<u>Project Final Report</u>

<u>Detecting Tight Communities in Facebook</u>

Naveenraj Palanisamy (nxp154130)

Manoj Kumar Natha (mxn151930)

Anandan Sundar (axs156730)

Abhishek Thangudu (axt153530)

Venkata Sai Praneeth Palakurthi (vxp150830)

Table of Contents

1. Introduction

The identification of tight communities in a network is very important to understand the structure of the Network. This is helpful in many areas like Medical science, Defense teams. In Medical Science this analysis can be used for detecting Cancer causing genes, in Defense for terrorist community detection. We can find the like-mindedness among the people in Social Networks.

2. Project Description and Algorithm Approach

2.1. Dataset Description

This project deals with the finding of tightness among communities across Facebook users. A large dataset has been taken from SNAP Website which consists of details about Facebook users and the relationships between them. The original IDs are replaced with new values. All the feature names are also replaced by generic names. The dataset also contain groups to signify common interests among the Users. The Users in the network are considered as Vertices of the Graph and the relationship among the Users are considered as Edges of the Graph.

2.2. Dataset Statistics

Dataset Link: http://snap.stanford.edu/data/egonets-Facebook.html

The following table describes the SNAP Facebook dataset as in the Dataset description:

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

The Facebook dataset resembles an Undirected Graph. This is because two Users in Facebook are mutually connected if they are friends unlike in Twitter or Google+ where a User follows another User which signifies a directed graph. There is no difference between weakly and strongly connected components in an undirected graph. This is the reason the sub graph of largest WCC and SCC shows same number of nodes in the above table. The Average clustering coefficient of the whole signifies the overall tightness of the network. As all nodes in the SCC are equal to the total number of nodes in the graph, there is only one connected component in the given data. Since it is an undirected graph, SCC and WCC are same. So, detecting strongly connected components in the graph is same as detecting connected components in this graph. Also, since we have only one connected component, we are increasing our project's scope to finding cliques in the graph. A completely connected sub-graph in the given network is called a clique.

The Clique (Completely connected sub-component) is different from a strongly connected sub-component in the following way. In a completely connected sub-component, there is an edge between all the distinct nodes in the sub-component whereas in strongly connected sub-component, a path between any two distinct nodes in the sub-component exists.

2.3. Project Approach

All the analysis requires constructing and analyzing a network which can be modeled as a Graph. Apache Spark has provided an API for graphs and parallel computation in graphs which is GraphX. GraphX API is used in this project for constructing the graph and for analyzing the graph. In addition to this, JGrapht API is used. JGrapht has additional functionalities for graph processing. Here we use JGrapht for finding the cliques across the graph.

As a first step, a Graph is built using the Facebook dataset with Users as the vertices and the relationship among them as edges. The graph is built in such a way that if there is an edge between two vertices, it means the Users are friends in Facebook. All the edge weights are unique and taken as 1.

