PROJECT REPORT

ON

CLUSTERING ALGORITHMS IN GRAPH THEORY AND SOCIAL NETWORK ANALYSIS

BY

NAMAN DEEP SRIVASTAVA

(2016B4A70891P)

UNDER THE GUIDANCE OF

DR. RAJIV KUMAR

PROFESSOR, DEPARTMENT OF MATHEMATICS

IN PARTIAL FULFILMENT OF THE COURSE

STUDY PROJECT (MATH F266)

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS

RAJASTHAN – 333031

(November 2018)

ACKNOWLEDGEMENT

I would like to thank the Vice Chancellor, Prof. Souvik Bhattacharyya, the Director, Prof. Ashoke Kumar Sarkar and Dr. Bhupendra Kumar Sharma, HoD, Department of Mathematics, who provided me an opportunity to enroll in this Study Project; so that I can enhance my understanding of Graph Theory.

I am obliged to my instructor Dr. Rajiv Kumar for his invaluable suggestions and help. Working on this project equipped me with indispensable skill of reading research literature in Mathematics which will go a long way. During the course of the project I faced various challenges, but with the help of my instructor I sailed.

Naman Deep Srivastava

ABSTRACT

This report is based on studies conducted by me over a period of four months of the Study Project. Firstly, I learned about the concept of graph clustering. Then I further learned about various algorithms used to implement clustering in graphs including Markov clustering algorithm, Girvan-Newman algorithm and the Attractiveness Based Community Detection algorithm. While Markov clustering is the most basic one, Girvan-Newman and ABCD deal with community detection in unweighted and weighted graphs respectively.

The second part of the project is the analysis of social networks using various graph clustering algorithms and obtaining results through their implementation. I have implemented three models namely Fat-Man evolutionary model, PageRank algorithm and the Watts-Strogatz generative model. All these models highlight different aspects and properties of the social network analysis.

Table of Contents

1. Introduction

Clustering is defined as - “process of organizing objects into groups whose members are similar in some way”. It is finding the natural grouping of items. In this project, we will first study the clustering problem. Then study about existing algorithms in literature and their complexities to tackle these clustering problems. Markov clustering is the most basic clustering algorithm and has been discussed in the following section.

The community detection is very famous and most exploited problem in the field of social network analysis. Note that we can have directed unweighted graphs, directed weighted graphs, undirected weighted graphs and undirected unweighted graphs and we are interested in studying graph clustering problem in each case. Each algorithm we have studied in this project is useful in one of above cases. GN algorithm was the stepping stone in this field of community detection. But due to its high complexity even on sparse graphs, GN algorithm was not feasible to use given limited power of everyday resources. Later on, CNM algorithm was introduced which was extension of GN algorithm but with much better complexity. Recently, for weighted networks ABCD (Attractiveness Based Community Detection) algorithm was introduced which is discussed in great details in the below sections. The most remarkable thing about ABCD algorithm is we don’t have to specify number of clusters before-hand here which is actually the case in real life problems. In addition to this, ABCD algorithm is very good in identifying communities of small order.

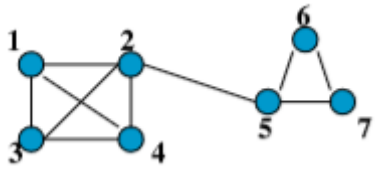
2. Markov Clustering

2.1 Introduction

Considering a graph, there will be many links within a cluster, and fewer links between clusters. This means if you were to start at a node you're more likely to stay within a cluster than travel between. This is what MCL (and several other clustering algorithms) is based on.

By doing random walks upon the graph, it may be possible to discover where the flow tends to gather, and therefore, where clusters are. Random walks on a graph are calculated using Markov Chains.

Demonstrating with the help of an example.



In one time step, a random walker at node 1 has a 33% chance of going to node 2,3 and 4 and 0% chance to nodes 5,6 and 7.

From node 2, 25% chance for 1, 3, 4, 5 and 0% for 6 and 7.

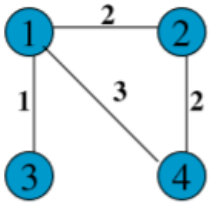
Transition matrix (or the probability matrix) would be given by:

	1	*2*	*3*	*4*	*5*	*6*	*7*
1	0	.25	.33	.33	0	0	0
2	.33	0	.33	.33	.33	0	0
3	.33	.25	0	.33	0	0	0
4	.33	.25	.33	0	0	0	0
5	0	.25	0	0	0	.5	.5
6	0	0	0	0	.33	0	.5
7	0	0	0	0	.33	.5	0

Probabilities for the next time step only depend on current probabilities (given the current probability). A random walk is an example of a Markov Chain, using the transition probability matrices.

Weighted Graph

To turn a weighted graph into a probability (transition) matrix, column normalize.



$$\begin{pmatrix} 0 & 2 & 1 & 3 \\ 2 & 0 & 0 & 2 \\ 1 & 0 & 0 & 0 \\ 3 & 2 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1/2 & 1 & 3/5 \\ 1/3 & 0 & 0 & 2/5 \\ 1/6 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 \end{pmatrix}$$

The flow is easier within dense regions than across sparse boundaries, however, in the long run this effect disappears. During the earlier powers of the Markov Chain, the edge weights will be higher in links that are within clusters, and lower between the clusters. This means there is a correspondence between the distribution of weight over the columns and the clusterings.

MCL deliberately boosts this affect by –Stopping partway in the Markov Chain –Then adjusting the transitions by columns. For each vertex, the transition values are changed so that

•Strong neighbours are further strengthened.

•Less popular neighbours are demoted.

This adjusting can be done by raising a single column to a non-negative power, and then re-normalizing. This operation is named “Inflation”. Taking the Markov chain powers is named “Expansion”.

Given such a matrix M and a real number $r > 1$, the column stochastic matrix resulting from inflating each of the columns of M with power coefficient r is written $\Gamma_r(M)$, and Γ_r is called the inflation operator with power coefficient r.

2.2 Significance of Expansion and Inflation

Expansion corresponds to computing random walks of higher length, which means random walks with many steps. It associates new probabilities with all pairs of nodes, where one node is the point of departure and the other is the destination. Since higher length paths are more common within clusters than between different clusters, the probabilities associated with node pairs lying in the same cluster will, in general, be relatively large as there are many ways of going from one to the other. Inflation will then have the effect of boosting the

probabilities of intra-cluster walks and will demote inter-cluster walks. This is achieved without any a priori knowledge of cluster structure. It is simply the result of cluster structure being present.

The inflation operator is responsible for both strengthening and weakening of current. (Strengthens strong currents, and weakens already weak currents). The inflation parameter, r, controls the extent of this strengthening/weakening. (In the end, this influences the granularity of clusters.) The expansion operator is responsible for allowing flow to connect different regions of the graph. The inflation operator is responsible for both strengthening and weakening of current.

2.3 Steps Involved

1. Input is an un-directed graph, power parameter e, and inflation parameter r.

2. Create the associated matrix.

3. Add self-loops to each node (optional).

4. Normalize the matrix.

5. Expand by taking the e^{th} power of the matrix.

6. Inflate by taking inflation of the resulting matrix with parameter r.

7. Repeat steps 5 and 6 **until a steady state is reached (convergence)**.

8. Interpret resulting matrix to discover clusters.

2.4 Convergence

It is not obvious that the result will converge. Convergence is not proven in the thesis, however it is shown experimentally that it often does occur. In practice, the algorithm converges nearly always to a "doubly idempotent" matrix:

1. It's at steady state.

2. Every value in a single column has the same number (homogeneous).

However, the final steady state may sometimes be cyclic and consist of a repeating series of matrices.

–In certain cases, the expansion and inflation act as inverses of each other. However, a slight change of parameters and the equilibrium is broken.

–Without self-loops, it's possible on bipartite graphs because of odd path lengths. Adding self-loops and slightly changing parameters fixes most of the problems.

Consider the following example.

Let the transition matrix obtained from the graph be M.

$$M$$

0.200	0.250	–	–	–	0.333	0.250	–	–	0.250	–	–
0.200	0.250	0.250	–	0.200	–	–	–	–	–	–	–
–	0.250	0.250	0.200	0.200	–	–	–	–	–	–	–
–	–	0.250	0.200	–	–	–	0.200	0.200	–	0.200	–
–	0.250	0.250	–	0.200	–	0.250	0.200	–	–	–	–
0.200	–	–	–	–	0.333	–	–	–	0.250	–	–
0.200	–	–	–	0.200	–	0.250	–	–	0.250	–	–
–	–	–	0.200	0.200	–	–	0.200	0.200	–	0.200	–
–	–	–	0.200	–	–	–	0.200	0.200	–	0.200	0.333
0.200	–	–	–	–	0.333	0.250	–	–	0.250	–	–
–	–	–	0.200	–	–	–	0.200	0.200	–	0.200	0.333
–	–	–	–	–	–	–	–	0.200	–	0.200	0.333

Expanding and Inflating in each step gives us the corresponding matrices as shown.

$$\Gamma_2 M^2,$$

0.380	0.087	0.027	–	0.077	0.295	0.201	–	–	0.320	–	–
0.047	0.347	0.210	0.017	0.150	0.019	0.066	0.012	–	0.012	–	–
0.014	0.210	0.347	0.056	0.150	–	0.016	0.046	0.009	–	0.009	–
–	0.027	0.087	0.302	0.062	–	–	0.184	0.143	–	0.143	0.083
0.058	0.210	0.210	0.056	0.406	–	0.083	0.046	0.009	0.019	0.009	–
0.142	0.017	–	–	–	0.295	0.083	–	–	0.184	–	–
0.113	0.069	0.017	–	0.062	0.097	0.333	0.012	–	0.147	–	–
–	0.017	0.069	0.175	0.049	–	0.016	0.287	0.143	–	0.143	0.083
–	–	0.017	0.175	0.012	–	–	0.184	0.288	–	0.288	0.278
0.246	0.017	–	–	0.019	0.295	0.201	–	–	0.320	–	–
–	–	0.017	0.175	0.012	–	–	0.184	0.288	–	0.288	0.278
–	–	–	0.044	–	–	–	0.046	0.120	–	0.120	0.278

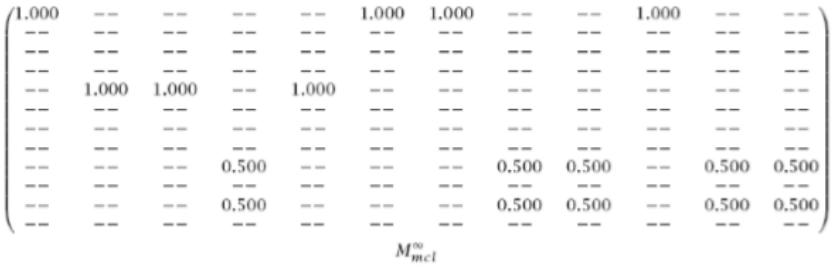
$$\Gamma_2(\Gamma_2 M^2 \cdot \Gamma_2 M^2)$$

