Detecting Tight Communities and Friend Recommendation on Facebook

Gayathri Balakumar, Prathyusha Kanmanth Reddy, Qianying Lin, Vidya Gopalan
The University of Texas at Dallas

1. Introduction

The incessant growth and usage of social media sites makes it evident that they improve connectedness with people across the globe. But this ceaseless growth also makes one's network and information generated in it, to be unmanageable. The notion of forming social circles within it helps in better organization by viewing friends as smaller groups of communities. A community in a social network refers to a group of people who are more tightly interconnected than the overall network. Users in a community tend to interact more frequently with each other and share common interests. Detecting such tight communities is one of the essential tools in social network analysis. Manual addition of circles is laborious and also requires constant updates as and when new connections are made. In this project, we are attempting to analyze the social media data of various users and detect their closely-knit groups and also make friend recommendations based on the communities detected.

1.1 Problem Statement

This project deals with ego network of facebook users and aims at addressing the below two problems:

- Automatically detect the user's tight communities:- given a single user and his social network, the goal is to identify his close circles, which are smaller and closer subset of his friend list.
- Building a friend recommendation system:- suggest users with choices to make friends with, based on the mutual friends from the circles identified as part of the first problem.

2. Related Work

In SNAP (Stanford Network Analysis Project), concept of viewing users as individual "Egos" was introduced. They formulated the problem of circle detection as a clustering problem on her ego-network, the network of friendships between her friends. Current project is constructed based on personal friendship network which draws ideas from the ego-network.

Ego Network

Ego network is defined as a network of friendships between a user and his friends. The user is referred as the 'ego' and the nodes in his network (or his friends) are 'alters'. Circles are formed by densely-connected sets of alters. Each circle is not only densely connected but its members also share common features. The following example shows the ego network of the ego 'u' and his alters 'v_i' [1].

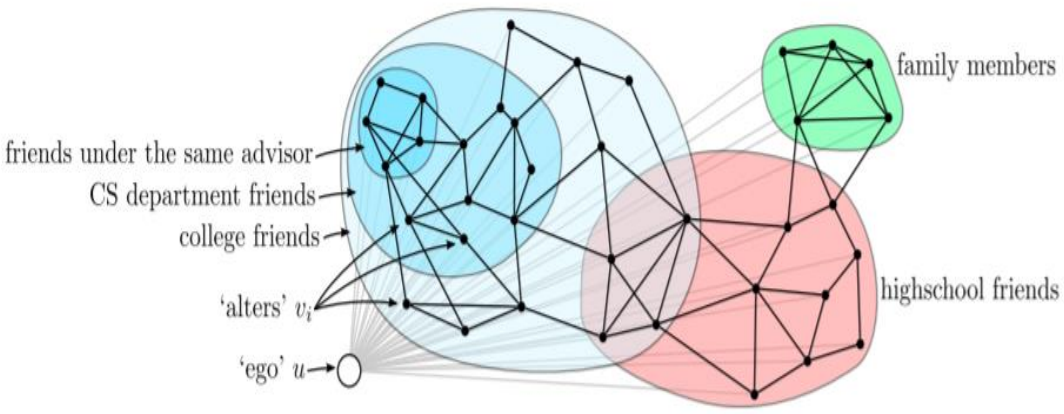

Fig 1. Ego network

3. Dataset Description

For the project work, facebook dataset provided by the Stanford University is selected. It consists data of multiple users identified with a node_id. Each of these users is called an 'ego' and each ego consists of 5 files:

- *Node_id.circles* which represents the ego's set of circles as circle name and a series of node ids. Each record in file denotes a circle.
- *Node_id.edges* gives the connection between the alters of the ego node. The ego node is assumed to follow every nodeId present in the file.
- *Node_id.featname* provides the names of each feature of the ego node.
- *Node_id.egofeat* contains a series of 0's and 1's that represents if the ego user had specified his features on facebook.
- *Node_id.feat* contains 0, 1 series of feature details for every alter of the ego user.
- *facebook_combined* file which has node ID pairs denoting connections of all the ego users who are connected on facebook instead of focusing on one ego. It's basically a list of all the edges on the network.

Statistics provided for the dataset [2]:

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

Fig 2. Statistics of the dataset provided by SNAP

4. Dataset Analysis & Pre-processing Techniques

This work uses the combined edges file and a undirected graph is built for the overall network involving all the ego users using the *igraph* package. Further analysis on the dataset is given below:

4.1 Dataset Analysis

- **Network Connectivity**
  The built network is checked for proper connectivity using *is_connected* function.
  g.is_connected(mode=STRONG) = True

- **Network Diameter**
  The diameter of a graph is the length of the longest geodesic in its network.
  Farthest vertices of the network is identified and the number of hops needed to reach from one of them to the other defines the overall network diameter.
  g.farthest_points(directed=False, unconn=True, weights=None) = (687, 3981, 8)

Path between two of them say 687 and 3918 is given by get_diameter function.
g.get_diameter(directed=False, unconn=True, weights=None)
= [687, 686, 698, 3437, 567, 414, 594, 3980, 3981]

g.diameter(directed=False, unconn=True, weights=None) = 8

- **Network Betweenness**
  Betweenness defines the number of geodesic for a vertex. Overall network betweenness is computed by calculating the mean of the betweenness of all the vertices in the graph.

