Social Networking Analysis

Submitted To:
Dr Sakthi Balan Muthiah

Dataset #1 Chosen from Orkut's Database

About the Dataset:
This is the social network of Orkut users and their connections.

Properties of Edges in the Dataset:

- Edges are undirected. That is, there is no difference between the edge from u to v and the edge from v to u; both are the edge {u, v}.
- Edges that are unweighted and only a single edge is allowed between any two nodes.
- Diameter: 10 edges

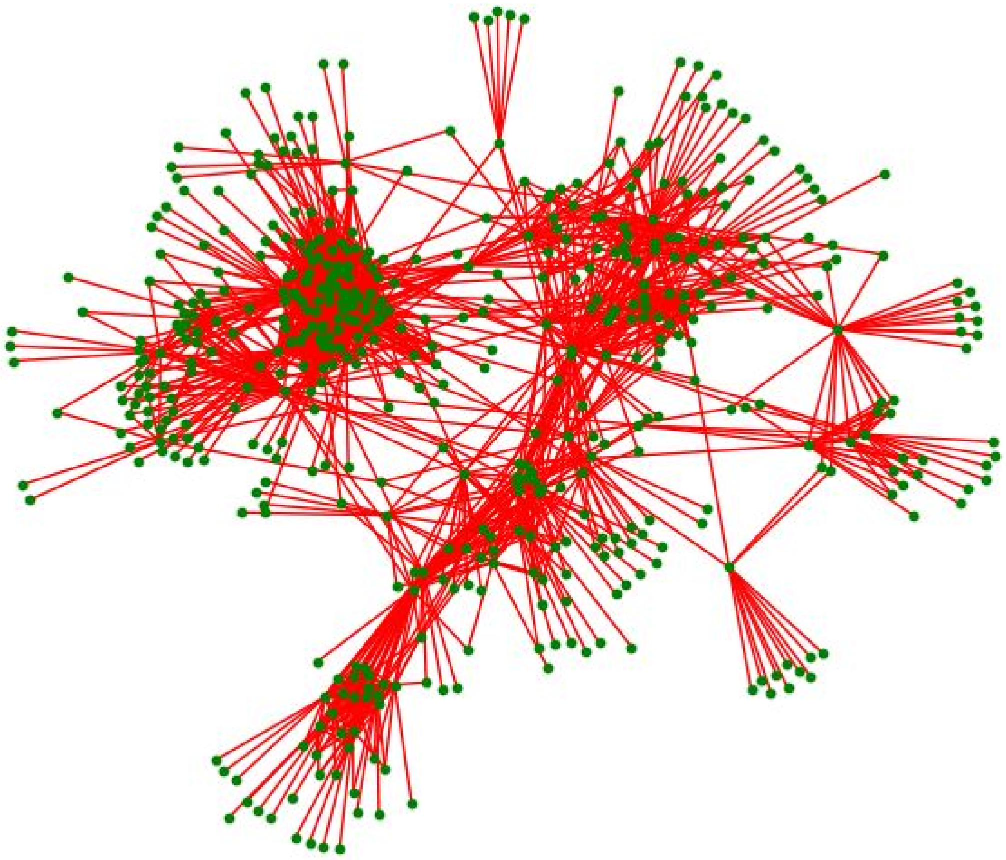
Complete Dataset
Link to Dataset: http://konect.uni-koblenz.de/networks/orkut-links
Size: 3,072,441 Vertices
Volume: 117,184,899 Edges
Average Degree: 70.281 edges / vertex
Each vertex represents a user.

Sampled Dataset
Size: 549 Vertices
Volume: 1925 Edges
Average Degree: 7.0128 edges / vertex

Source of Dataset: The Max Planck Institute for Software Systems, IMC 2007 paper.

Process of Sampling: We've filtered out all the nodes which are between 7000-8000 using python code, so as to get a better and more representative dataset.

Directed Graph Image



Initial Setup

Firstly, we imported all the required Python libraries and dependencies. The NetworkX library for studying graphs and networks. Matplotlib is mainly used for plotting graphs.