0.448	0.080	0.023	–	0.068	0.426	0.359	–	–	0.432	–	–
0.018	0.285	0.228	0.007	0.176	0.006	0.033	0.005	–	0.007	–	–
0.005	0.223	0.290	0.022	0.173	–	0.010	0.017	0.003	0.001	0.003	0.001
–	0.018	0.059	0.222	0.040	–	0.001	0.187	0.139	–	0.139	0.099
0.027	0.312	0.314	0.028	0.439	0.005	0.054	0.022	0.003	0.010	0.003	0.001
0.116	0.007	0.001	–	0.004	0.157	0.085	–	–	0.131	–	–
0.096	0.040	0.013	–	0.037	0.083	0.197	0.001	–	0.104	–	–
–	0.012	0.042	0.172	0.029	–	0.002	0.198	0.133	–	0.133	0.096
–	0.001	0.015	0.256	0.009	–	–	0.266	0.326	–	0.326	0.346
0.290	0.021	0.002	–	0.017	0.323	0.260	–	–	0.316	–	–
–	0.001	0.015	0.256	0.009	–	–	0.266	0.326	–	0.326	0.346
–	–	0.001	0.037	0.001	–	–	0.039	0.069	–	0.069	0.112

$(\Gamma_2 \circ Squaring)$ iterated four times on M

0.807	0.040	0.015	–	0.034	0.807	0.807	–	–	0.807	–	–
–	0.090	0.092	–	0.088	–	–	–	–	–	–	–
–	0.085	0.088	–	0.084	–	–	–	–	–	–	–
–	0.001	0.001	0.032	0.001	–	–	0.032	0.031	–	0.031	0.031
–	0.777	0.798	–	0.786	–	0.001	–	–	–	–	–
0.005	–	–	–	–	0.005	0.005	–	–	0.005	–	–
0.003	0.001	–	–	0.001	0.003	0.003	–	–	0.003	–	–
–	–	0.001	0.024	–	–	–	0.024	0.024	–	0.024	0.024
–	–	0.002	0.472	0.001	–	–	0.472	0.472	–	0.472	0.472
0.185	0.005	0.001	–	0.004	0.185	0.184	–	–	0.185	–	–
–	–	0.002	0.472	0.001	–	–	0.472	0.472	–	0.472	0.472
–	–	–	0.001	–	–	–	0.001	0.001	–	0.001	–

Considering to perform these operations infinite number of times, we definitely achieve convergence.



M_{mcl}^{∞}

2.5 Interpreting Clusters

To interpret clusters, the vertices are split into two types. Attractors, which attract other vertices, and vertices that are being attracted by the attractors. Attractors have at least one positive flow value within their corresponding row (in the steady state matrix). Each attractor is attracting the vertices which have positive values within its row.

Attractors and the elements they attract are swept together into the same cluster. In this case, {1, 6, 7, 10}, {2, 3, 5}, {4, 8, 9, 11, 12}

2.6 Time Complexity

O(N3), where N is the number of vertices.

–N3cost of one matrix multiplication on two matrices of dimension N.

–Inflation can be done in O(N2) time

–The number of steps to converge is not proven, but experimentally shown to be ~10 to 100 steps, and mostly consist of sparse matrices after the first few steps.

Speed may be improved through pruning.

–Inspect matrix and set small values directly to zero (assume they would have reached there eventually anyways).

–Works well when the diameter of the clusters is small. (Nonhomogeneous distributions of weight)

3. Girvan-Newman Algorithm

3.1 About the algorithm

The Girvan–Newman algorithm detects communities by progressively removing edges from the original network. The connected components of the remaining network are the communities. Instead of trying to construct a measure that tells us which edges are the most central to communities, the Girvan–Newman algorithm focuses on edges that are most likely "between" communities.

Vertex betweenness is an indicator of highly central nodes in networks. For any node , vertex betweenness is defined as the number of shortest paths between pairs of nodes that run through it. It is relevant to models where the network modulates transfer of goods between known start and end points, under the assumption that such transfer seeks the shortest available route.

The Girvan–Newman algorithm extends this definition to the case of edges, defining the "edge betweenness" of an edge as the number of shortest paths between pairs of nodes that run along it. If there is more than one shortest path between a pair of nodes, each path is assigned equal weight such that the total weight of all of the paths is equal to unity. If a network contains communities or groups that are only loosely connected by a few inter-group edges, then all shortest paths between different communities must go along one of these few edges. Thus, the edges connecting communities will have high edge betweenness (at least **one** of them). By removing these edges, the groups are separated from one another and so the underlying community structure of the network is revealed.

3.2 Steps Involved

The algorithm's steps for community detection are summarized below

1. The betweenness of all existing edges in the network is calculated first.
2. The edge with the highest betweenness is removed.
3. The betweenness of all edges affected by the removal is recalculated.
4. Steps 2 and 3 are repeated until no edges remain.

The fact that the only betweennesses being recalculated are only the ones which are affected by the removal, may lessen the running time of the process' simulation in computers.

However, the betweenness centrality must be recalculated with each step, or severe errors occur. The reason is that the network adapts itself to the new conditions set after the edge removal. For instance, if two communities are connected by more than one edge, then there is no guarantee that **all** of these edges will have high betweenness. According to the method, we know that **at least one** of them will have, but nothing more than that is known. By recalculating betweennesses after the removal of each edge, it is ensured that at least one of the remaining edges between two communities will always have a high value.

The end result of the Girvan–Newman algorithm is a dendrogram. As the Girvan–Newman algorithm runs, the dendrogram is produced from the top down (i.e. the network splits up into different communities with the successive removal of links). The leaves of the dendrogram are individual nodes.

3.3 Python Code

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


def edge_to_remove(G):
    dict1 = nx.edge_betweenness_centrality(G)
    print(dict1)
    print()
    list_of_tuples = dict1.items()
    list_of_tuples = sorted(list_of_tuples, key=lambda x: x[1], reverse=True)
    print(list(list_of_tuples))
    print()
    print(list(list_of_tuples)[0])
    print(list(list_of_tuples)[0][0])
    print()
    return list(list_of_tuples)[0][0]


def girvan(G):
    c = nx.connected_component_subgraphs(G)
    l = len(list(c))
    print('The number of connected components are ', l)

    while(l==1):
        G.remove_edge(*edge_to_remove(G))
        c = nx.connected_component_subgraphs(G)
        l = len(list(c))
        print('The number of connected components are ', l)

    return c


# G = nx.barbell_graph(5,2)
G = nx.karate_club_graph()
# G = nx.barbell_graph(10,5)

nx.draw_networkx(G)
```

```
plt.show()
c = girvan(G)


for i in c:
    print(i)
    print(i.nodes())
    print('....................')