  nwBetweeness = g.betweenness(vertices=None, directed=False, cutoff=None, weights=None, nobigint=True)
  meanNwBetweeness= reduce(lambda x, y: x + y, nwBetweeness) / len(nwBetweeness) = 5436.171

- **Network Degree**
  Degree is the most fundamental property of a vertex which gives the number of edges connected to it. Mean degree of all the vertices is calculated to understand the overall connectedness of the network and the distribution of degrees is plotted**.**

  nwDegrees = g.degree()
  meanNwDegree= reduce(lambda x, y: x + y, nwDegrees) / len(nwDegrees) = 43

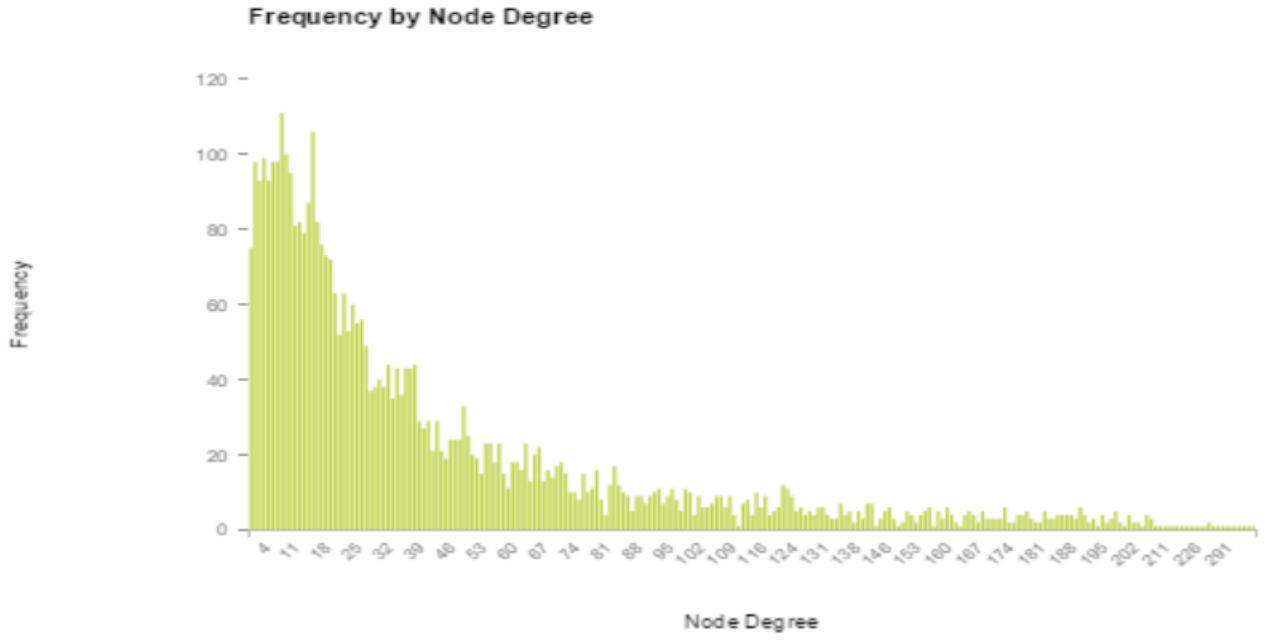

Fig 3. Node Degree vs Frequency graph

4.2 Preprocessing Techniques

The network is looped through every node checking for its number of neighbours. Based on neighbour count significance of a node in the network is found. Threshold values were fixed as more than 300 neighbours for most significant nodes and less than two neighbours for insignificant nodes.

4.2.1 Island Removal

The community detection involves nodes with overlapping network. Those nodes which have zero or one edge and do not contribute much to the strength of the network. Such nodes with less than two neighbors are identified to be insignificant and are named islands. It is a good practice to eliminate all such island nodes before proceeding to apply community detection algorithms on the graph as they may remain as inappropriate information for the algorithms.

1. Identify the islands on the network

```
for v in vertices:
    friends_list = g.neighbors(vertex=v, mode=ALL)
    if (len(friends_list) < 2):
        island_list.append(v)
```

Island nodes set on the network = set([43, 3974, 2569, 11, 12, 3853, 15, 3856, 3729, 18, 2195, 918, 1560, 2457, 2842, 3183, 668, 3230, 287, 801, 674, 2596, 37, 550, 4008, 4010, 1834, 1581, 3935, 4015, 692, 1206, 4024, 1466, 607, 447, 3650, 4035, 4022, 292, 1096, 585, 74, 3451, 335, 209, 210, 3798, 215, 3748, 602, 911, 2269, 1119, 608, 3984, 891, 613, 358, 2079, 1386, 875, 3820, 3453, 624, 3031, 3570, 883, 1145, 114, 3071, 892, 3709, 638, 2470])

All the islands are double checked with their degrees to be 1.

```
island_degree_list=[]
for i in island_list:
  island_degree_list.append(g.degree(i))

print  set(island_degree_list) = set([1])
```

2. Delete all the vertices matching the above user IDs from the original graph.

```
g.delete_vertices(island_list)
```

Vertices counts are verified after the deletion of nodes to ensure correctness of the resultant graph.

```
print len(set(vertices))    = 4039
print len(set(island_list))    =75
print len(set(newVertices)) =3964
```

4.2.2 Building Core Node Subgraph

Communities are to be found for users identified to have considerable size of network. Significant nodes identified to have more than 300 neighbors are named as core nodes and the community detection algorithms are applied to one of them.

