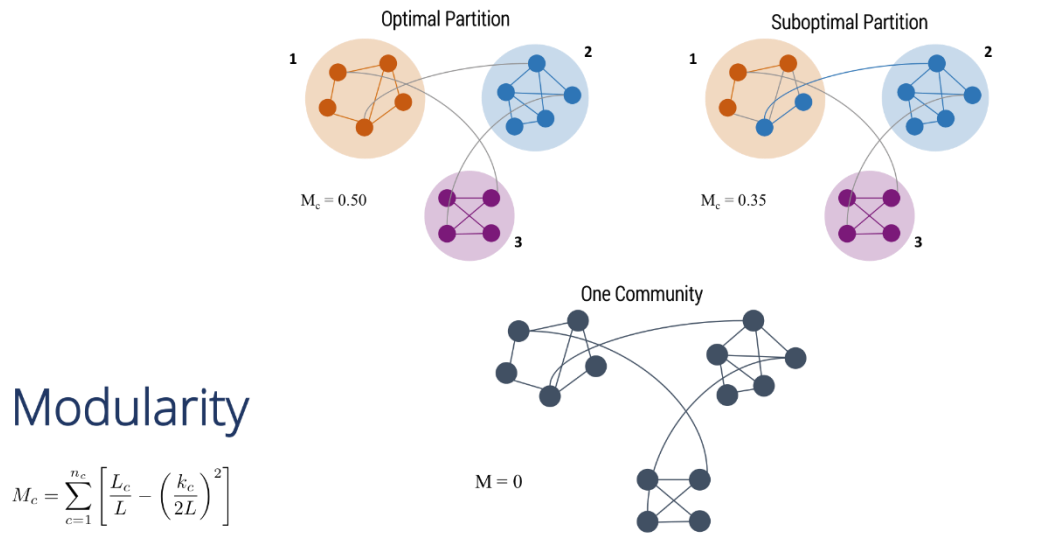
Mc=∑c=1nc[LcL−(kc2L)2]

where Nc is the number of communities, Lc is the number of links within the community, kc is the total degree of nodes belonging to the community, and L is the total number of links in the network. The illustration below that is taken from the slide deck shows how modularity is calculated.



Community Structures | EF Legara | 2017 NTU Winter School on Complexity Science

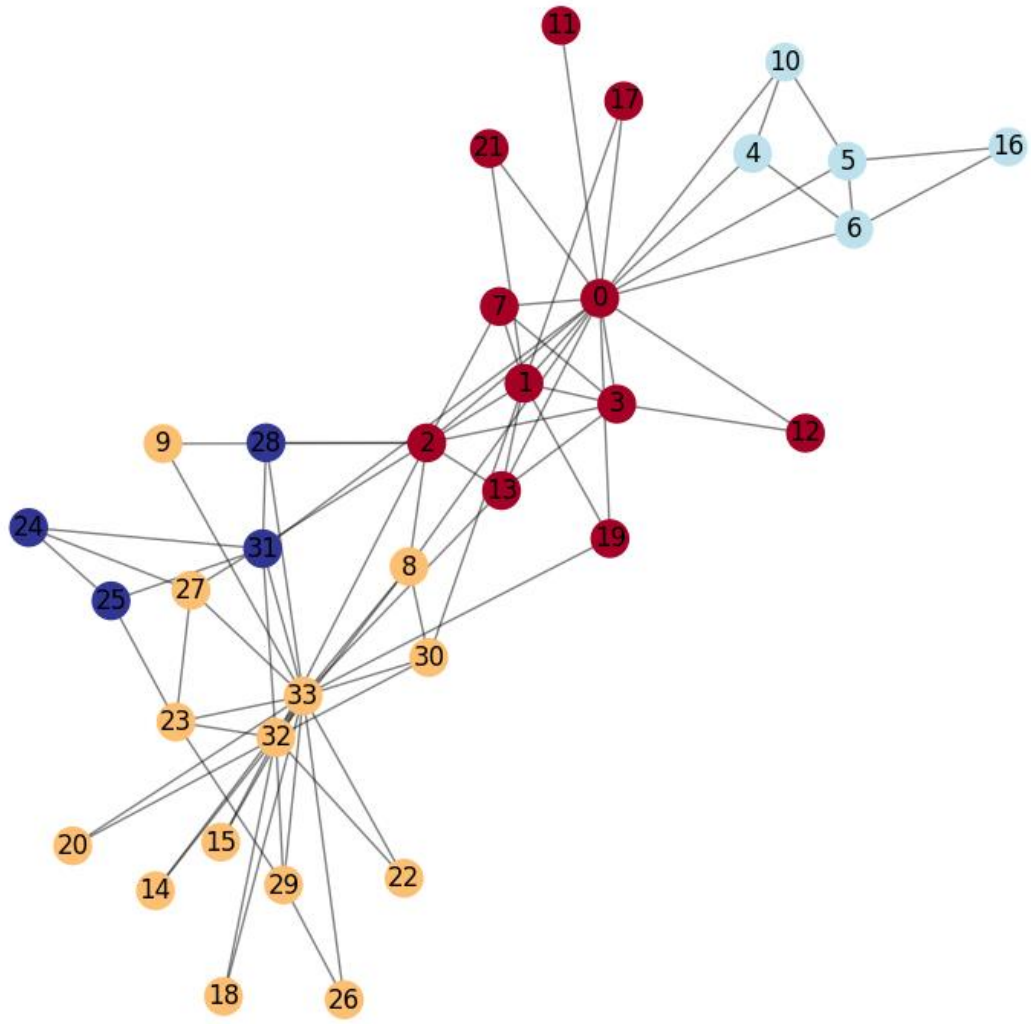
The higher the network modularity is, the more optimal the partitioning. And, if you only have a single community, M=0 .