nx.draw_networkx(G)
plt.show()
```

3.4 Results

3.4.1 Barbell Graph Implementation

The function barbell_graph(m1, m2) of the NetworkX library returns the Barbell Graph, i.e., two complete graphs connected by a path.

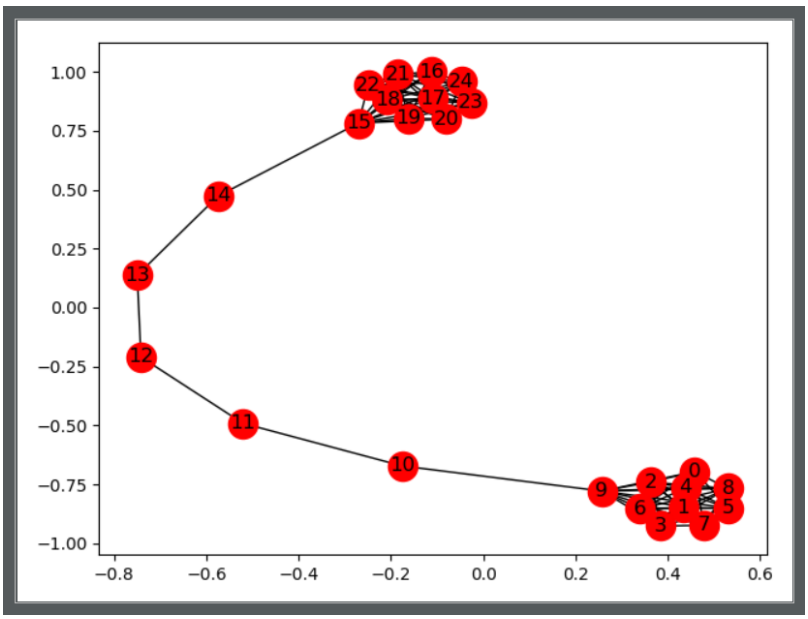


Fig 3.1 Barbell Graph with m1=10 and m2=5

For m1 > 1 and m2 >= 0. Two identical complete graphs K_{m1} form the left and right bells, and are connected by a path P_{m2}. The 2*m1+m2 nodes are numbered.

0, …, m1-1 for the left barbell, m1, …, m1+m2-1 for the path, and m1+m2, …, 2*m1+m2-1 for the right barbell. The 3 subgraphs are joined via the edges (m1-1,m1) and (m1+m2-1,m1+m2). If m2=0, this is merely two complete graphs joined together.

Girvan-Newman algorithm has been used for clustering of nodes.

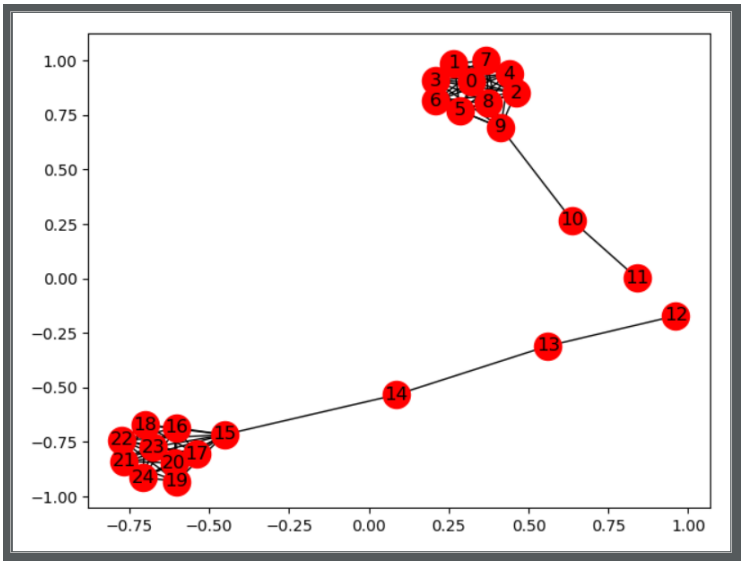


Fig 3.2 Girvan-Newman implementation for clustering

3.4.2 Zachary's Karate Club Implementation

A social network of a karate club was studied by Wayne W. Zachary for a period of three years from 1970 to 1972. The network captures 34 members of a karate club, documenting pairwise links between members who interacted outside the club. During the study a conflict arose between the administrator "John A" and instructor "Mr. Hi" (pseudonyms), which led to the split of the club into two. Half of the members formed a new club around Mr. Hi; members from the other part found a new instructor or gave up karate. Based on collected data Zachary correctly assigned all but one member of the club to the groups they actually joined after the split.

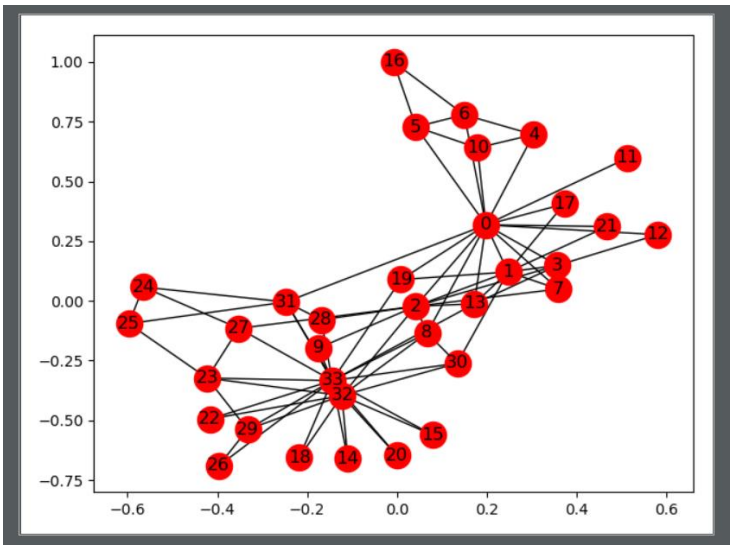


Fig 3.3 Zachary's Karate Club Graph

Before the split each side tried to recruit adherents of the other party. Thus, communication flow had a special importance and the initial group would likely split at the "borders" of the network. Zachary used the maximum flow – minimum cut **Ford–Fulkerson algorithm** from “source” Mr. Hi to “sink” John A: the cut closest to Mr. Hi that cuts saturated edges divides the network into the two factions. Zachary correctly predicted each member's decision except member #9, who went with Mr. Hi instead of John A.

Each integer represents one karate club member and a pair indicates the two members interacted. The data set is summarized below and also in the adjoining image. Node 1 stands for the instructor, node 34 for the club administrator / president.

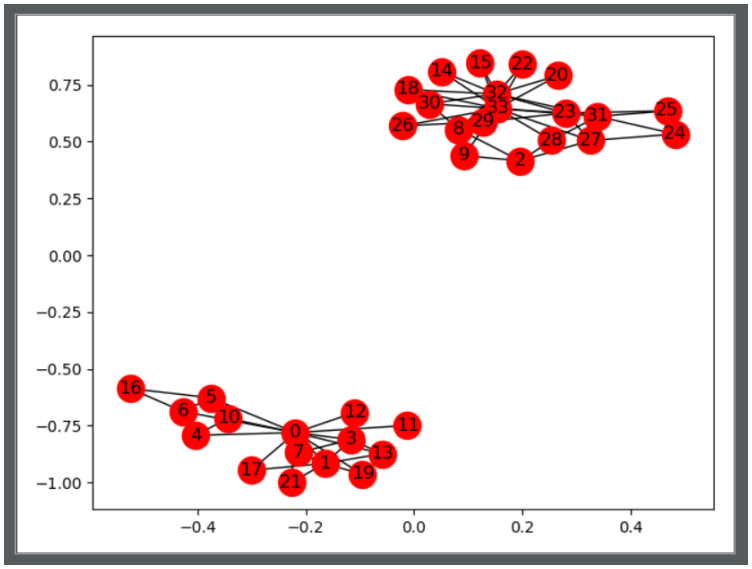


Fig 3.4 Girvan-Newman implementation for clustering

The Girvan-Newman algorithm has been used to predict the members of the two groups after the split.

4. Weighted Graph Clustering

4.1 Problem Statement and Motivation

Finding the structure of a community can be considered as a graph clustering problem, which in turn can be considered as an optimization problem.

We suppose that each person or a community has a density value and each pair of persons or communities has an attractiveness value. The social network is a graph, each person is a node, edges are the relationship between people. Given a sparse graph $G(V, E, W_V, S_E)$ where V is the node set, E is the edge set, W_V and S_E are the weights of node set and edge set respectively. We are interested in finding the clusters of G as communities.

Undirected graphs are the most common models of networks, where the directions of the connections are unimportant and can be safely ignored. Here we considered only undirected graph. The weight of node implies the core degree of the person in the network, and the weight of edge means the attractiveness between the two nodes. The result of graph clustering should partition a graph into several sub-graph(clusters), each part has a weight value, what's more, there are attractiveness values between clusters which similar with the edge weights. The candidate communities should have weights higher than the attractiveness with other clusters. The optimization objective function is equation, $\mathcal{P}$ is the partitions of a graph.

$$\operatorname{argmax}_{\mathcal{P}}\{\sum_{k\in\mathcal{P}} W(k) - \sum_{i,j\in\mathcal{P}} S(i,j)\}$$

4.2 Related Definitions

1) Density of cluster

Cluster density is the average of all the weights of nodes in the cluster. That is to say, if cluster i has Q_i nodes, each node's weight is W_a, $a \in 1,2, ..., Q_i$, then the cluster density of i is:

$$W_i = \frac{\sum_{a=1}^{Q_i} W_a}{Q_i}$$

2) Attractiveness between clusters

Attractiveness between clusters is the ration of the sum of all the edges' weights between the two clusters and the product of the node number of the two clusters. The number of edges between cluster i and j is q, the edge weight is S_e, e ∈ 1,2, ...,q , community i has Q_i nodes, and community j has Q_j nodes, then the attractiveness between cluster i and j is:

$$S_{ij} = \frac{\sum_{e=1}^{q} S_e}{Q_i \times Q_j}$$

3) Inter-interested clusters

If cluster i and cluster j are inter-interested clusters, then they must satisfy the following conditions:

$$q \geq Q_i,\ q \geq Q_j$$

4) Community

A cluster i can be a community, it must satisfy that:

$$S_{ij} < W_i + W_j,\ \forall j$$

Cluster j is the inter-interested cluster of cluster i.

4.3 Attractiveness Based Community Detection Algorithm (ABCD)

4.3.1 Procedure

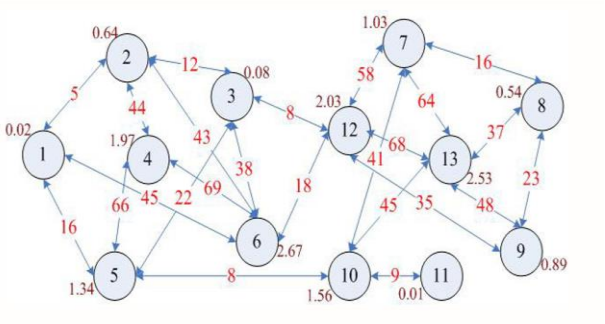
Initially, each node is considered as a single cluster. The algorithm is an agglomerative algorithm. If we want to do the merger for cluster i, firstly, we need to find which cluster among all its inter-interested clusters would get the highest attractiveness with it, which will be denoted by j. But after finding the two clusters, we can not do the merger directly, because the attractiveness between them may be very small, meaning that they may not be of the same community, so we have to make some other judgment. Only S_{ij} meets the condition:

$$Sij \geq Wi + Wj$$

The cluster i and j will be merged. In addition, there may be two special cases during the merger. The first is that cluster I may not have inter-interested clusters, then cluster i will not merge with any other clusters, it will be a community; the second is that there are more than

one clusters have the highest attractiveness with cluster i, and satisfies the expression at the same time, then we merge cluster i with any one of them. Cluster attractiveness matrix S is a k-order matrix, where $S_{ij} = S_{ji}$ denotes the attractiveness between the cluster i and j, k is changing in every iteration. Assuming that the total number of node is n, then the attractiveness matrix S is a n-order matrix at the begin. The matrix S below shows the attractiveness of the graph shown in Figure beside the matrix S. Since the matrix S is sparse, so we can use the triplet to store the elements of the matrix.

$$S = \begin{pmatrix} 0 & 5 & 0 & 0 & 16 & 45 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 12 & 44 & 0 & 43 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 12 & 0 & 0 & 22 & 38 & 0 & 0 & 0 & 0 & 0 & 8 & 0 \\ 0 & 44 & 0 & 0 & 66 & 69 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 16 & 0 & 22 & 66 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 \\ 45 & 43 & 38 & 69 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 18 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 0 & 41 & 0 & 58 & 64 \\ 0 & 0 & 0 & 0 & 0 & 0 & 16 & 0 & 23 & 0 & 0 & 0 & 37 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 23 & 0 & 0 & 0 & 35 & 48 \\ 0 & 0 & 0 & 0 & 8 & 0 & 41 & 0 & 0 & 0 & 9 & 0 & 45 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 & 18 & 58 & 0 & 35 & 0 & 0 & 0 & 68 \\ 0 & 0 & 0 & 0 & 0 & 0 & 64 & 37 & 48 & 45 & 0 & 68 & 0 \end{pmatrix}$$



The number of clusters would be reduced after merging, the density of clusters will change, and the attractiveness between clusters will also change accordingly, so it is necessary to update the matrix S. When we update the new attractiveness matrix, we can make use of the old one to reduce the amount of calculation. Before the merge, the collection of clusters is CM_{pre}, the corresponding attractiveness matrix is S', the number of clusters is k'; after the merger, the collection of clusters is CM_{cur}, and the corresponding attractiveness matrix is S, the number of clusters is k. We use the following mathematical expression to denote CM_{pre}.

$$CM_{pre} = \{cm_l | l = 1, 2, ..., k'\}$$

Where cml denotes the l-th cluster in CM_{pre} . If the cluster p in CM_{cur} contains a number of clusters in CM_{pre} , and the number is m, that is cluster p in CM_{cur} is formed by the merger of m clusters in CM_{pre}, the mathematical expression is:

$$CM_{cur}^{p} = \{cm_t | cm_t \in CM_{pre}, t = 1, 2, ..., m\}$$

Where CM^{p}_{cur} denotes the cluster p in CM_{cur} . Then, the attractiveness between community i and community j, that is the element S_{ij} in matrix S, can be updated by formula:

$$S_{ij} = \frac{\sum_{cm_r \in CM^i_{cur}, cm_t \in CM^j_{cur}} S'_{cm_r, cm_t} \times Q_{cm_r} \times Q_{cm_t}}{Q_i \times Q_j}$$

Updating the elements in S one by one with the above-mentioned formula, then we get the new attractiveness matrix. ABCD algorithm can be divided into **two main steps**, iterating between the two steps to get clusters:

1. Merge the pair of clusters which has the largest attractiveness.

2. Calculate or update the cluster density and cluster attractiveness matrix;

Executing the update of cluster density and attractiveness matrix, and the cluster merger process iteratively, until the **structure of clusters does not change**, or **there is only one cluster left.**

4.3.2 Time Complexity

In initial, the time required to calculate attractiveness matrix is O(nk), where n is the number of nodes, and k denotes the average number of inter-interested nodes for all nodes. The time consuming of merger of each iteration is $O(m_i)$, the time for updating attractiveness matrix is $O(m^2_i)$, so the time complexity of each iteration is $O(m^2_i)$, where m_i denotes the number of clusters at the beginning of i-th iteration. The maximum number of iterations is t , so the total time complexity of ABCD algorithm is $O(nk + tm^2)$.

Based on the experimental results, the number of iterations is much smaller than the number of nodes, especially for large-scale network, the number of merger is of several orders of magnitude smaller than the number of nodes.

5. Social Network Analysis

5.1 Introduction

Social network analysis (**SNA**) is the process of investigating social structures through the use of networks and graph theory. It characterizes networked structures in terms of nodes (individual actors, people, or things within the network) and the ties, edges, or links (relationships or interactions) that connect them. Examples of social structures commonly visualized through social network analysis include social media networks, memes spread, friendship and acquaintance networks, collaboration graphs, kinship, disease transmission, and sexual relationships. These networks are often visualized through sociograms in which nodes are represented as points and ties are represented as lines.