Core nodes set on the network = set([0, 1912, 107, 1684, 3437])

Core nodes identified are verified to be the ego nodes from the dataset. Average degree of core nodes is computed

```
mean_core_degree = reduce(lambda x, y: x + y, core_degree_list) / len(core_degree_list)
mean_core_degree = 687
```

Node 0 is selected as user for whom the communities have to be detected and a subgraph with all friends of 0 node is built.

```
node0_graph = g.subgraph(node0_friends_list, implementation = "auto")
```

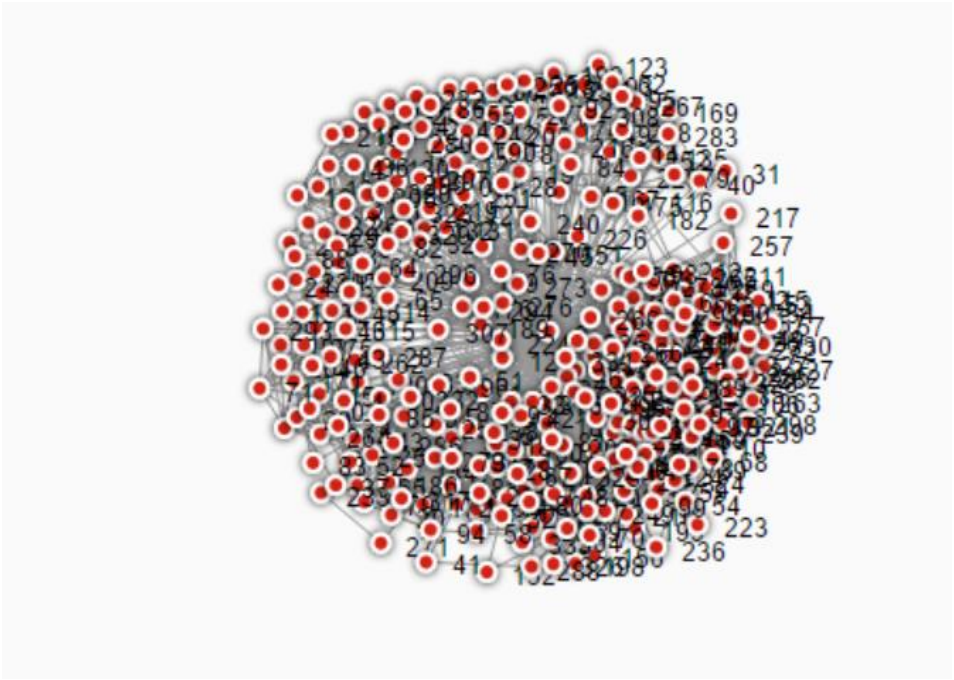

Fig 4. Sub-graph of node 0

5. Proposed Solution and Experimental Results

5.1 Problem 1 - Community Detection

If a graph is formally defined as an ordered set of vertices and edges, i.e., G=(V,E), a community on a graph can be defined as its subset such that the vertices are found to cluster into tightly-knit groups and the connectedness of the edges are found to be more within the subgraph.

One way of finding close communities on facebook can be achieved by finding out all the possible cliques on the graph. A clique is a maximal complete subgraph i.e. all the nodes in the subgraph are connected to each other and are not subsumed by any other complete subgraph. A clique imposes strict definition for a community where every member in the sub group has to be a friend with every other member. Maximal_Cliques function of igraph library is used to find the cliques.

cliques_0 = node0_graph.maximal_cliques(min =2 , max =10)

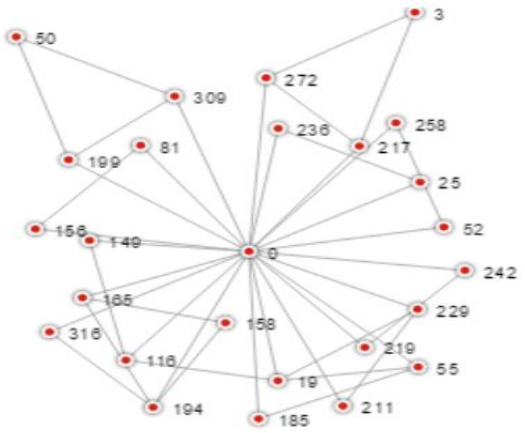

Fig 5. Sample cliques detected using the above function.

This project uses the below algorithms from igraph package to identify communities

- Girvan-newman - edge betweenness algorithm from igraph package
- Clauset-Newman-Moore - fast greedy algorithm from igraph package
- Walktrap algorithm from igraph package
- Infomap from igraph package

5.1.1 The Girvan–Newman algorithm

The Girvan–Newman algorithm[9] detects communities by sequentially 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 important to communities, the Girvan–Newman algorithm focuses on edges that are most likely between communities.