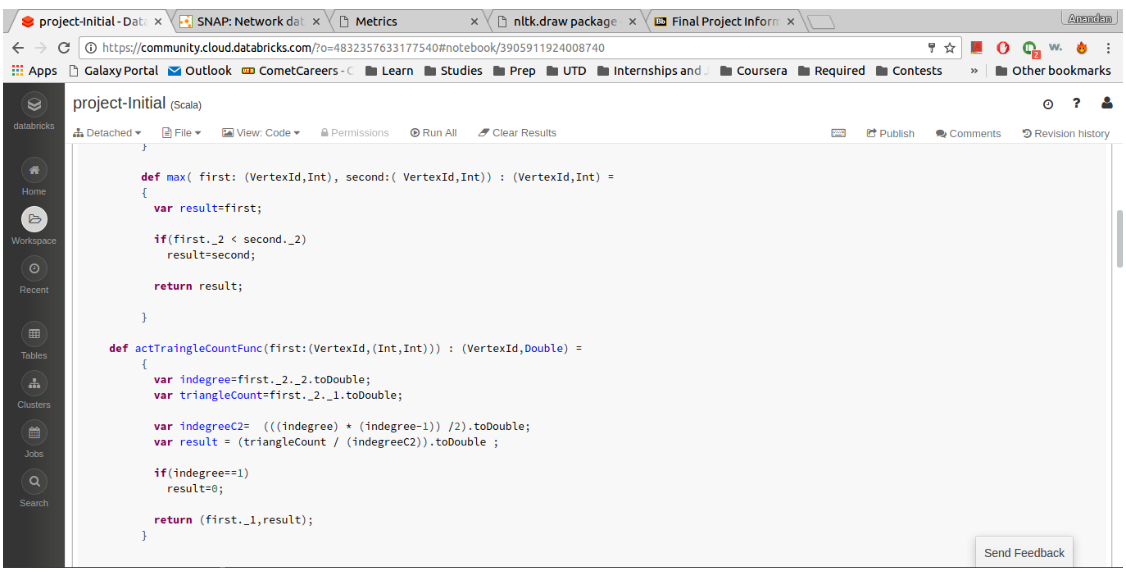
The second step is finding the strongly connected components and cliques (completely connected sub-graph) in the network-graph. To find this, we are using Bron-Kerbosch algorithm which is in Jgrapht. We get all the cliques in the graph listed. Now, we find the clustering coefficient of all vertices in the graph using the triangle count. Clustering coefficient of a vertex is measured based on number of triangle counts formed for each vertex to total number of triangles that can be formed at a vertex (${}^{n}C_{2}$ where n is number of neighbors for that Vertex). Then, tightness of each clique is determined using the clustering coefficient. Clustering coefficient of a clique is the average of the clustering coefficients of each vertex present in the clique. The value of clustering coefficient ranges in between 0 and 1. Based on these values we can determine how tight the given clique is.

The third step is determining the tightness for each clique in the overall network and ordering the cliques based on the tightness. These results are recorded and reported in next section.

3. Experimental Analysis and Results

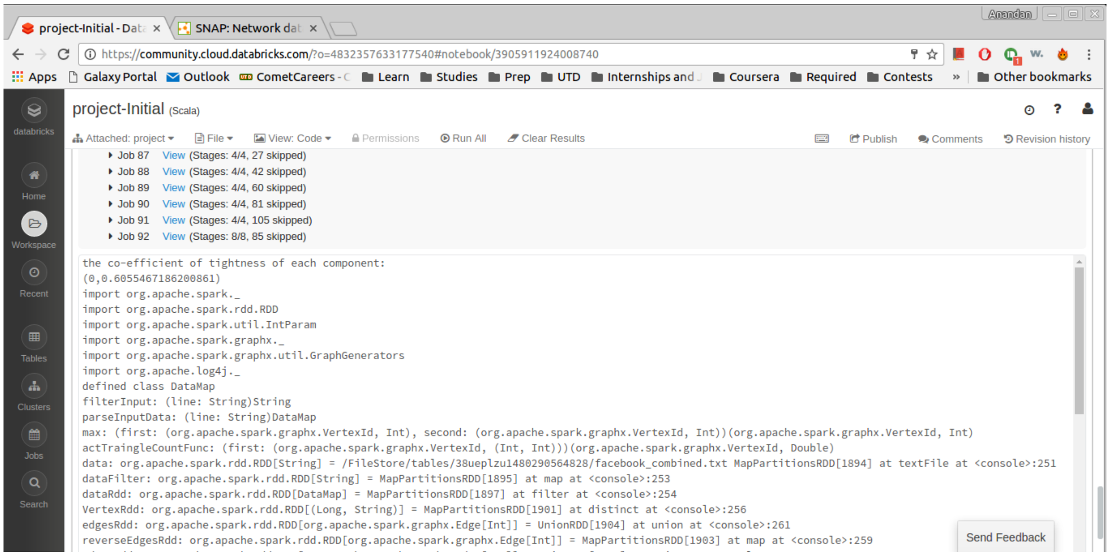
DFS (Depth First Search) traversal is used for finding the strongly connected components in a given graph network. GraphX have provided a direct method for finding the strongly connected components. In this dataset, entire graph is a strongly connected component.