Visual representation of social networks is important to understand the network data and convey the result of the analysis. Many of the analytic softwares have modules for network visualization. Exploration of the data is done through displaying nodes and ties in various layouts, and attributing colours, size and other advanced properties to nodes. Visual representations of networks may be a powerful method for conveying complex information, but care should be taken in interpreting node and graph properties from visual displays alone, as they may misrepresent structural properties better captured through quantitative analysis.

Social network analysis is used extensively in a wide range of applications and disciplines. Some common network analysis applications include data aggregation and mining, network propagation modelling, network modelling and sampling, user attribute and behaviour analysis, community-maintained resource support, location-based interaction analysis, social sharing and filtering, recommender systems development, and link prediction and entity resolution.

5.2 Key Ideas

The following models and algorithms give us an idea about how social network analysis can be very helpful in conveying the result of an analysis.

1. **The Contagions**
   Is obesity contagious? If your friend is obese, will you also get obese?
   Is happiness contagious? The answer is yes. If your friend's friend is happy, then so will you be. We will be exploring this concept with the help of the Fat-man Evolution Model.
2. **Searching in a Network**
   How do you search a node in a billion nodes. It is actually not that difficult taking into consideration, a social network. It just takes log(n) steps, i.e., just 10 steps for 1 billion nodes. This is known as the Small World Phenomenon.
3. **Google Page Rank**
   This relates to ranking the nodes in a network. This helps us understand which is the most important person (node) in the friendship (network) or how the search results are ranked by Google.

5.3 Granovetter's Strength of weak ties

In real world, there is less chance of you getting information about a job/or referral for a job from a friend or someone you know than from an acquaintance, i.e., someone with whom you have weak ties. This happens in more than 90% of the cases.

Let A, B, C, D and E are my friends. A, B, C, D are in the same company as me. So, I know almost everything they know, but my friend E being in a different company has a different world altogether and what he knows is something new for me. Therefore, we are more likely to get new information or job referrals from someone who is not so close to us. In this way, weak ties prove to be very strong in some cases.

6. Fat-Man Evolutionary Model

6.1 Introduction

The Fat-Man evolutionary model is based on three important concepts:

1. Homophily – 'Like attracts like'
2. Closures – Triadic, Focal and Membership
3. Social Influence

Fat-Man Hypothesis: "If your friends are fat, then the probability of you gaining weight increases."

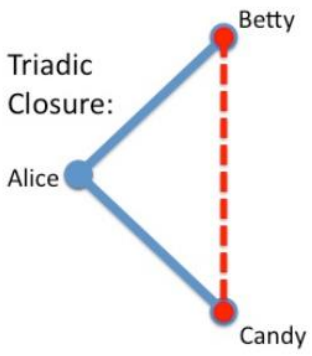
We'll be modelling a city with different people having different body weights and BMI values. Now we'll define the above-mentioned concepts in our context.

Homophily: People having similar body weights tend to become friends with each other.

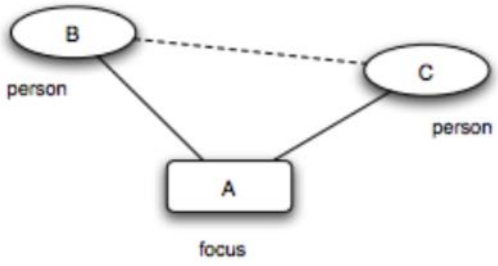
Triadic Closure: Two people having common friends, tend to become friends with each other.

Focal Closure: Two people having the same social foci, i.e., going to the same gym or restaurant, tend to become friends with each other.

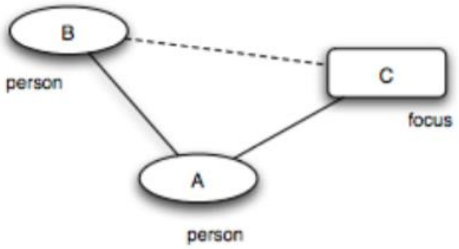
Membership Closure: Suppose my friend goes to a food outlet, so I also start going to that food outlet (and start gaining weight).



(a) Triadic Closure



(b) Focal Closure



(c) Membership Closure

Fig 6.1 Types of Closure in a social-affiliation network

Assumption for the model:

Previous edges of the network don't get deleted with time, only new edges get added to the network.

The Model:

So, we take 50 people in a city and assign random BMI values, ranging from 15 to 40, to all the people, where a BMI of 15 implies 'Underweight' and a BMI of 40 implies 'Obese'.

There are two kinds of nodes in the network – people and the social foci, they are associated with. We have 5 social foci including a gym and a food outlet. Initially, everybody is a part of only one social focus and with time, people start becoming part of more social foci because of membership closure, which in turn affects their BMI values.

6.2 Homophily and Triadic Closure

The probability of two people becoming friends is inversely proportional to the difference between their BMIs. Let X and Y be two nodes representing 2 people, then:

$$P(Edge\ between\ X\ and\ Y) \propto \frac{1}{BMI(X) - BMI(Y) + c}$$

The arbitrary constant c is added to the denominator so that the probability doesn't tend towards infinity when X and Y have the exact same BMI.

Now, a question arises. Does the chance of friendship increase between two people if they have a lot of common friends? The answer is yes.

Let p be the probability of two people becoming friends when they have 1 common friend.

Then,

(1-p): Probability of two people not becoming friends when they have a common friend

$(1-p)^k$: Probability of two people not becoming friends when they have k common friends

$1-(1-p)^k$: Probability of two people becoming friends when they have k common friends

It is quite intuitive from the expression of probability that as the value of k, i.e., the number of common friends increases, the probability of friendship between two people goes higher and higher.

6.3 Social Influence and Analysis

We know that if a person is connected to a gym, he is expected to have a lower BMI value and one who is connected to a food outlet is expected to have a higher BMI value. For every node, which is connected to a gym, we reduce its BMI value by 1 in each iteration and every node, which is connected to a food outlet, we increase its BMI value by 1 in each iteration. This captures the social influence, the reason being shared context.

Example: If two students have a brilliant teacher, both will start scoring well in exams.

Similarly, here also social influence plays a major role.

Now, what actually do we want to analyse with the help of this model?

We want to analyse: -

1. Change in density of the network.
2. How the number of obese people changes with time.
3. How the number of edges between these obese people changes with time.

6.4 Python Code

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

def create_graph():
    G = nx.Graph()

    for i in range(1,50):
        G.add_node(i)
    return G

def visualize(G,t):
    time.sleep(1)
    labeldict = get_labels(G)
    nodesize = get_size(G)
    color_array = get_colors(G)
```

```
    print(labeldict)
    pos = nx.spring_layout(G, k=0.15, iterations=20)

nx.draw_networkx(G,labels=labeldict,pos=pos,node_size=nodesize,node_color=color_arr
ay)
    # plt.figure(figsize=(100, 50),dpi=180)
    plt.savefig('C:\\Users\\Naman\\Desktop\\Fatman
Hypothesis\\evolution_'+str(t)+'.jpg')
    plt.clf()
    plt.cla()
    nx.write_gml(G,'C:\\Users\\Naman\\Desktop\\Fatman
Hypothesis\\evolution_'+str(t)+'.gml')



def assign_bmi(G):
    for each in G.nodes():
        G.node[each]['name'] = random.randint(15,40)
        G.node[each]['type'] = 'person'

def get_labels(G):
    dict1 = {}
    print(G.nodes())
    for each in G.nodes():
        dict1[each] = G.node[each]['name']
    return dict1


def get_size(G):
    array1 = []
    for each in G.nodes():
        if(G.node[each]['type']=='person'):
            array1.append(G.node[each]['name']*10)
        else:
            array1.append(800)
    return array1


def add_foci_nodes(G):
    n = G.number_of_nodes()
    i=n+1
    foci_nodes = ['gym','food_outlet','movie_club','karate_club','yoga_club']
    for j in range(0,5):
        G.add_node(i)
        G.node[i]['name'] = foci_nodes[j]
        G.node[i]['type'] = 'foci'
        i=i+1


def get_colors(G):
    c=[]
    for each in G.nodes():
        if(G.node[each]['type']=='person'):
            if(G.node[each]['name']==15):
                c.append('green')
            elif(G.node[each]['name']==40):
                c.append('yellow')
            else:
                c.append('blue')
        else:
            c.append('red')
    return c


def get_foci_nodes():
    f=[]
    for each in G.nodes():
        if(G.node[each]['type']=='foci'):
```

```
            f.append(each)
    return f


def get_persons_nodes():
    p=[]
    for each in G.nodes():
        if(G.node[each]['type']=='person'):
            p.append(each)
    return p


def add_foci_edges():
    foci_nodes = get_foci_nodes()
    person_nodes = get_persons_nodes()
    for each in person_nodes:
        r = random.choice(foci_nodes)
        G.add_edge(each,r)


def homophily(G):
    pnodes = get_persons_nodes()
    for u in pnodes:
        for v in pnodes:
            if(u!=v):
                diff = abs(G.node[u]['name']-G.node[v]['name'])
                p = (float)(1)/(diff+1000)
                r = random.uniform(0,1)
                if(r<p):
                    G.add_edge(u,v)


def cmn(u,v,G):
    nu = set(G.neighbors(u))
    nv = set(G.neighbors(v))
    return len(nu & nv)

def closure(G):
    array1=[]
    for u in G.nodes():
        for v in G.nodes():
            if(u!=v and (G.node[u]['type']=='person' or
G.node[v]['type']=='person')):
                k = cmn(u,v,G)
                p = 1-pow((1-0.01),k)
                tmp=[]
                tmp.append(u)
                tmp.append(v)
                tmp.append(p)
                array1.append(tmp)
    for each in array1:
        u = each[0]
        v = each[1]
        p = each[2]
        r = random.uniform(0,1)
        if(r<p):
            G.add_edge(u,v)


def change_bmi(G):
    fnodes = get_foci_nodes()
    for each in fnodes:
        if(G.node[each]['name']=='food_outlet'):
            for each1 in G.neighbors(each):
                if(G.node[each1]['name']!=40):
                    G.node[each1]['name'] += 1
        if (G.node[each]['name'] == 'gym'):
            for each1 in G.neighbors(each):
```

```
                if (G.node[each1]['name'] != 15):
                    G.node[each1]['name'] -= 1

G = create_graph()
assign_bmi(G)
add_foci_nodes(G)

add_foci_edges()

time.sleep(10)
t=0
visualize(G,t)
# nx.write_gml(G,'evolution_0.gml')
for t in range(0,10):
    # visualize(G)
    homophily(G)
    # visualize(G)
    closure(G)
    change_bmi(G)
    visualize(G,t+1)
```