Edge betweenness of an edge is used to measure the number of shortest paths through the edge. If there is more than one shortest path between a pair of nodes, each path is assigned equal weight and the total weights add up to one.

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.

Cmd 10

```
#community detection with centrality based approach using edge betweeness
communities = node0_graph.community_edge_betweenness(directed=False)
clusters = communities.as_clustering()
print clusters.modularity
print clusters

```

```
0.355975924968
Clustering with 334 elements and 27 clusters
[ 0] 0, 3, 5, 7, 9, 10, 11, 13, 17, 18, 21, 22, 23, 25, 26, 27, 33, 34, 35,
     39, 44, 48, 49, 50, 53, 54, 56, 57, 59, 60, 61, 63, 66, 68, 69, 70, 72,
     75, 77, 78, 80, 81, 89, 91, 96, 97, 98, 99, 102, 106, 109, 110, 111, 113,
     114, 115, 117, 120, 121, 124, 125, 126, 128, 133, 134, 138, 140, 145,
     148, 150, 153, 156, 157, 160, 161, 162, 164, 168, 170, 177, 178, 180,
     182, 191, 192, 195, 198, 199, 200, 201, 202, 203, 210, 211, 212, 213,
     218, 220, 221, 223, 224, 225, 227, 228, 235, 237, 239, 241, 246, 247,
     249, 250, 254, 257, 259, 260, 261, 263, 265, 266, 269, 274, 276, 278,
     279, 281, 282, 284, 285, 290, 291, 295, 298, 300, 301, 302, 304, 305,
     309, 310, 311, 312, 316, 318, 319, 321, 322, 324, 325, 326, 327, 328,
     330, 331, 333
[ 1] 1, 20, 42, 47, 51, 67, 73, 85, 87, 94, 118, 122, 155, 172, 179, 183, 186,
     188, 196, 231, 238, 243, 255, 286, 287, 289, 317, 332
```

Fig 6. Sample Code and output using edge-betweenness

5.1.2 The Clauset-Newman-Moore algorithm

It is a fast, yet greedy algorithm[8] that tries to find strong communities by optimizing modularity. Initially, every vertex belongs to a distinct community. Then, the communities are merged iteratively such that each merge is locally optimal.

```
#community detection using fast greedy algorithm
fastGreedy = node0_graph.community_fastgreedy()
FGcluster = fastGreedy.as_clustering()
print FGcluster.modularity
print FGcluster
```

```
0.410649056099
Clustering with 334 elements and 8 clusters
[0] 0, 1, 5, 20, 23, 26, 29, 30, 37, 41, 42, 43, 46, 47, 48, 51, 52, 58, 64,
    65, 67, 73, 74, 81, 83, 85, 87, 89, 90, 93, 94, 99, 100, 111, 117, 118,
    122, 137, 138, 142, 145, 152, 155, 156, 158, 163, 165, 170, 171, 172, 174,
    175, 179, 181, 182, 183, 184, 186, 188, 190, 194, 196, 197, 198, 200, 203,
    206, 217, 218, 219, 222, 223, 224, 230, 231, 233, 238, 242, 243, 244, 245,
    249, 255, 258, 271, 272, 275, 286, 287, 288, 289, 294, 303, 304, 307, 311,
    316, 317, 322, 326, 332
```

Fig 7. Sample code and output using fast greedy algorithm

5.1.3 Walktrap algorithm

This algorithm[11] tries to find densely connected communities in a graph using random walks. A walk is any route through a graph from vertex to vertex along the edges. The idea is that if we perform random walks on the graph, then the walks are more likely to stay within the same community because there are only a few edges that lead outside a given community. It is slower than fast greedy approach, but is said to provide more accurate results.

```
#community detection using walk trap algorithm
walkTrap = node0_graph.community_walktrap()
WTcluster = walkTrap.as_clustering()
print WTcluster.modularity
print WTcluster
```

```
0.351626414573
Clustering with 334 elements and 40 clusters
[ 0] 0, 5, 17, 23, 26, 33, 39, 48, 70, 109, 111, 114, 117, 124, 128, 138, 145,
     148, 150, 170, 175, 199, 203, 210, 218, 223, 224, 261, 281, 290, 295,
     302, 303, 304, 309, 311, 316, 331
[ 1] 1, 42, 67, 81, 118, 122, 156, 183, 322
[ 2] 2, 12, 16, 24, 107, 108, 132, 136, 141, 154, 215, 292, 299, 313, 320, 329
[ 3] 3, 9, 10, 11, 13, 21, 22, 25, 27, 34, 35, 44, 49, 50, 53, 54, 56, 57, 59,
     60, 61, 63, 66, 68, 72, 75, 77, 78, 89, 91, 96, 97, 98, 99, 102, 106,
     110, 113, 115, 120, 125, 126, 133, 134, 140, 153, 157, 161, 162, 164,
     168, 177, 178, 180, 191, 192, 195, 200, 201, 202, 211, 212, 213, 220,
     221, 225, 227, 228, 237, 239, 241, 246, 247, 250, 254, 257, 260, 263,
     265, 266, 269, 274, 276, 278, 279, 282, 284, 285, 291, 298, 300, 301,
     305, 310, 312, 318, 319, 321, 324, 327, 328, 330
[ 4] 4, 71, 144, 173, 187, 207, 262, 264, 293, 315
```