Main logic of code:



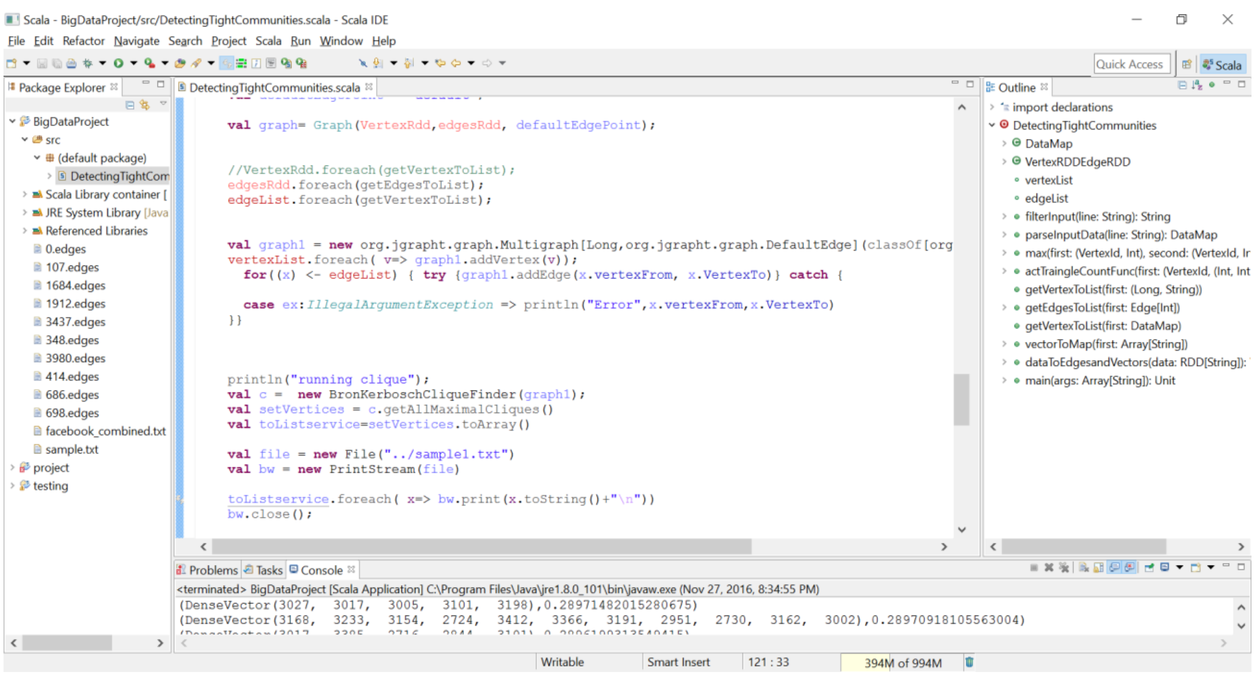

```
        def max( first: (VertexId,Int), second:( VertexId,Int)) : (VertexId,Int) =
        {
          var result=first;

          if(first._2 < second._2)
            result=second;

          return result;

        }

    def actTraingleCountFunc(first:(VertexId,(Int,Int))) : (VertexId,Double) =
        {
          var indegree=first._2._2.toDouble;
          var triangleCount=first._2._1.toDouble;

          var indegreeC2=  (((indegree) * (indegree-1)) /2).toDouble;
          var result = (triangleCount / (indegreeC2)).toDouble ;

          if(indegree==1)
            result=0;

          return (first._1,result);
        }
```

Output showing the average clustering coefficient of the whole graph:

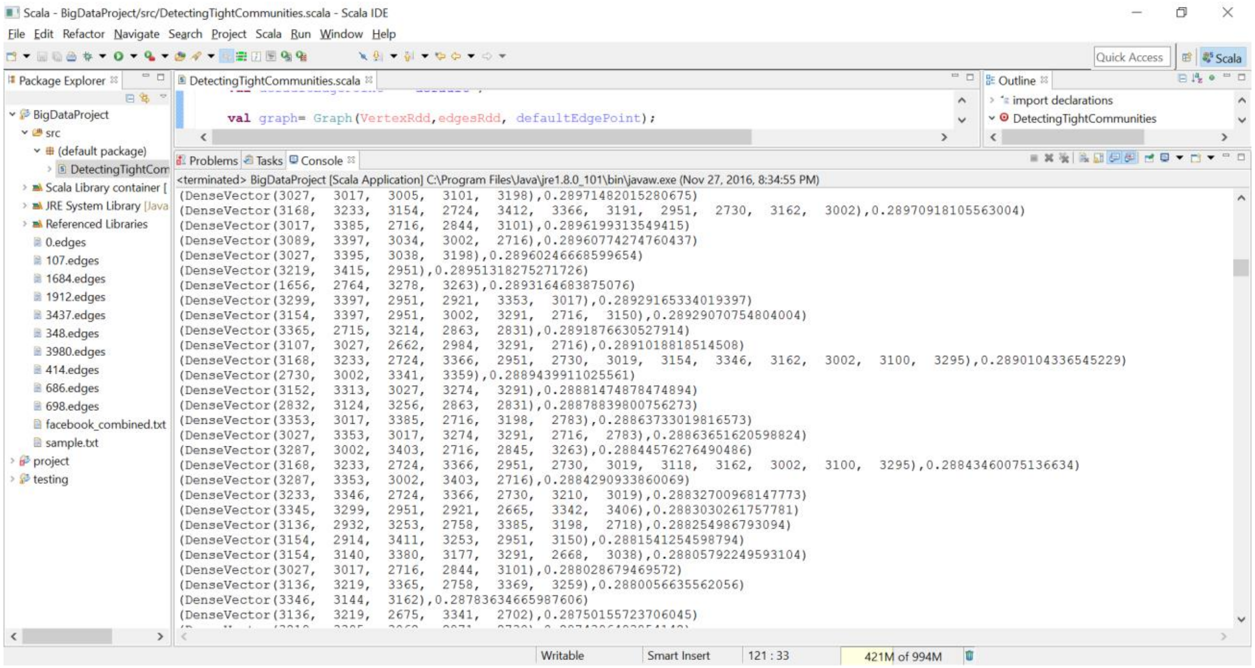


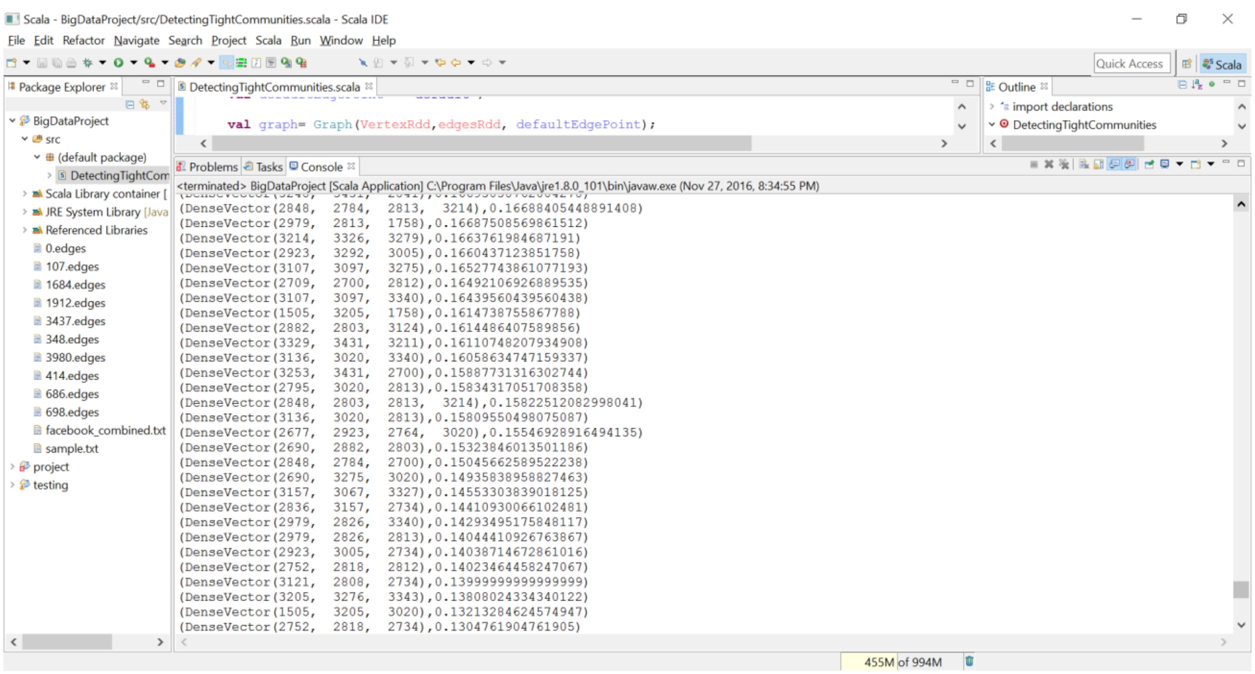
Bron-Kerbosch algorithm can be used for finding maximal cliques in an undirected graph. JGraphT has provided direct method for finding cliques in a graph.