Community Structures | EF Legara | 2017 NTU Winter School on Complexity Science

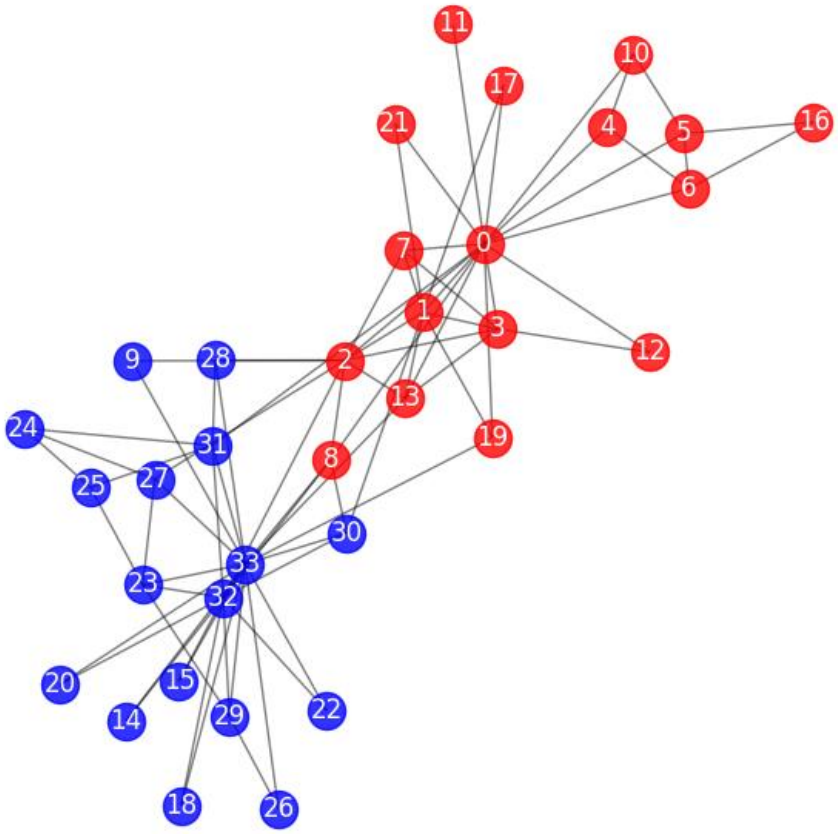
```
[33] import community.community_louvain as cl
     partition = cl.best_partition(K)
```

```
plt.figure(figsize=(10,10))
plt.axis('off')

nx.draw_networkx_nodes(K, pos, cmap=plt.cm.RdYlBu, node_color=list(partition.values()))
nx.draw_networkx_edges(K, pos, alpha=0.5)
nx.draw_networkx_labels(K, pos)
plt.title("22MCB0030 Sarthak tripathi")
plt.show()
```