Fig 8. Sample code and output using WalkTrap algorithm

5.1.4 Infomap

This algorithm finds the community structure of the network according to the Infomap method of Martin Rosvall and Carl T. Bergstrom. It supports both directed and undirected graphs. The community structure is represented through a two-level nomenclature based on Huffman coding: one to distinguish communities in the network and the other to distinguish nodes in a community. The problem of finding the best partition is expressed as minimizing the quantity of information needed to represent some random walk in the network using this nomenclature. With a partition containing few inter-community links, the walker will probably stay longer inside communities, therefore only the second level will be needed to describe its path, leading to a compact representation[7].

```
#community detection using info map algorithm
infoMap = node0_graph.community_infomap()
print infoMap.modularity
print infoMap.as_cover()
```

```
0.391379834333
Cover with 24 clusters
[ 0] 3, 7, 9, 10, 11, 13, 17, 18, 21, 22, 23, 25, 26, 27, 33, 34, 35, 39, 44,
     49, 50, 53, 54, 56, 57, 59, 60, 61, 63, 66, 68, 70, 72, 75, 78, 80, 89,
     91, 96, 97, 98, 99, 102, 106, 109, 110, 111, 113, 114, 115, 117, 120,
     121, 124, 125, 126, 128, 133, 134, 138, 140, 148, 150, 153, 157, 160,
     161, 162, 164, 168, 170, 177, 178, 180, 191, 192, 195, 199, 200, 201,
     202, 203, 210, 211, 212, 213, 218, 220, 221, 225, 227, 228, 235, 237,
     239, 241, 246, 247, 250, 254, 257, 260, 261, 263, 265, 266, 269, 274,
     276, 278, 279, 281, 282, 284, 285, 290, 291, 295, 298, 300, 301, 302,
     304, 305, 309, 310, 311, 312, 316, 318, 319, 321, 324, 325, 326, 327,
     328, 330, 331, 333
[ 1] 2, 12, 14, 16, 24, 28, 36, 38, 86, 104, 107, 108, 129, 132, 136, 141,
     143, 154, 159, 204, 205, 209, 215, 232, 251, 277, 297, 299, 313, 320,
     323, 329
[ 2] 1, 20, 42, 47, 48, 51, 67, 73, 81, 85, 87, 94, 118, 122, 172, 179, 183,
     186, 188, 196, 231, 238, 243, 255, 286, 287, 289, 317, 332
[ 3] 0, 5, 145, 156, 175, 217, 223, 224, 268, 272, 292, 303, 322
[ 4] 31, 40, 62, 79, 92, 95, 123, 135, 167, 169, 214, 216, 252, 267, 283, 308
[ 5] 32, 101, 119, 127, 131, 151, 176, 189, 240, 270, 273, 296, 307
[ 6] 4, 71, 144, 173, 187, 207, 262, 264, 293, 315
```

Fig 9. Sample code and output using Infomap algorithm

5.1.5 Evaluation Metrics

There are two types of measures defined for clustering problems: internal and external criteria[6].

- Internal criterion: Assesses the quality of the detected community structure. This means you don't have any reference structure to which you could compare the estimated structure. Example: Modularity
- External criterion: Compare the estimated community structure to a reference community structure (aka. ground truth, gold standard, etc.). Example: NMI, (A)RI, purity

The dataset provided by Stanford Large Network Dataset Collection does not give us any actual community structures to refer to or compare our community structure to. So we cannot perform any external measures at all. We used Modularity to measure the strength of the community and it is by far the best measure to evaluate communities. Modularity quantifies the extent to which the given graph partitions into communities. It presents a systematic tendency to have more intra community links. So communities with high modularity are known to have dense connections between the nodes within modules but sparse connections between nodes in different modules.

Modularity values for communities detected using different algorithms:

Algorithm	Modularity
Edge betweenness	0.355
Fast Greedy	0.410
Info map	0.385
Walk trap	0.351