6.5 Results

6.5.1 Graph Visualisation

The red nodes depict social foci, blue coloured nodes depict people with their BMI value displayed inside the node. Green colour is for underweight people, having BMI value of 15 and yellow colour is for obese people, having BMI value of 40.

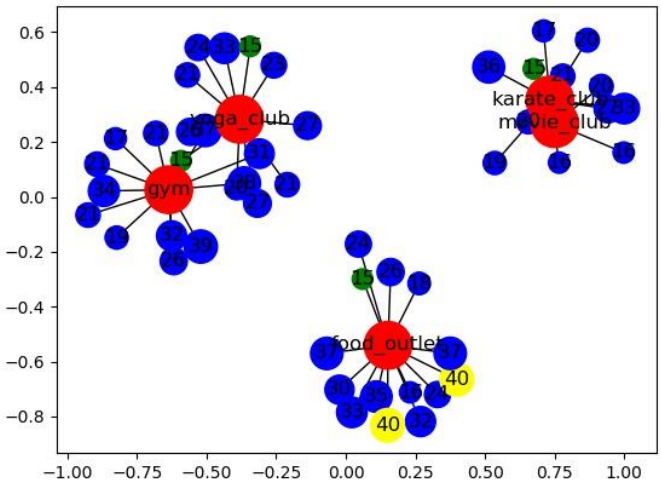
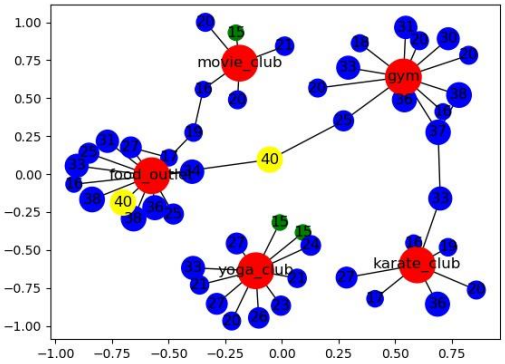
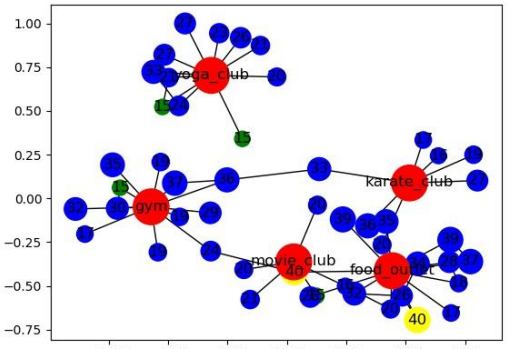
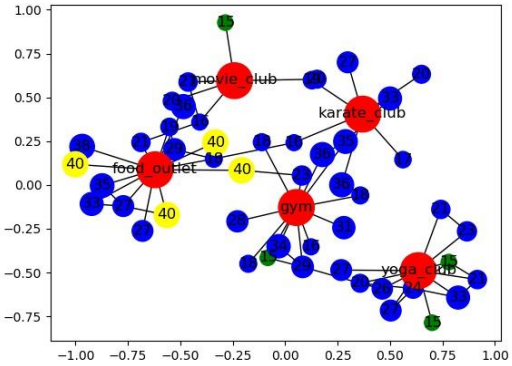
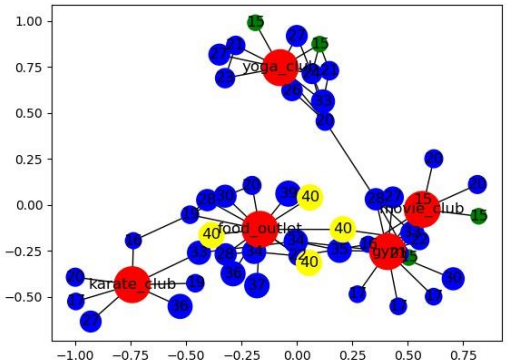


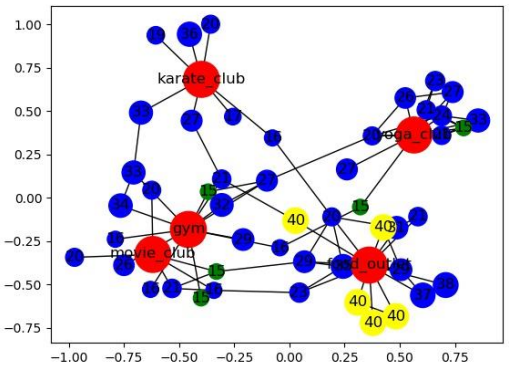
Fig 6.2 Initial Graph

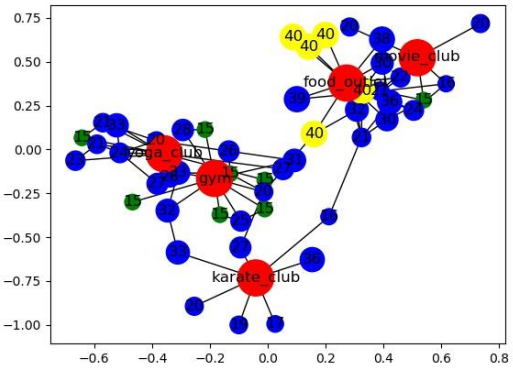

movie_club
gym
40
food_outlet
40
yoga_club
karate_club

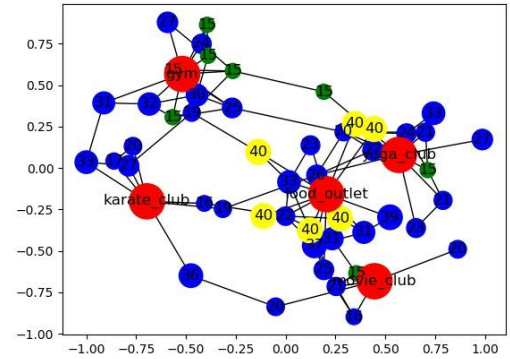

yoga_club
karate_club
gym
movie_club
food_outlet
40

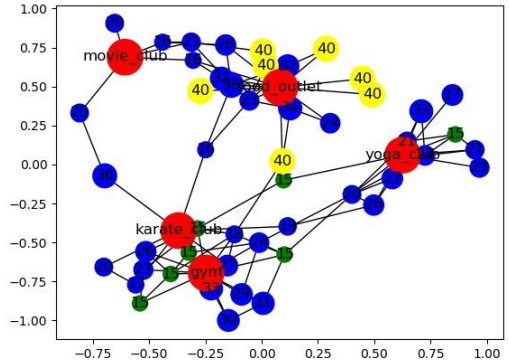

movie_club
karate_club
40
40
food_outlet
40
gym
40
yoga_club


yoga_club
40
movie_club
food_outlet
40
40
gym
40
karate_club


karate_club
gym
40
movie_club
40
40
40
40


40
40
movie_club
food_outlet
40
40
gym
karate_club


gym
40
40
40
karate_club
40
40
movie_club


movie_club
40
40
40
40
40
40
yoga_club
40
karate_club
gym

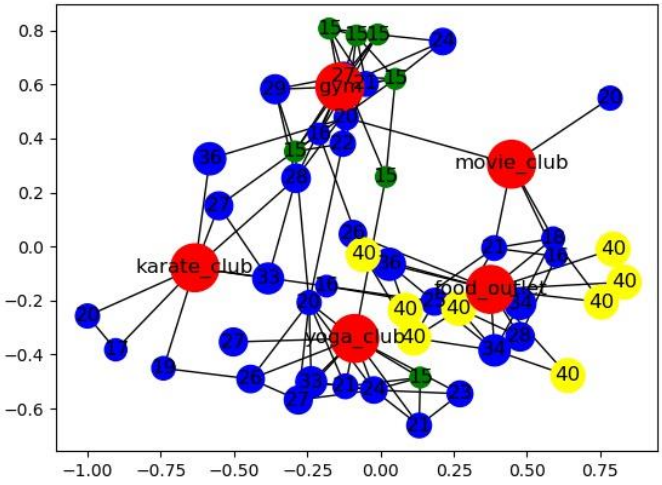


Fig 6.3 Graphs obtained by 10 subsequent iterations of the algorithm

It can be observed that more and more people are getting associated with the food outlet with time and hence their BMI value is going up. This is the reason why the number of obese people, i.e., the yellow nodes in the graph can be seen increasing with time.

Similar is the case with the green nodes, i.e., the underweight people.