The algorithm produces four (4) partitions. Let's compare this partition to the actual.

```
[37] plt.figure(figsize=(8,8));
     plt.axis("off");
     nx.draw_networkx_nodes(K,pos, nodelist=sarthak, node_color='r',
                            node_size=300, alpha=0.8);
     nx.draw_networkx_nodes(K,pos, nodelist=tripathi, node_color='b',
                            node_size=300, alpha=0.8);
     nx.draw_networkx_edges(K, pos, alpha=0.5);
     nx.draw_networkx_labels(K, pos, font_color="white");
```



Community Detection Algorithm using datasets

Step- 1 Load the Dataset

```
[43] !wget "https://storage.googleapis.com/kaggle-data-sets/741/1378/compressed/hero-network.csv.zip?X-Goog-Algorithm=GOOG4-RSA-SHA256&X-Goog-Credential=gcp-kaggle-com%40kaggle-161607.iam.gserviceac

     The name is too long, 776 chars total.
     Trying to shorten...
     New name is hero-network.csv.zip?X-Goog-Algorithm=GOOG4-RSA-SHA256&X-Goog-Credential=gcp-kaggle-com@kaggle-161607.iam.gserviceaccount.com%2F20230528%2Fauto%2Fstorage%2Fgoog4_request&X-Goog-Date=
     --2023-05-28 15:48:25--  https://storage.googleapis.com/kaggle-data-sets/741/1378/compressed/hero-network.csv.zip?X-Goog-Algorithm=GOOG4-RSA-SHA256&X-Goog-Credential=gcp-kaggle-com%40kaggle-1616
     Resolving storage.googleapis.com (storage.googleapis.com)... 142.250.136.128, 142.250.148.128, 142.251.172.128, ...
     Connecting to storage.googleapis.com (storage.googleapis.com)|142.250.136.128|:443... connected.
     HTTP request sent, awaiting response... 200 OK
     Length: 2428374 (2.3M) [application/zip]
     Saving to: 'hero-network.csv.zip?X-Goog-Algorithm=GOOG4-RSA-SHA256&X-Goog-Credential=gcp-kaggle-com@kaggle-161607.iam.gserviceaccount.com%2F20230528%2Fauto%2Fstorage%2Fgoog4_request&X-Goog-Date=

     hero-network.csv.zi 100%[===================>]   2.32M  --.-KB/s    in 0.01s

     2023-05-28 15:48:25 (180 MB/s) - 'hero-network.csv.zip?X-Goog-Algorithm=GOOG4-RSA-SHA256&X-Goog-Credential=gcp-kaggle-com@kaggle-161607.iam.gserviceaccount.com%2F20230528%2Fauto%2Fstorage%2Fgoog

[44] !unzip "/content/hero-network.csv.zip" -d '/content/sample_data'

     Archive:  /content/hero-network.csv.zip
       inflating: /content/sample_data/hero-network.csv
```

```
[49] marvel = pd.read_csv('/content/sample_data/hero-network.csv', header=None)

[50] marvel.head()
```

	0	1
0	hero1	hero2
1	LITTLE, ABNER	PRINCESS ZANDA
2	LITTLE, ABNER	BLACK PANTHER/T'CHAL
3	BLACK PANTHER/T'CHAL	PRINCESS ZANDA
4	LITTLE, ABNER	PRINCESS ZANDA

```
[51] M = nx.from_pandas_edgelist(marvel, 0, 1)
```