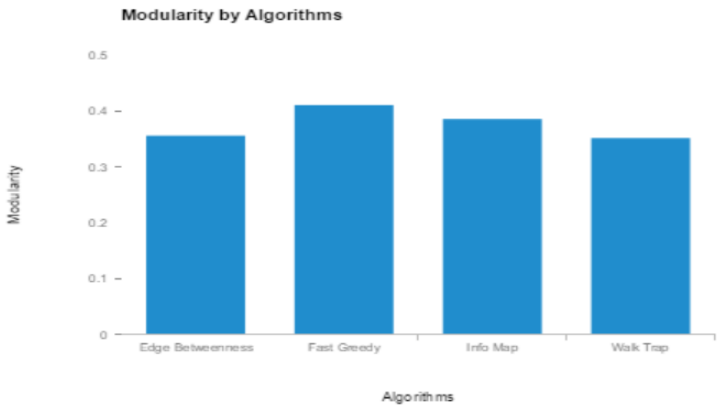

Fig 10. Modularity values for various algorithms

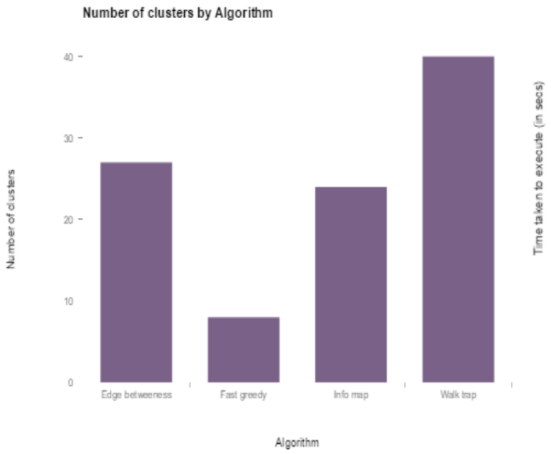

Fig 11. Number of clusters in each algorithm

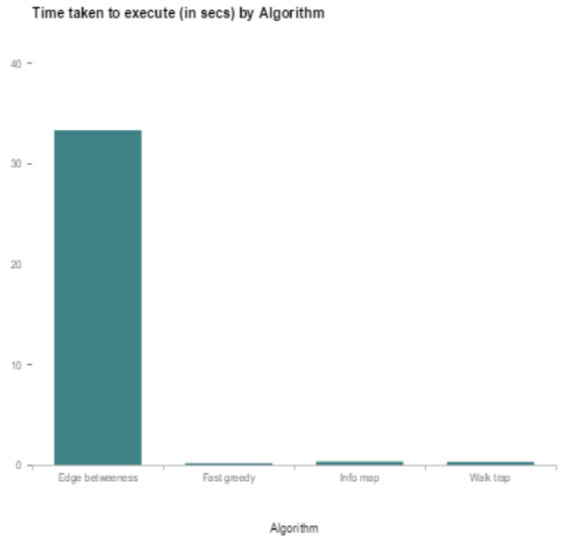

Fig 12. Time taken per algorithm for execution

Based on modularity, number of tighter clusters and time taken, fast greedy algorithm showed having upper hand over the other algorithms. So, the output communities from fast greedy is opted as input for next problem recommendation system.