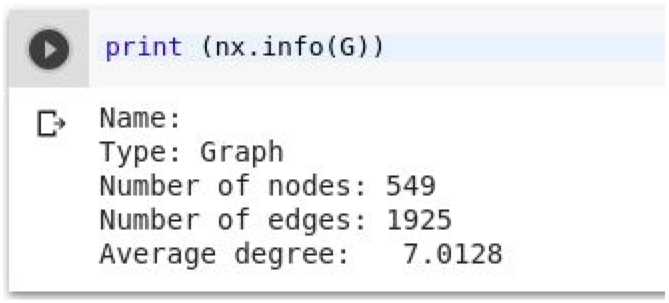
```
import networkx as nx
import matplotlib.pyplot as plt
import math
import matplotlib.colors as mcolors
```

This command sets the backend of matplotlib to the 'inline' backend.

```
%matplotlib inline
```

This command reads a graph from a list of edges.

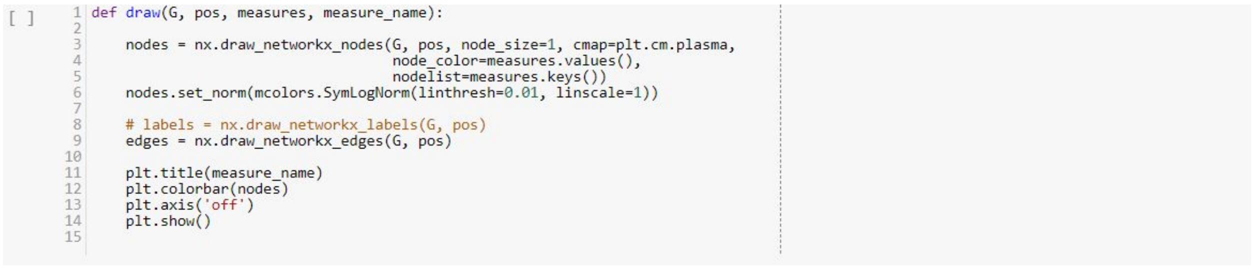
```
G=nx.read_edgelist('sample.txt')
```



```
print (nx.info(G))
```

```
Name:
Type: Graph
Number of nodes: 549
Number of edges: 1925
Average degree:   7.0128
```

This command is used to draw the nodes of the graph G. This draws only the nodes of the graph G.



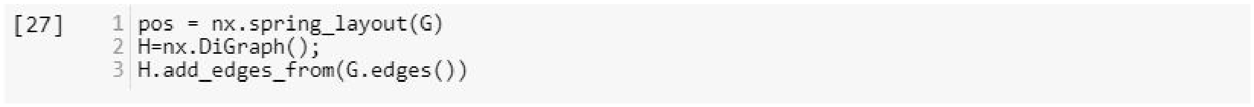
```
def draw(G, pos, measures, measure_name):

    nodes = nx.draw_networkx_nodes(G, pos, node_size=1, cmap=plt.cm.plasma,
                                   node_color=measures.values(),
                                   nodelist=measures.keys())
    nodes.set_norm(mcolors.SymLogNorm(linthresh=0.01, linscale=1))

    # labels = nx.draw_networkx_labels(G, pos)
    edges = nx.draw_networkx_edges(G, pos)

    plt.title(measure_name)
    plt.colorbar(nodes)
    plt.axis('off')
    plt.show()
```