Graph Statistics

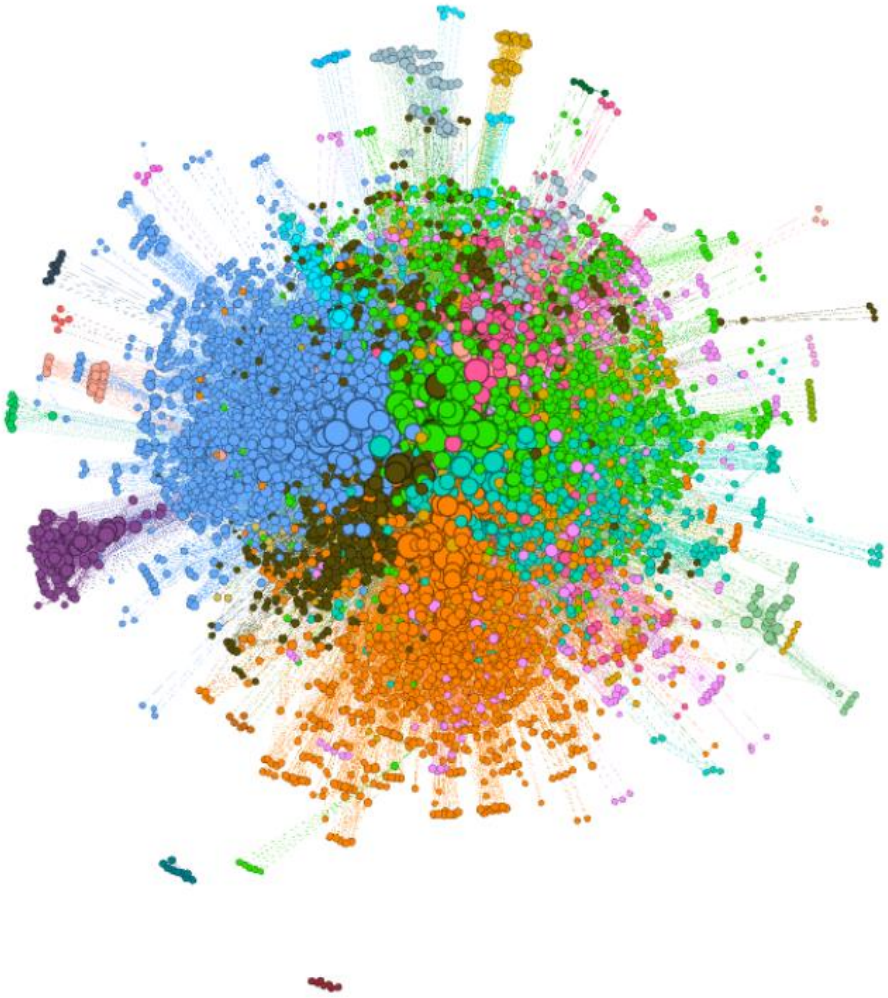
```
[52] M.size() #number of edges

     167220

[53] M.order() #number of nodes

     6428

[54] M_cliques = list(nx.find_cliques(M))

[55] print (M_cliques[0:2])

     [['YOUNG, MR.', 'HAVOK/ALEX SUMMERS ', 'MULTIPLE MAN/JAMES A', 'POLARIS/LORNA DANE', 'FIXX', 'YOUNG, AGNES', 'GREYSTONE/BRIAN YOUN', 'BLACK, RACHEL', 'YOUNG, LOUISE', 'ARCHER/JUDE BLACK', 'SHARD
```

Finding Communities

```
[57] import community.community_louvain as cl
     partition = cl.best_partition(M)
```

```
[58] len(set(partition.values()))

     30
```

```
[84] plt.figure(figsize=(10,10))
     plt.axis('off')

     pos = nx.spring_layout(M)
     nx.draw_networkx_nodes(M, pos, cmap=plt.cm.RdYlBu, node_color=list(partition.values()))
     nx.draw_networkx_edges(M, pos, alpha=0.4)
     plt.savefig("Marvel_Universe.png")
```



Above image show the communities in the dataset.

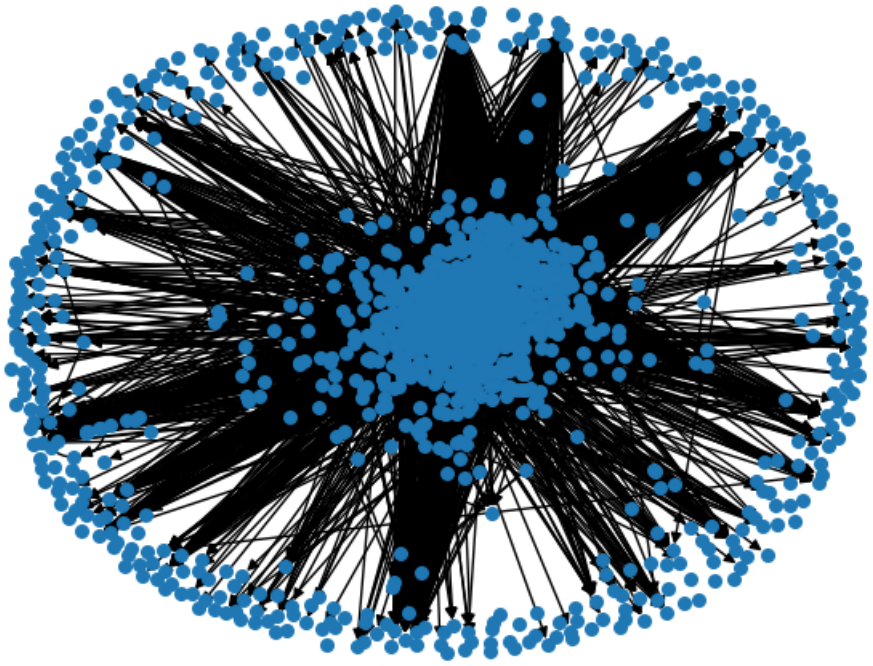
Political Blog Dataset:

Here, we use the dataset utilized in the paper titled "The political blogosphere and the 2004 US Election" by L. A. Adamic and N. Glance (2005). It's a directed network of hyperlinks between weblogs on US politics.

Political Blogs

```
[60] P = nx.read_gml("/content/polblogs copy.gml")

[61] P.order(), P.size()

     (1490, 19090)

[62] pos = nx.spring_layout(P)
     nx.draw(P, pos, node_size=30)
     plt.show()
```

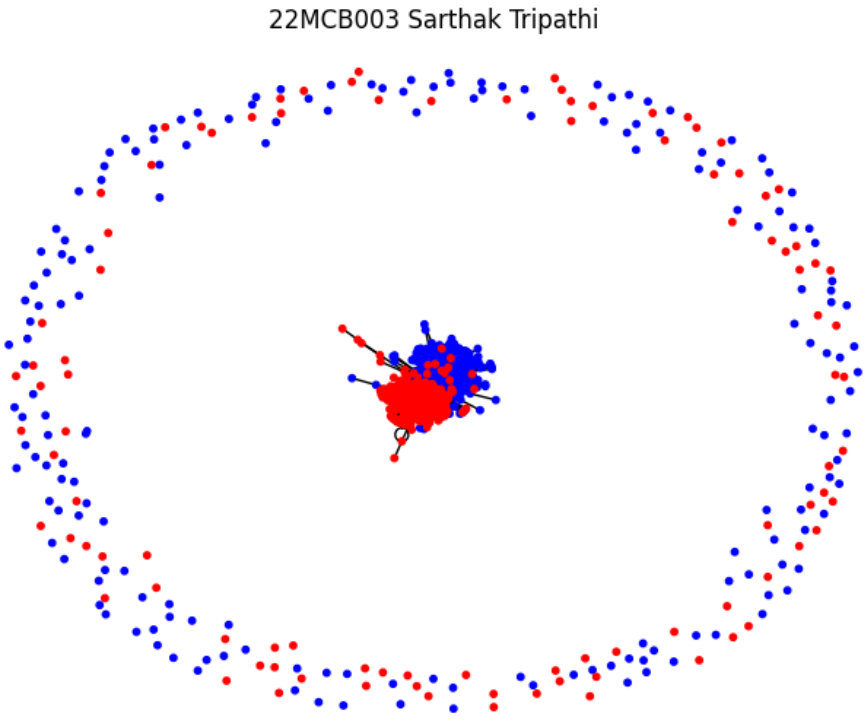
22MCB0030 Sarthak Tripathi



Now the next step is to found out the communities between the edges:

Finding Communities

To perform the community detection algorithm, the directed graph needs to be made into an undirected one.



Extract Giant Component

This function distinguishes the largest component of an "igraph" or a "network" object and illustrates them as a list which contains the edgelist of the giant component. If the input graph was bipartite and the "bipartite.proj" was TRUE, it will projet it and you can decide to which project you want to continue to work with that.

```
GC_nodes = max(nx.connected_components(PR), key=len)
GC = PR.subgraph(GC_nodes).copy()
```

```
[69] type(GC)
```

```
networkx.classes.graph.Graph
```