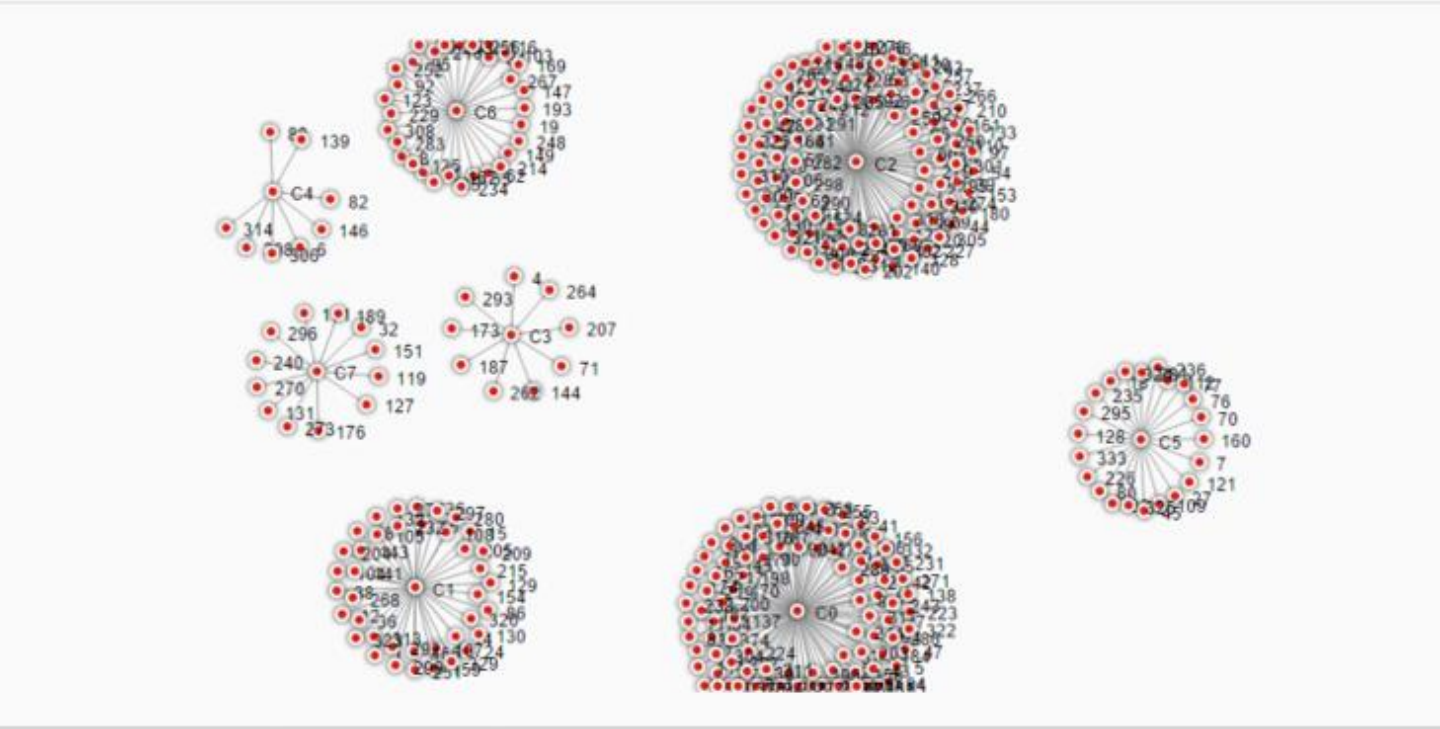

Fig 13. Communities identified by fast greedy algorithm

5.2 Problem 2 - Recommendation System

The project also focuses on building a simple recommendation system on the facebook dataset. A recommendation system is a special kind of information filtering that focuses on returning only those which the user is likely to be interested in. In this project, we aim at recommending based on number of mutual friends from people who are identified to be in common communities. Friend recommendation in this dataset is similar to the co-occurrence problem we learnt in class. We used the "Pairs" method and the pseudo-code is as following in Fig 14.

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             for all term u ∈ Neighbors(w) do
5:                 Emit(pair (w, u), count 1)        ▷ Emit count for each co-occurrence

1: class Reducer
2:     method Reduce(pair p, counts [c1, c2, ...])
3:         s ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             s ← s + c                              ▷ Sum co-occurrence counts
6:         Emit(pair p, count s)
```

Fig 14. Pairs Algorithm pseudo-code

The advantage of "Pairs" method is that it is easy to understand and implemented. The disadvantage is that there are a lot of pairs to sort and shuffle around (that is why it took half an hour to run). The reason why we

didn't use the "Stripes" is that there are only a few friends in common between any two persons. If we build a stripe of friends of all persons, the stripe object will be too large to process in memory as it is a heavy weight object.

5.2.1 Implementation of the Algorithm:

We implemented the following algorithm to build the recommender system. It's done using Map Reduce with Pairs approach[11]. In our dataset, the first column is considered as "fromUser" and the second column is considered as "toUser". The "toUser" is a friend of "fromUser".

- **Map phase**
  1. Emit <fromUser, toUser, -1> for all toUser. Let's say there are *n* 'toUser', then we'll emit *n* records for describing that 'fromUser' and 'toUser' are friends. They are already friends, so we emit a '-1' to indicate it.
  2. Emit <toUser1, toUser2, 1> for all the possible combination of 'toUser1' and 'toUser2' from 'toUser', and they have mutual friend, 'fromUser'. We will emit 1 along with it to indicate that this pair is a possible recommendation we can suggest.
- **Reduce phase**
  1. Just by summing how many mutual friends they have between them, we will obtain the total count of mutual friends between all the pairs.
  2. Sort the result based on the number of mutual friends.
- **Recommendation phase**
  1. Now, choose any user you want to recommend friends to. Say, user with ID= X. We scan through the pairs generated in reduce method and fetch list of all possible users that we can recommend.
  2. Scan the facebook_combined file to get the friends list of user X.
  3. Compare these two lists and fetch user IDs which are not in the friends list already. Every node from obtained list of suggestions is checked to be present in a common community with the selected user X.
  4. Those nodes that satisfies all the above points are recommended as a friends to the user X.

In short, we emit ((fromUser, toUser), count 1) in the Mapper phase and sum the counts in the Reducer phase. Then we remove the 'toUser' who are already friends of a given 'fromUser' and recommend friends based on the counts of mutual friends and common communities.

```
def predict(entry):
  return (entry[0],entry[1]),entry[2]

mutualFriendsRDD =sc.parallelize(mutualFriends)
prediction=mutualFriendsRDD.map(predict)

PredictionRDD = prediction.reduceByKey(lambda a, b: a + b)
sortedRdd=PredictionRDD.sortBy(lambda a: -a[1])
print sortedRdd.collect()
```

```
[((2624, 2625), 163), ((2602, 2604), 163), ((2607, 2611), 161), ((2601, 2602), 161), ((2586, 2590), 155), ((1800, 1804), 153), ((2600, 2601), 149), ((2625,
2630), 147), ((2593, 2600), 142), ((2615, 2619), 137), ((2611, 2615), 136), ((2654, 2655), 134), ((2559, 2560), 132), ((2560, 2561), 122), ((2542, 2543), 1
22), ((1799, 1800), 121), ((1827, 1833), 120), ((2507, 2520), 114), ((2623, 2624), 114), ((2409, 2410), 112), ((2564, 2573), 112), ((2410, 2414), 112), ((1
833, 1835), 111), ((2549, 2550), 110), ((2638, 2646), 105), ((2539, 2542), 105), ((2604, 2607), 104), ((2550, 2551), 102), ((2500, 2504), 101), ((2630, 263
1), 99), ((1584, 1589), 99), ((1886, 1888), 98), ((2646, 2654), 98), ((1746, 1750), 97), ((2492, 2495), 96), ((2631, 2638), 96), ((2546, 2549), 96), ((252
0, 2521), 94), ((2579, 2586), 94), ((2619, 2623), 93), ((2510, 2511), 93), ((2423, 2428), 92), ((2556, 2559), 92), ((2578, 2579), 92), ((2464, 2467), 91),
```

Fig 15. List of all pairs and count of their mutual friends