Main logic of code:



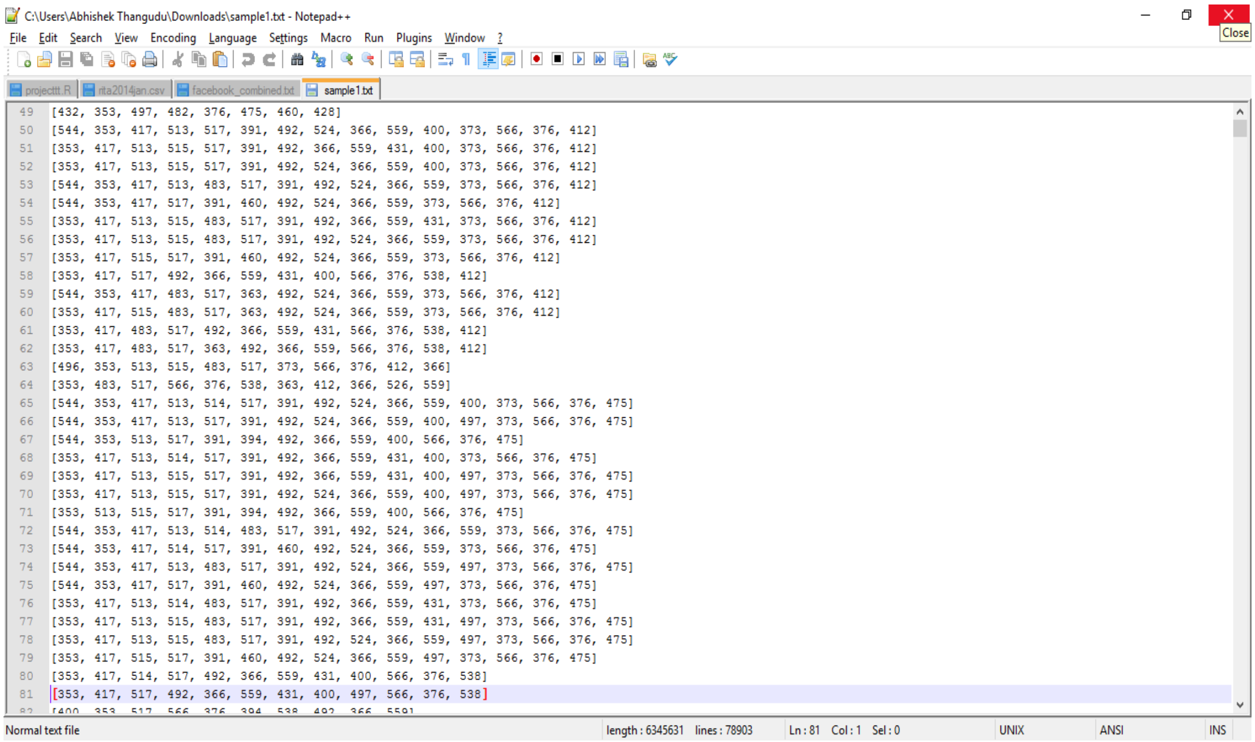
Output showing the cliques in the decreasing order of co-efficient of tightness:





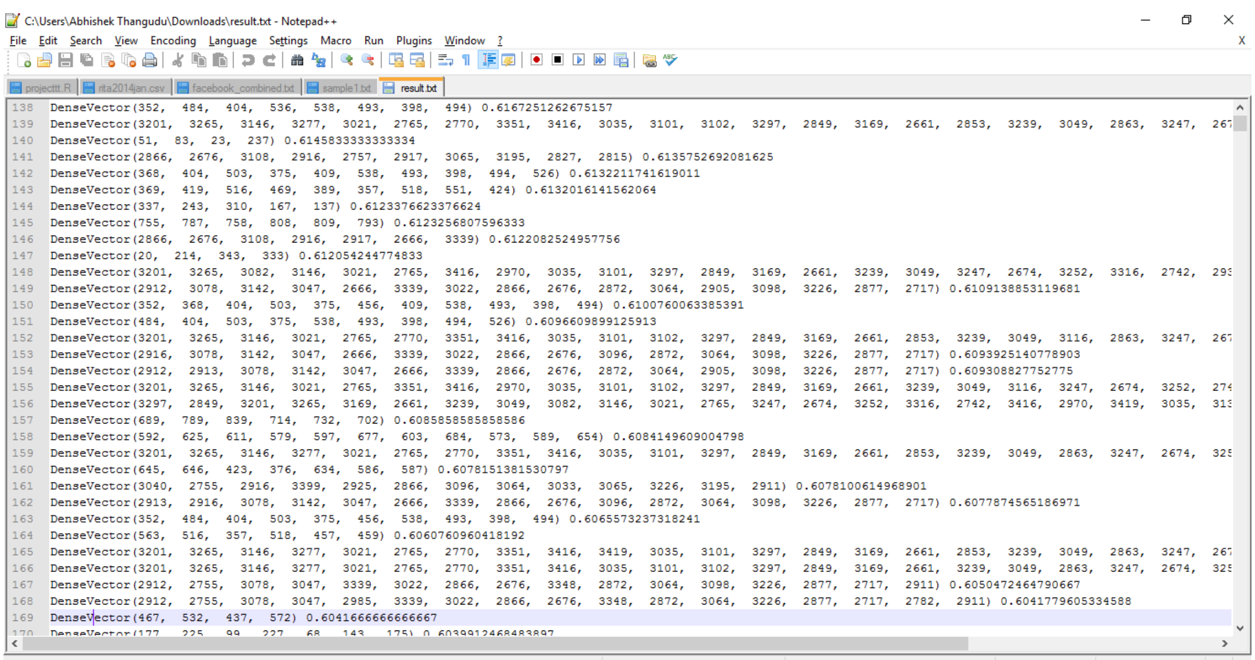
Output saved in text files:

i) All the cliques in the graph (file: sample1.txt) :




```
[432, 353, 497, 482, 376, 475, 460, 428]
[544, 353, 417, 513, 517, 391, 492, 524, 366, 559, 400, 373, 566, 376, 412]
[353, 417, 513, 515, 517, 391, 492, 366, 559, 431, 400, 373, 566, 376, 412]
[353, 417, 513, 515, 517, 391, 492, 524, 366, 559, 400, 373, 566, 376, 412]
[544, 353, 417, 513, 483, 517, 391, 492, 524, 366, 559, 373, 566, 376, 412]
[544, 353, 417, 517, 391, 460, 492, 524, 366, 559, 373, 566, 376, 412]
[353, 417, 513, 515, 483, 517, 391, 492, 366, 559, 431, 373, 566, 376, 412]
[353, 417, 513, 515, 483, 517, 391, 492, 524, 366, 559, 373, 566, 376, 412]
[353, 417, 515, 517, 391, 460, 492, 524, 366, 559, 373, 566, 376, 412]
[353, 417, 517, 492, 366, 559, 431, 400, 566, 376, 538, 412]
[544, 353, 417, 483, 517, 363, 492, 524, 366, 559, 373, 566, 376, 412]
[353, 417, 515, 483, 517, 363, 492, 524, 366, 559, 373, 566, 376, 412]
[353, 417, 483, 517, 492, 366, 559, 431, 566, 376, 538, 412]
[353, 417, 483, 517, 363, 492, 366, 559, 566, 376, 538, 412]
[496, 353, 513, 515, 483, 517, 373, 566, 376, 412, 366]
[353, 483, 517, 566, 376, 538, 363, 412, 366, 526, 559]
[544, 353, 417, 513, 514, 517, 391, 492, 524, 366, 559, 400, 373, 566, 376, 475]
[544, 353, 417, 513, 517, 391, 492, 524, 366, 559, 400, 497, 373, 566, 376, 475]
[544, 353, 513, 517, 391, 394, 492, 366, 559, 400, 566, 376, 475]
[353, 417, 513, 514, 517, 391, 492, 366, 559, 431, 400, 373, 566, 376, 475]
[353, 417, 513, 515, 517, 391, 492, 366, 559, 431, 400, 497, 373, 566, 376, 475]
[353, 417, 513, 515, 517, 391, 492, 524, 366, 559, 400, 497, 373, 566, 376, 475]
[353, 513, 515, 517, 391, 394, 492, 366, 559, 400, 566, 376, 475]
[544, 353, 417, 513, 514, 483, 517, 391, 492, 524, 366, 559, 373, 566, 376, 475]
[544, 353, 417, 514, 517, 391, 460, 492, 524, 366, 559, 373, 566, 376, 475]
[544, 353, 417, 513, 483, 517, 391, 492, 524, 366, 559, 497, 373, 566, 376, 475]
[544, 353, 417, 517, 391, 460, 492, 524, 366, 559, 497, 373, 566, 376, 475]
[353, 417, 513, 514, 483, 517, 391, 492, 366, 559, 431, 373, 566, 376, 475]
[353, 417, 513, 515, 483, 517, 391, 492, 366, 559, 431, 497, 373, 566, 376, 475]
[353, 417, 513, 515, 483, 517, 391, 492, 524, 366, 559, 497, 373, 566, 376, 475]
[353, 417, 515, 517, 391, 460, 492, 524, 366, 559, 497, 373, 566, 376, 475]
[353, 417, 514, 517, 492, 366, 559, 431, 400, 566, 376, 538]
[353, 417, 517, 492, 366, 559, 431, 400, 497, 566, 376, 538]
[400, 353, 517, 566, 376, 394, 538, 492, 366, 559]
```