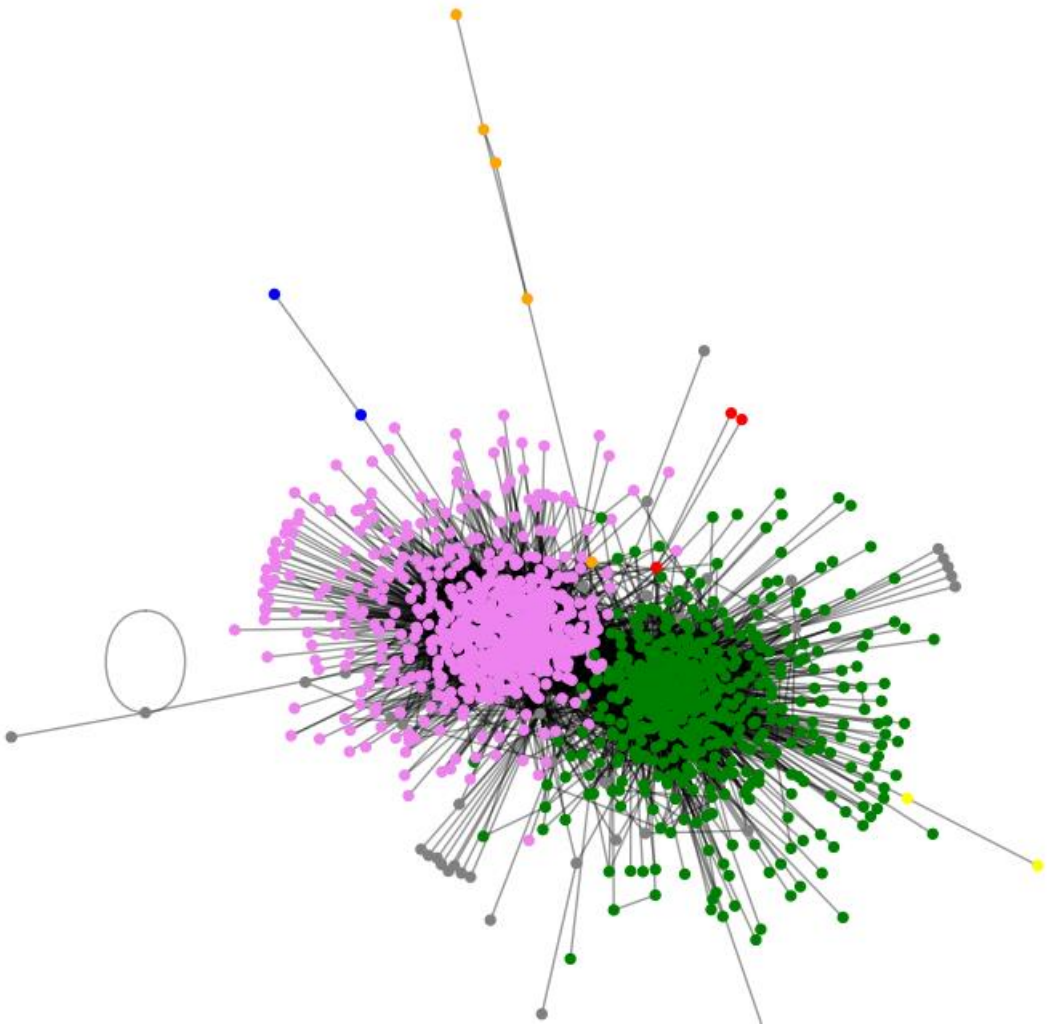
```
[71] import community.community_louvain as cl
partition = cl.best_partition(GC)
```

```
[72] print ("22MCB0030 SARTHAK TRIPATHI")
print ("Number of Communities: ", len(set(partition.values())))
```

```
22MCB0030 SARTHAK TRIPATHI
Number of Communities:  10
```