```
#Select one user for whom friend suggestion has to be made
fromuser=115
#Filter mutual friend list for the selected user
suggestions_115_1 = sortedRdd.filter(lambda x:x[0][0]==fromuser).map(lambda x:(x[0][1],x[1]))
suggestions_115_2 = sortedRdd.filter(lambda x:x[0][1]==fromuser).map(lambda x:(x[0][0],x[1]))
suggestions_115 = suggestions_115_1.union(suggestions_115_2)
suggestions_115_sorted = suggestions_115.sortBy(lambda x:-x[1])
suggestions_115_RDD = suggestions_115_sorted.map(lambda x:x[0])
print suggestions_115_RDD.collect()
```

```
[116, 111, 41, 138, 149, 112, 114, 20]
```

Fig 16. List of suggestions made initially

```
#Get all friends of user 115
friends_115_1= egoRDD.filter(lambda x:x[0]==fromuser).map(lambda x:x[1])
friends_115_2= egoRDD.filter(lambda x:x[1]==fromuser).map(lambda x:x[0])
friends_115 = friends_115_1.union(friends_115_2)
print friends_115.collect()
```

```
[116, 137, 140, 144, 149, 192, 214, 220, 226, 262, 312, 326, 343, 0, 2, 14, 17, 19, 20, 28, 41]
```

Fig 17. List of existing friends

```
#Get all non friends of user 115
already_friends = suggestions_115_RDD.intersection(friends_115)
suggestions = suggestions_115_RDD.subtract(already_friends)
print suggestions.collect()
```

```
[111, 112, 114, 138]
```

Fig 18. Final list of recommendations that can be made

```
#Narrow down suggestion based on communities
#Communities detected by fastgreedy is opted because of better modularity
suggestion_list = suggestions.collect()
community_based_suggestion=[]
for cluster_index in range(8):
  for member in suggestion_list:
    if member in FGcluster[cluster_index] and 115 in FGcluster[cluster_index]:
      community_based_suggestion.append(member)

print community_based_suggestion
```

```
[114]
```

Fig 19. Recommending friends after checking if they co-exist in a community

7. Coding language used

The entire project is built in Pyspark using various algorithms in the igraph package. All this is done on Databricks. Databricks is a managed web-based platform for running Apache Spark, that provides automated cluster management. We can include various libraries on the clusters and 'python-igraph' (https://pypi.python.org/pypi/python-igraph) is one such library which we used for running the community detection algorithms.

8. Challenges Faced

- Understanding of graph data structure and learning new ML algorithms to apply on it.
- Finding appropriate graph libraries to use the algorithms was time consuming as we had to do a lot of research on various techniques.
- No usual pre processing or evaluation strategies were applicable on the graph data, yet we came up with two strategies for pre-processing.
- Coming up with our own algorithm and not an existing ML library function for recommending friends based on communities.
- Find a way to perform data visualization of graph data. After trials on few softwares, Sap lumira was used.

9. Conclusion

In summary, we performed graph analysis and studied the various properties of the social network. After properly understanding the entire network, we performed community detection algorithms on the graph and studied the modularity in each case. Apart from this, the detected communities were used to recommend friends to various users by calculating the mutual friends between the alters. This is an extra feature which we decided to implement in order to put the detected communities to proper use. The current dataset does not have specific features listed for each user. If they were provided, we can use them to detect users with similar features and recommend friends.

10. Contribution of Team members

We built the entire project by splitting into 2 parts-community detection and recommender system. All of us performed graph analysis and built the graph and then we worked on each problem by pair programming.

Graph analysis and community detection - Gayathri Balakumar, Vidya Gopalan
Graph analysis and recommender system - Prathyusha Kanmanth Reddy, Qianying Lin
Report Writing - Gayathri Balakumar, Vidya Gopalan, Prathyusha Kanmanth Reddy, Qianying Lin

11. References

[1] J. McAuley and J. Leskovec. Learning to Discover Social Circles in Ego Networks. NIPS, 2012.
[2] Stanford Large Network Dataset Collection: http://snap.stanford.edu/data/egonets-Facebook.html
[3] http://snap.stanford.edu/snappy/doc/reference/CommunityGirvanNewman.html
[4] http://snap.stanford.edu/snappy/doc/reference/CommunityCNM.html
[5] M. Rosvall and C. T. Bergstrom: Maps of information flow reveal community structure in complex networks, PNAS 105, 1118 (2008).
[6] https://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-clustering-1.html
[7] On Accuracy of Community Structure Discovery Algorithms: https://arxiv.org/ftp/arxiv/papers/1112/1112.4134.pdf

[8] Aaron Clauset, M. E. J. Newman, and Cristopher Moore, Finding community structure in very large networks
[9] Michelle Girvan, and M. E. J. Newman, Community structure in social and biological networks
[10] Pascal Pons and Matthieu Latapy, Computing communities in large networks using random walks
[11]An improved algorithm for mutual friends recommendation application of SNS in Hadoop: http://repo.lib.hosei.ac.jp/bitstream/10114/10983/1/12t2011.pdf