ii) Final result of cliques in the graph with decreasing order of co-efficient of tightness (File : result.txt):




```
DenseVector(352,  484,  404,  536,  538,  493,  398,  494) 0.6167251262675157
DenseVector(3201,  3265,  3146,  3277,  3021,  2765,  2770,  3351,  3416,  3035,  3101,  3102,  3297,  2849,  3169,  2661,  2853,  3239,  3049,  2863,  3247,  267
DenseVector(51,  83,  23,  237) 0.6145833333333334
DenseVector(2866,  2676,  3108,  2916,  2757,  2917,  3065,  3195,  2827,  2815) 0.6135752692081625
DenseVector(368,  404,  503,  375,  409,  538,  493,  398,  494,  526) 0.6132211741619011
DenseVector(369,  419,  516,  469,  389,  357,  518,  551,  424) 0.6132016141562064
DenseVector(337,  243,  310,  167,  137) 0.6123376623376624
DenseVector(755,  787,  758,  808,  809,  793) 0.6123256807596333
DenseVector(2866,  2676,  3108,  2916,  2917,  2666,  3339) 0.6122082524957756
DenseVector(20,  214,  343,  333) 0.612054244774833
DenseVector(3201,  3265,  3082,  3146,  3021,  2765,  3416,  2970,  3035,  3101,  3297,  2849,  3169,  2661,  3239,  3049,  3247,  2674,  3252,  3316,  2742,  293
DenseVector(2912,  3078,  3142,  3047,  2666,  3339,  3022,  2866,  2676,  2872,  3064,  2905,  3098,  3226,  2877,  2717) 0.6109138853119681
DenseVector(352,  368,  404,  503,  375,  456,  409,  538,  493,  398,  494) 0.6100760063385391
DenseVector(484,  404,  503,  375,  538,  493,  398,  494,  526) 0.6096609899125913
DenseVector(3201,  3265,  3146,  3021,  2765,  2770,  3351,  3416,  3035,  3101,  3102,  3297,  2849,  3169,  2661,  2853,  3239,  3049,  3116,  2863,  3247,  267
DenseVector(2916,  3078,  3142,  3047,  2666,  3339,  3022,  2866,  2676,  3096,  2872,  3064,  3098,  3226,  2877,  2717) 0.6093925140778903
DenseVector(2912,  2913,  3078,  3142,  3047,  2666,  3339,  2866,  2676,  2872,  3064,  2905,  3098,  3226,  2877,  2717) 0.609308827752775
DenseVector(3201,  3265,  3146,  3021,  2765,  3351,  3416,  2970,  3035,  3101,  3102,  3297,  2849,  3169,  2661,  3239,  3049,  3116,  3247,  2674,  3252,  274
DenseVector(3297,  2849,  3201,  3265,  3169,  2661,  3239,  3049,  3082,  3146,  3021,  2765,  3247,  2674,  3252,  3316,  2742,  3416,  2970,  3419,  3035,  313
DenseVector(689,  789,  839,  714,  732,  702) 0.6085858585858586
DenseVector(592,  625,  611,  579,  597,  677,  603,  684,  573,  589,  654) 0.6084149609004798
DenseVector(3201,  3265,  3146,  3277,  3021,  2765,  2770,  3351,  3416,  3035,  3101,  3297,  2849,  3169,  2661,  2853,  3239,  3049,  2863,  3247,  2674,  325
DenseVector(645,  646,  423,  376,  634,  586,  587) 0.6078151381530797
DenseVector(3040,  2755,  2916,  3399,  2925,  2866,  3096,  3064,  3033,  3065,  3226,  3195,  2911) 0.6078100614968901
DenseVector(2913,  2916,  3078,  3142,  3047,  2666,  3339,  2866,  2676,  3096,  2872,  3064,  3098,  3226,  2877,  2717) 0.6077874565186971
DenseVector(352,  484,  404,  503,  375,  456,  538,  493,  398,  494) 0.6065573237318241
DenseVector(563,  516,  357,  518,  457,  459) 0.6060760960418192
DenseVector(3201,  3265,  3146,  3277,  3021,  2765,  2770,  3351,  3416,  3419,  3035,  3101,  3297,  2849,  3169,  2661,  2853,  3239,  3049,  2863,  3247,  267
DenseVector(3201,  3265,  3146,  3277,  3021,  2765,  2770,  3351,  3416,  3035,  3101,  3102,  3297,  2849,  3169,  2661,  3239,  3049,  2863,  3247,  2674,  325
DenseVector(2912,  2755,  3078,  3047,  3339,  3022,  2866,  2676,  3348,  2872,  3064,  3098,  3226,  2877,  2717,  2911) 0.6050472464790667
DenseVector(2912,  2755,  3078,  3047,  2985,  3339,  3022,  2866,  2676,  3348,  2872,  3064,  3226,  2877,  2717,  2782,  2911) 0.6041779605334588
DenseVector(467,  532,  437,  572) 0.6041666666666667
DenseVector(177,  225,  99,  227,  68,  143,  175) 0.6039912468483897
```

4. Future work

The analysis can further be extended by comparing the features of each User in a given tight community and thereby determining which feature made them to be in a tight community. We can also measure connectivity of each Facebook group by considering only sub-graph of a group separately and measuring the tightness. But in the dataset provided, groups of different ego networks are not connected properly; they are just named in a sequential manner. There is no way connecting the members in one ego network to other.

5. Conclusion

Finding Cliques is one of the best ways in determining the tight communities in a given network. Strongly connected components can just tell if there is a path between any two distinct users that can be connected whereas in a Clique there is a direct connection among every distinct User in the sub-network.

6. References

1. http://spark.apache.org/graphx/
2. http://jgrapht.org/javadoc/org/jgrapht/alg/BronKerboschCliqueFinder.html
3. http://stackoverflow.com/questions/31217642/finding-cliques-or-strongly-connected-components-in-apache-spark-using-graphx
4. https://en.wikipedia.org/wiki/Bron%E2%80%93Kerbosch_algorithm