6.5.2 Change Plots

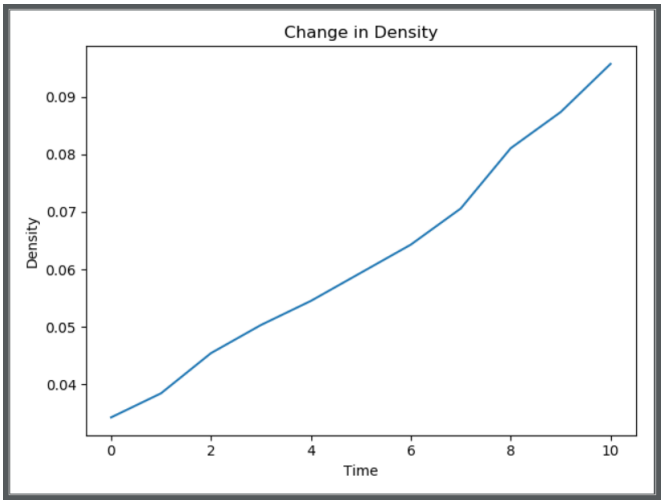


Fig 6.4 Density vs Time Graph

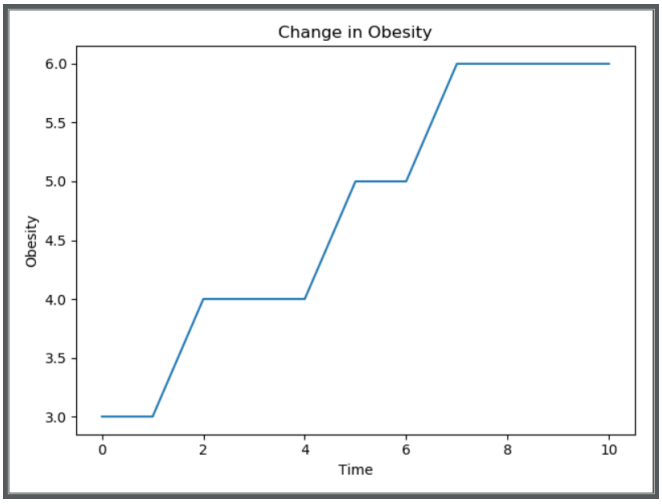


Fig 6.5 Number of Obese People vs Time

The results obtained by our model are clearly in favour of the Fat-Man Hypothesis.

7. Small World Phenomenon

7.1 Introduction

Is the world connected? The answer to this question is surprisingly yes.

According to the small world phenomenon, on an average, only around six people are needed to connect any two people in the world.

It can basically be considered as a search algorithm as the first person needs to know how to get to the second person.

7.2 Milgram's Experiment

Milgram's experiment was conceived in an era when a number of independent threads were converging on the idea that the world is becoming increasingly interconnected.

Fig 7.1 One possible path of message in Milgram's experiment

Information packets were initially sent to "randomly" selected individuals. They included letters, which detailed the study's purpose, and basic information about a target contact person. The letter was not randomly passed on to any of an individual's friend but to a friend who was closer to the target. He observed that in six steps, on an average, the letter reached from the starting person to the target who didn't know each other at all.

From this, we conclude that though the world is big, but it is surprisingly small. 'Big', as in, there are around 7 billion people in the world and 'Small', as in, on an average only six people can connect any 2 people in the world.

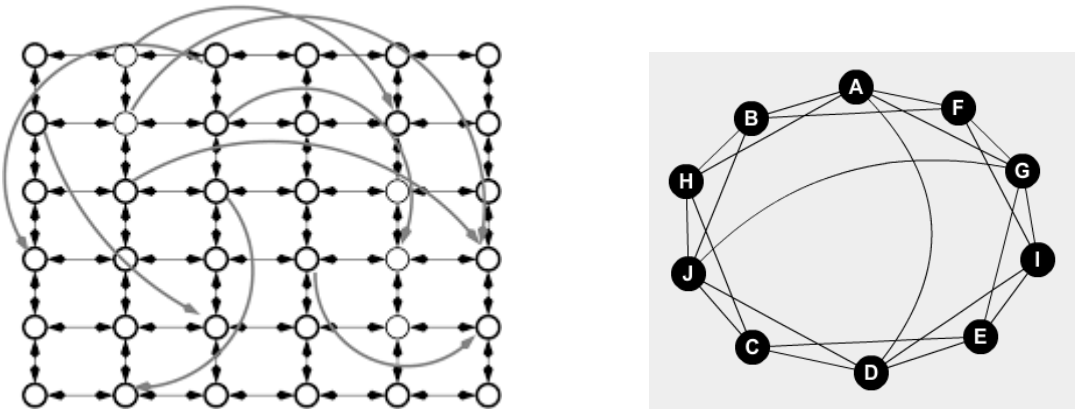
Watts-Strogatz also conducted a similar experiment on email networks and observed the small world phenomenon. They concluded that the two basic concepts due to which this phenomenon was observed are homophily and weak ties.

7.3 Watts-Strogatz Generative Model

7.3.1 The Model

It is an algorithm or a way in which we can construct graphs with small world properties.

Firstly, we consider a cycle, put all the nodes in a circle and then start drawing edges to their nearest neighbours (4 edges for each node). This is similar to a grid network where basically people know others close to them. Then, we randomly rewire few edges. The resultant network is indeed a small world, i.e., any 2 nodes in the network are very close to each other, even if the number of nodes is very large.



(a) 2D grid network *(b) 1D network*

Fig 7.2 Small world networks showing strong and weak ties

It is fascinating that:

1. The world is connected.
2. Not only this, any two people in the world are connected by an average of six links.
3. The search algorithm that people use to find the destination is a local method. ‘Local method’ means that nobody has the global picture of the network that who is friends with whom, they just do their local job, i.e., send the letter to their immediate neighbour.

There is no central authority but still we are able to search. We always choose the next person such that he is closer to the destination. Generally, a person has a lot of links in his locality

and also a few links outside, which are known as weak ties. Random rewiring generates the weak ties in the network of our model. **Addition of weak ties decreases the diameter of a network** gradually. The diameter of a graph is the maximum eccentricity of any vertex in the graph. That is, it is the greatest distance between any pair of vertices.

In Fig. 7.2 above, straight lines depict strong ties that are with immediate neighbours or close friends and curved lines depict weak ties.

7.3.2 Myopic/Decentralized Search

The word ‘myopic’ is derived from the word ‘myopia’ which means short-sightedness, i.e., you look at only short distances and not long distances. This is what happens in a decentralized search, we don’t know about the entire network, we know only about our locality and based on our locality, we decide where to pass the letter. Hence, the name ‘Myopic search’.

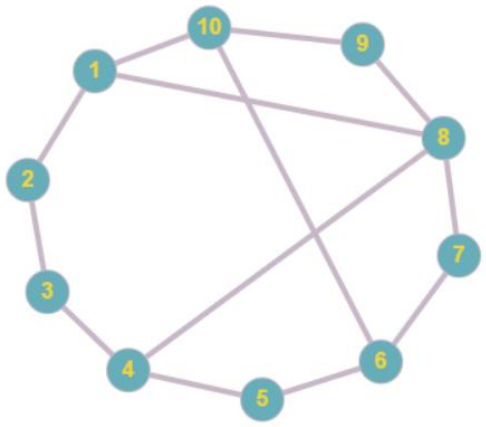


Fig 7.3 Small World Network

In Fig. 7.3, suppose node 1 is the source and node 6 is the target. Now, 1 will look for its neighbours that are close to 6, so it goes to 10 as it is closer. It is important to note that 1 will not have any information about the weak ties of its neighbours. A person will only know his own weak ties. So, 1 can pass on the letter to 8, 10 or 2, based on their distance from the target.

Distance of 2 from target = 4

Distance of 10 from target = 4 (as 1 doesn’t know about 10’s weak ties.)

Distance of 8 from target = 2

Therefore, the length of this path (1-8-7-6) in myopic search is 3.

But, is this search optimal, i.e., did we travel along the best path possible? The answer is no.

Optimal path here would be 1-10-6, i.e., of length 2. Optimal path can be found only when we have global knowledge of the complete network, but when we have information only about our locality, which actually happens in the real world, the myopic search seems more realistic and appealing.

7.3.3 Python Code

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

def add_edges(G):
    list_nodes = []
    for i in G.nodes():
        list_nodes.append(i)
    n = G.number_of_nodes()
    # print(list_nodes)

    for i in range(0, len(list_nodes)):
        G.add_edge(list_nodes[i], list_nodes[i-1])
        G.add_edge(list_nodes[i], list_nodes[i-2])
        target = i+1
        if(target>n-1):
            target = target-n
            G.add_edge(list_nodes[i], target)
        else:
            G.add_edge(list_nodes[i], target)

        target = i+2
        if (target>n-1):
            target = target-n
            G.add_edge(list_nodes[i], target)
        else:
            G.add_edge(list_nodes[i], target)


def add_long_link(G):
    v1 = random.choice(np.array(G.nodes()))
    v2 = random.choice(np.array(G.nodes()))
    while(v1==v2):
        v1 = random.choice(np.array(G.nodes()))
        v2 = random.choice(np.array(G.nodes()))
    G.add_edge(v1,v2)

def find_best_neighbour(G,c,v):
    dis = G.number_of_nodes()
    for each in G.neighbors(c):
        dis1 = len(nx.shortest_path(H,source=each,target=v))
        if(dis1<dis):
            dis=dis1
            choice=each
    return choice


def myopic_search(G,u,v):
    path = [u]
```

```
    current = u
    while(1):
        w = find_best_neighbour(G, current, v)
        path.append(w)
        current = w
        if(current==v):
            break
    return path


def set_path_colors(G,p,p1):
    c=[]
    for each in G.nodes():
        if(each==p1[0]):
            c.append("red")
            print(each)
        elif(each==p1[len(p1)-1]):
            c.append("red")
            print(each)
        elif(each in p and p1 and each!=p1[0] and each!=p1[len(p1)-1]):
            c.append("yellow")
        elif(each in p and each not in p1):
            c.append("blue")
        elif(each in p1 and each not in p):
            c.append("green")
        elif(each not in p1 and each not in p):
            c.append("black")

    return c

x1=[]
y1=[]

# for num in [100,200,300,400,500,600,700,800,900,1000]:
for num in [100,200,300,400,500]:
    G = nx.Graph()
    G.add_nodes_from(range(0,num))

    # Add ties based on Homophily
    add_edges(G)
    # for each in G.nodes():
    #     print(each, ':', end=' ')
    #     for each1 in G.neighbors(each):
    #         print(each1, end=' ')
    #     print('\n')