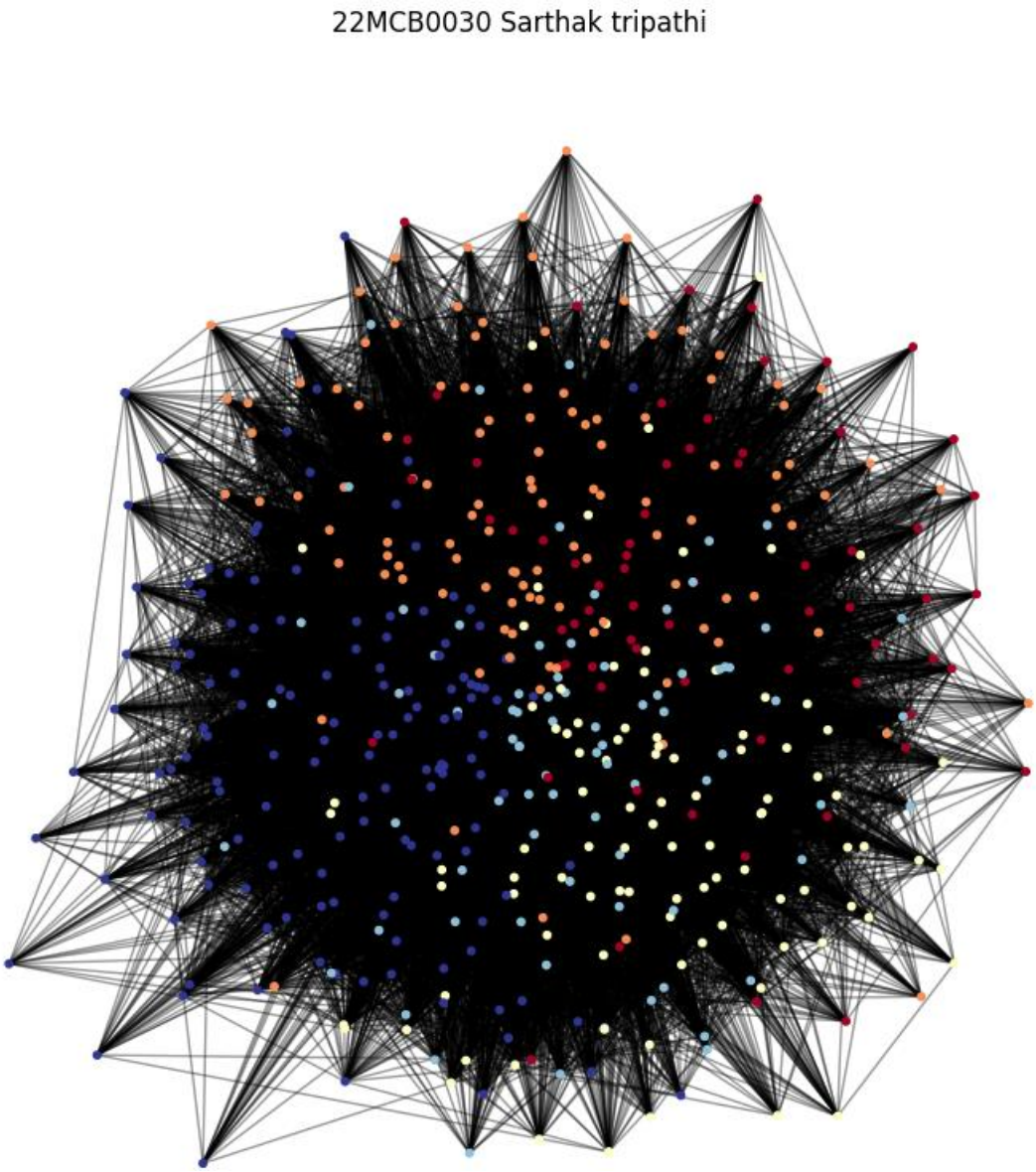
```
[74] palette = ['blue', 'green', 'red', 'yellow', 'orange', 'violet'] + 5 * ['grey']

for k, v in partition.items():
    GC.nodes()[k]["louvain-val"] = v
colors = [palette[GC.nodes()[node]["louvain-val"]] for node in GC.nodes()]
```

```
plt.figure(figsize=(10,10))
plt.axis('off')
pos = nx.spring_layout(GC, scale=3)
nx.draw_networkx_nodes(GC, pos, node_color=colors, node_size=18, label=True)
nx.draw_networkx_edges(GC, pos, alpha=0.4)
plt.title("22MCB0030 SARTHAK TRIPATHI")
plt.savefig("22mcb0030_polblogs.png")
```



Word Co-occurrence Network

```
[79] S = nx.read_gml("/content/word-net.gml", label="label")

[80] import community.community_louvain as cl
     partition = cl.best_partition(S,weight="weight")

     plt.figure(figsize=(10,10))
     plt.axis('off')

     pos = nx.spring_layout(F, iterations=150)
     nx.draw_networkx_nodes(F, pos,cmap=plt.cm.RdYlBu, node_color=list(partition.values()), node_size=10, label=True)
     nx.draw_networkx_edges(F, pos, alpha=0.4)
     plt.title("22MCB0030 Sarthak tripathi")
     plt.savefig("22mcb0030_Frames.png")
```



Conclusion

Community detection is a powerful tool for graph analysis. From terrorist detection to healthcare initiatives, these algorithms have found their way into many real-world use cases.

The Python NetworkX package offers powerful functionalities when it comes to analysing graph networks and running complex algorithms like community detection. However, if you're looking to operationalize your graph algorithms and are looking for functionalities such as incremental updates, data persistency, and better performance, you will need to consider using a graph database in conjunction with NetworkX.