    # Add weak ties
    # add_long_link(G)

    H = G.copy()

    x = [0]
    y = [nx.diameter(G)]
    t=0

    # 10% of number of nodes
    while(t<=G.number_of_nodes()/10):
        add_long_link(G)
        t=t+1
        x.append(t)
        y.append(nx.diameter(G))
    plt.title("For "+str(num)+" Nodes")
    plt.xlabel("Number of weak ties added")
    plt.ylabel("Diameter")
    plt.plot(x,y)
    plt.show()

    m=[]
```

```
    o=[]
    x=[]
    t=0

    # diametrically opposite points
    for u in range(0,(int)(G.number_of_nodes()/2)-1):
        v=u+G.number_of_nodes()/2
        p = myopic_search(G,u,v)
        p1 = nx.shortest_path(G,source=u,target=v)
        m.append(len(p))
        o.append(len(p1))
        x.append(t)
        t=t+1
    print(G.number_of_nodes(), np.average(m))
    y1.append(np.average(m))
    x1.append(G.number_of_nodes())
    plt.plot(x, m, 'r')
    plt.plot(x, o, 'b')
    plt.xlabel('Node')
    plt.ylabel('Path Length')
    plt.gca().legend(('myopic', 'optimal'))
    plt.show()
    colors = set_path_colors(G, p, p1)
    nx.draw_networkx(G, node_color=colors)

    plt.show()

plt.plot(x1,y1)
plt.xlabel('Number of nodes')
plt.ylabel('Average myopic path length')
plt.show()




print(p1)
print(p)


print(colors[0],colors[40])
pos = nx.circular_layout(G)
```

7.3.4 Results

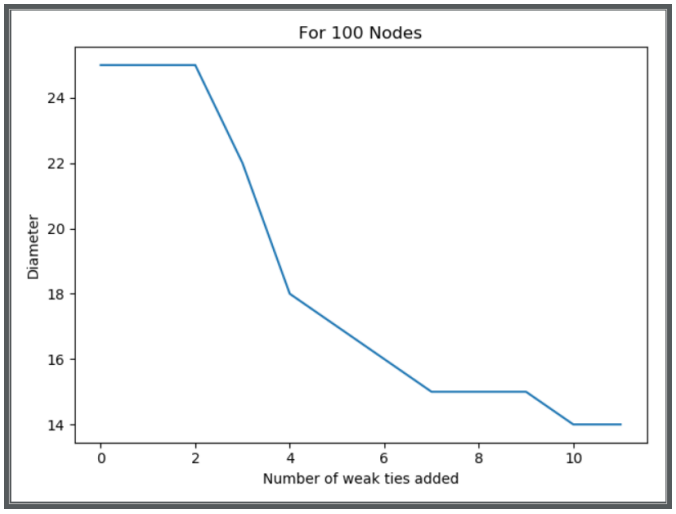


Fig 7.4 Change in diameter of network with addition of weak ties

The addition of weak ties increases the chance of reach of a node to a far-off node, thereby decreasing the diameter of the network.

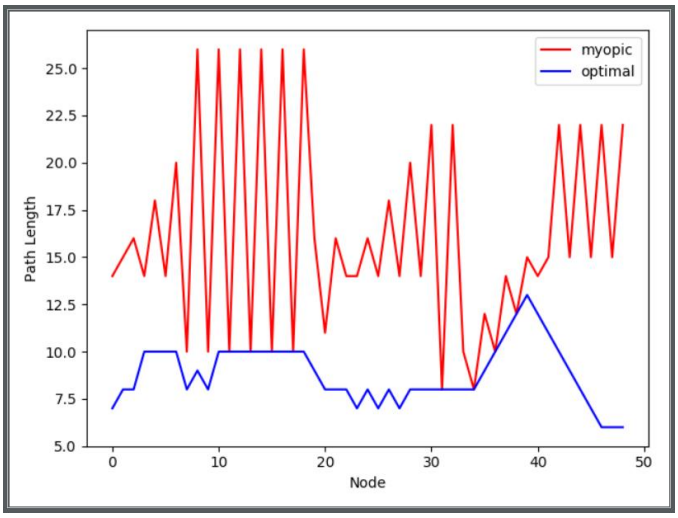


Fig 7.5 Comparison between myopic and optimal search

From Fig 7.5, it can be observed that the myopic path length is always greater than or equal to the optimal path length from the source to the target node. Here we have considered 100 nodes, and path lengths are calculated from a node to its diametrically opposite node, considering circular layout of the network. E.g. The y-axis values corresponding to 0 of x-axis depict the path lengths considering 0^{th} node as the source and 49^{th} node as the target.

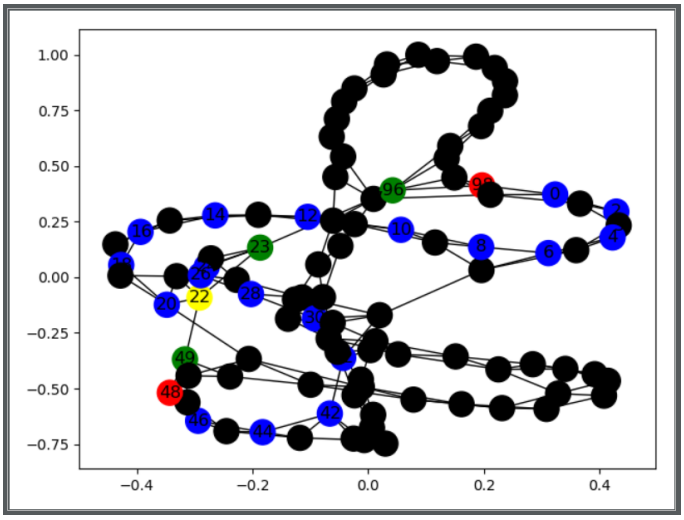


Fig 7.6 Graph showing myopic and optimal path

In Fig. 7.6, the red nodes depict the source and the target. The nodes common in optimal and myopic path are shown in yellow, nodes that are in optimal path but not in myopic path are

shown in green, nodes that are in myopic path but not in optimal path are shown in blue and the nodes that are in neither of the paths are shown in black colour.

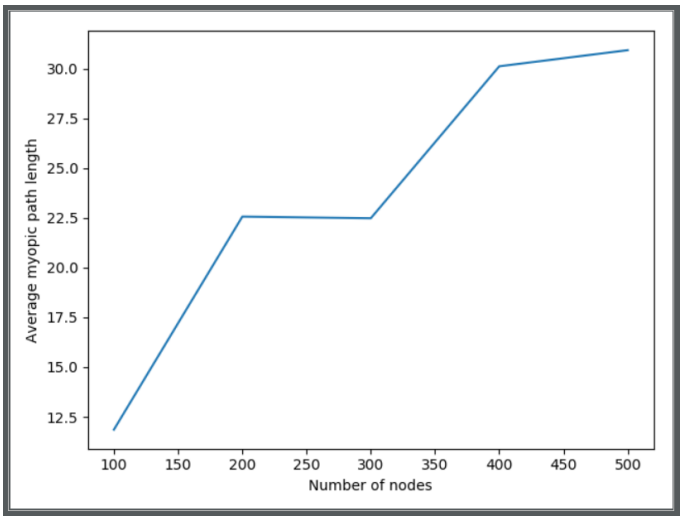


Fig 7.7 Plot showing increase in average path length with nodes in the network

In this model, we have added our own random number of weak ties, therefore we are getting on an average around 30 steps to reach from 1 person to another in a network of 500 people, as can be observed in Fig. 7.7. But, in real life, the number of weak ties is much greater than our assumption, and hence according to Watts-Strogatz, we have on an average 6 degrees of separation between any two people in the real world. The Watts-Strogatz model also implies a fixed number of nodes and thus cannot be used to model network growth.

8. PageRank Algorithm

8.1 Introduction

PageRank algorithm is applied on a web graph which is a directed graph consisting of web pages as nodes linked by hyperlinks.

We are interested in knowing that whenever we search something on Google, on what basis does Google rank its pages, that are the search results.

PageRank assigns a rank or score to every search result. The higher the page's score, the further up the search results list, it will appear. The scores are partially determined by the number of web pages that link to the target page. Each link is counted as a vote for the target.

The underlying assumption is that more important websites are likely to receive more links from other websites.

The PageRank algorithm is implemented using the **Points Distribution Method**. We start with assigning fixed equal number of points to each node. After that, every node distributes its points to its neighbours, that are connected to it by its out links. Point distribution happens for every node in the graph.

8.2 Steps Involved

1. Create a directed graph with n nodes.

2. Assign 100 points to each node.

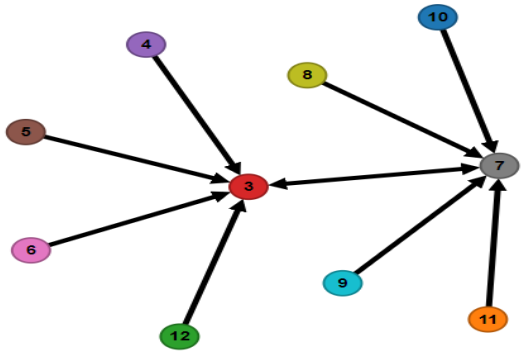
3. Keep distributing the points until convergence is achieved.

4. Higher the indegree of a node, higher is the amount of points accumulated by that node in subsequent iterations.

5. Get nodes' ranking as per the points accumulated.

8.3 Handling Points Sink

There may be a case when a situation as shown in graph below arises in a big network, i.e., one or more nodes have no out links or two nodes have directed edges towards each other. Such nodes accumulate all the points in subsequent iterations and hence lead to distorted results.



Here the nodes 3 and 7 will accumulate all the points, which may give inaccurate results in the form of rankings. This case may not arise in all the graphs, but we must handle it.

So, we let the node have 80% of its points and the rest 20% of its points are distributed equally among all the other nodes including itself. This is similar to the implementation of the tax system in a country, like 30% of the salary is taken from people who earn more than a threshold and that money is used for the development and welfare of all the people of the country.

8.4 Python Code

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

def add_edges(G,p):
    for i in G.nodes():
        for j in G.nodes():
            if(i!=j):
                r = random.random();
                if(r<=p):
                    G.add_edge(i,j)
                else:
                    continue
    return G


def initialize_points(G):
    points = [100 for i in range(G.number_of_nodes())]
    return points
```

```
def distribute_points(G, points):
    prev_points = points
    new_points = [0 for i in range(G.number_of_nodes())]

    for i in G.nodes():
        out = G.out_edges(i)
        if(len(out)==0):
            new_points[i] += prev_points[i]
        else:
            share = (float)(prev_points[i])/len(out)
            for each in out:
                new_points[each[1]] += share
    return G, new_points


def handle_points_sink(G, points):
    for i in range(len(points)):
        points[i] = (float)(points[i])*0.8

    n = G.number_of_nodes()
    extra = ((float)(n)*100*0.2)/n
    for i in range(len(points)):
        points[i] += extra
    return points

def keep_distributing_points(G, points):
    prev_points = points
    print("Enter # to stop")

    while(1):
        G, new_points = distribute_points(G, prev_points)
        print(new_points)

        new_points = handle_points_sink(G, new_points)
        char = input()
        if(char=='#'):
            break
        prev_points = new_points
        nx.draw_networkx(G,node_size=np.array(prev_points)*10)
        plt.show()

    return G, new_points


def get_nodes_sorted_by_points(points):
    points_array = np.array(points)
    nodes_sorted_by_points = np.argsort(-points_array)
    return nodes_sorted_by_points

def main():
    G = nx.DiGraph()
    G.add_nodes_from([i for i in range(10)])
    G = add_edges(G,0.3)

    points = initialize_points(G)
    print(points)
    nx.draw_networkx(G, node_size=np.array(points) * 10)
    plt.show()

    G, points = keep_distributing_points(G, points)

    nodes_sorted_by_points = get_nodes_sorted_by_points(points)
    print("nodes_sorted_by_points", nodes_sorted_by_points)

    #Compare to inbuilt function
    pr = nx.pagerank(G)
```

```
    pr_sorted = sorted(pr.items(),key = lambda x:x[1], reverse = True)
    for i in pr_sorted:
        print(i[0], end=' ')


main()
```

8.5 Convergence

[100, 100, 100, 100, 100, 100, 100, 100, 100, 100]

Enter # to stop

[66.66666666666667, 225.00000000000003, 200.0, 0, 166.66666666666669, 58.333333333333336, 33.333333333333336, 108.33333333333334, 58.333333333333336, 83.33333333333334]

[37.777777777777786, 98.33333333333334, 306.66666666666674, 0, 161.11111111111111, 67.22222222222223, 28.88888888888889, 147.22222222222223, 67.22222222222223, 85.55555555555556]

[38.96296296296297, 96.11111111111111, 267.1111111111111, 0, 191.62962962962962, 90.92592592592594, 29.48148148148148, 91.37037037037038, 90.92592592592594, 103.4814814814815]

[45.44197530864198, 109.14814814814815, 268.2962962962963, 0, 166.5530864197531, 89.3358024691358, 34.26172839506173, 95.49876543209876, 89.3358024691358, 102.1283950617284]

[46.29267489711934, 110.87061728395062, 255.10123456790126, 0, 170.65119341563786, 89.14880658436215, 33.90090534979424, 101.6485596707819, 89.14880658436215, 103.23720164609054]

[46.14658984910837, 110.84480658436216, 261.5286255144033, 0, 170.8417009602195, 86.45992866941016, 34.19658710562415, 102.80336899862827, 86.45992866941016, 100.71846364883403]

[45.50840420667581, 109.46036982167352, 262.1326310013718, 0, 171.9670708733425, 87.02837274805671, 33.52492363968908, 102.09216424325562, 87.02837274805671, 101.25769071787838]

[45.48087903673222, 109.45679261088252, 262.19467032464564, 0, 171.64043656759645, 87.3007589330895, 33.668717524767565, 101.55454629477214, 87.3007589330895, 101.40243977442466]

[45.59186038876189, 109.6349052449322, 261.716884816339, 0, 171.50923700330236, 87.385803113753, 33.70731727317991, 101.58621012487936, 87.385803113753, 101.48197892109944]

[45.624832103182115, 109.71275167793529, 261.6958357505665, 0, 171.48004540944643, 87.3129244602686, 33.728527712293186, 101.70086188801845, 87.3129244602686, 101.43129653802099]

[45.61105391268315, 109.68613352272453, 261.74948175393865, 0, 171.52043145937424, 87.28928033951827, 33.71501241013893, 101.72507950194947, 87.28928033951827, 101.41424676015468]

[45.601144733241924, 109.66716360631676, 261.780830377369, 0, 171.52994137951578, 87.29370444132594, 33.710465802707915, 101.70712999433437, 87.29370444132594, 101.41591522386257]

[45.6011120650757, 109.66632886274434, 261.7736705438731, 0, 171.52628196027473, 87.30115392649405, 33.71091072636335, 101.69800511553845, 87.30115392649405, 101.42138287314245]

[45.60321724076198, 109.67041403417554, 261.7667591595329, 0, 171.52277019454004, 87.30170848917304, 33.71236876617132, 101.6991227242842, 87.30170848917304, 101.42193090218818]

[45.603753934758494, 109.67151964669037, 261.7660308590159, 0, 171.52279094800187, 87.30047409568607, 33.71251490725018, 101.70132396907279, 87.30047409568607, 101.42111754383846]

[45.60346373411634, 109.671007613251, 261.76737020470676, 0, 171.52345520000765, 87.29999926398614, 33.71229801169026, 101.7016566573181, 87.29999926398614, 101.42075005093784]

#

nodes_sorted_by_points [2 4 1 7 9 5 8 0 6 3]

We apply the PageRank algorithm on our web graph with 10 nodes having 100 points each initially. After sufficient number of iterations, we observe that the values of points are changing by a very small amount, so we can consider that convergence is achieved and we can stop the iteration process. As soon as we stop iterating, the algorithm outputs the ranking of the web pages, i.e., the order in which they will be displayed.

8.6 Results

The size of the node is proportional to the points accumulated by it at a particular time (in a particular iteration). Initially the size of all nodes is same as we have allocated 100 points to each of the nodes.

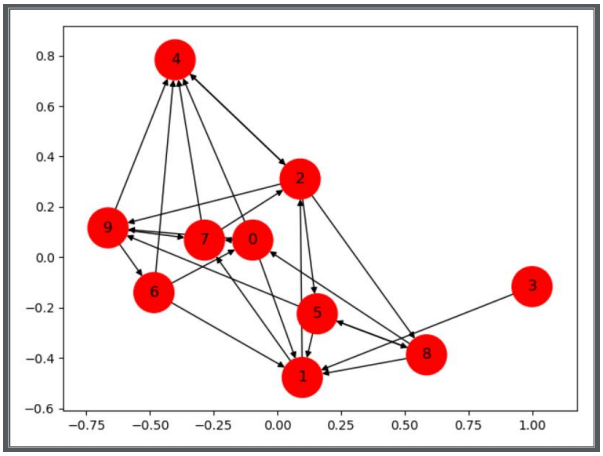


Fig 8.1 Initial Web Graph

After convergence is achieved, we can see that the size of the nodes is not the same. The size of nodes is in accordance with the order in which those web pages will be displayed in the search results list.

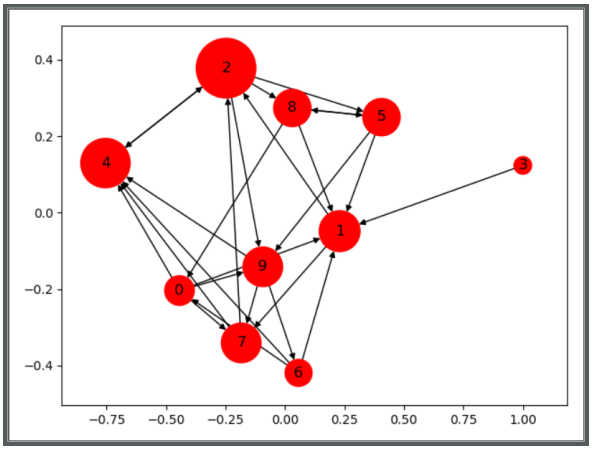


Fig 8.2 Web Graph after PageRank Implementation

E.g. – 2 has the maximum size, so it will be displayed as the 1st search result, then 4, then 1, and so on.

9. Conclusion

In our study project, we have studied graph clustering techniques and its role in community detection. Graphs are the most widely used data abstraction when it comes to store huge chunks of data. And it's very important to derive patterns and analyse data so clustering techniques become so helpful as classifying data into various clusters on the basis of some similar property. In this study project, we aim to study data clustering problem in detail and its variations. Basic linear algebra concepts are mostly used for matrix manipulations. We studied the Markov Clustering algorithm, including the concepts of expansion and inflation and using them iteratively for grouping the clusters. We then saw the time complexity of this algorithm and that how it can be improved, and concluded that a clustering algorithm with a relatively better time complexity could have been used.

This led us towards the study of Girvan-Newman algorithm which is widely used for community detection. Here we learned the concepts of vertex and edge betweenness, and extracting the clusters with the successful removal of links.

For weighted graph clustering, we considered the attractiveness values as a parameter for finding the clusters in the Attractiveness Based Community Detection (ABCD) algorithm.

Social network analysis is very new and exciting field and we can use clustering techniques in it to extract a lot of interesting information from huge data. So we implemented community detection on two graphs as part of implementation of clustering. First, we took a barbell graph and clustered it. Then we implemented clustering using Girvan-Newman algorithm on the Zachary-Karate club graph which is quite famous. Corresponding plots are shown in report which are self-explanatory. Community detection is not at all different from simple clustering problem - it's just real life layman name of standard clustering problem. Another important part of the project was social network analysis - actual practical implementation with the help of models. It gives us brief idea which algorithm to prefer in which situation.

Social network comes in different forms and structures. Mapping social media networks can enable a better understanding of the variety of ways individuals form groups and organize online. Our social network maps have illustrated three different structures of connection around different kinds of topics.

A working definition of network science is the study of network representations of physical, biological, and social phenomenon leading to predictive models of these phenomenon. Initiation of a field of network science would be appropriate to provide a body of rigorous results that would improve the predictability of the engineering design of complex networks and also speed up basic research in a variety of application areas.

After analysis of all the three objectives under study of social networks, it can be concluded that social network analysis plays a major role in change detection and plotting curves that give us relation between various causes and effects. It also throws light on the connectedness of the entire world and how big companies like Google, rank their search results based on various factors.

But one must note that this field of graph clustering is relatively very new and a lot of new approaches are still being introduced. Concluding, graph theory is very useful in a lot of different fields, and ‘Graph Clustering’ is one of the most widely studied problem in the field of graph theory and linear algebra.

10. References

1. M.E.J. Newman. *Fast algorithm for detecting community structure in networks*. Physical Review E, 2004.69(6), P.066133.

2. B.Cai, H.Wang, H.Zheng, H.Wang. "*Evaluation repeated random walks in community detection of social networks*," Proceedings of the Ninth International Conference on Machine Learning and Cybernetics, Qingdao, 11-14 July 2010.

3. P. Pons and M. Latapy. *Computing communities in large networks using random walks. J. Graph Algorithms* Appl., vol. 10, no. 2, pp. 191C218, 2006.

4. Jie Chen and Yousef Saad. *Dense subgraph extraction with application to community detection*. IEEE Transactions on Knowledge and Data Engineering, vol. 24, NO. 7, JULY 2012. pp1216-1230.

5. Dode, Albi & Hasani, Silvester. (2017). *PageRank Algorithm*. 10.9790/0661-1901030107. 2278-661. 10.9790/0661-1901030107.

6. Opsahl, Tore & Vernet, Antoine & Alnuaimi, Tufool & George, Gerard. (2017). *Revisiting the Small-World Phenomenon: Efficiency Variation and Classification of Small-World Networks. Organizational Research Methods.* 20. in press. 10.1177/1094428116675032.

7. *Graph Clustering - Survey, Satu Elisa Schaeffer,2007*

8. *Numerical Linear Algebra and Applications, 2nd edition, Biswa Nath Datta*.