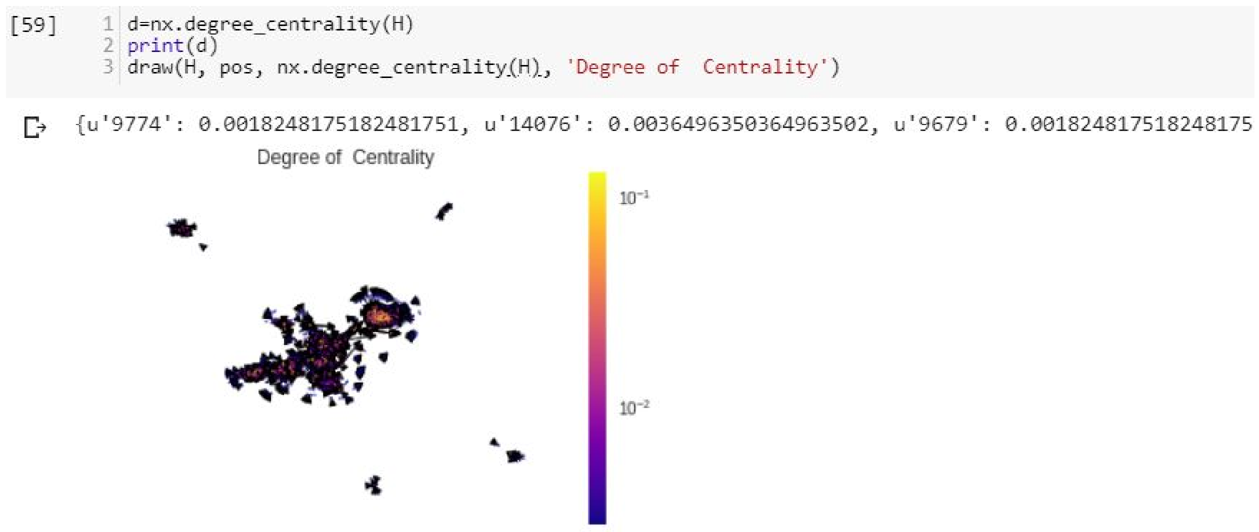
A DiGraph stores nodes and edges with optional data, or attributes. This is a Base class for directed graphs.



```
[27]
pos = nx.spring_layout(G)
H=nx.DiGraph();
H.add_edges_from(G.edges())
```

Degree Centrality:

It is defined as the number of links incident upon a node (i.e., the number of tie that a node has).



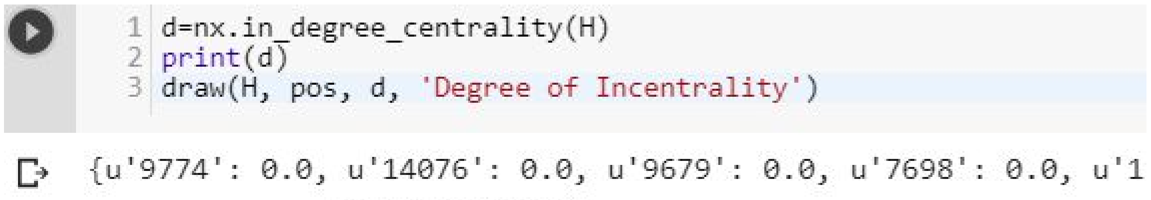
```
[59]  d=nx.degree_centrality(H)
      print(d)
      draw(H, pos, nx.degree_centrality(H), 'Degree of  Centrality')
```

```
{u'9774': 0.0018248175182481751, u'14076': 0.0036496350364963502, u'9679': 0.001824817518248175
```

First five Values are:
0.0018248175182481751
0.0036496350364963502
0.0018248175182481751
0.003649635036496350
0.0018248175182481751

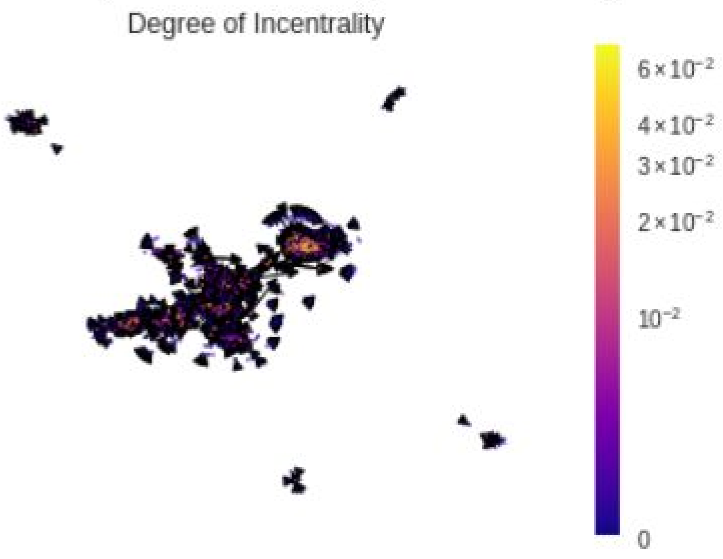
InDegree Centrality:

In-degree is the number of incoming links or the number of predecessor nodes



```
d=nx.in_degree_centrality(H)
print(d)
draw(H, pos, d, 'Degree of Incentrality')
```

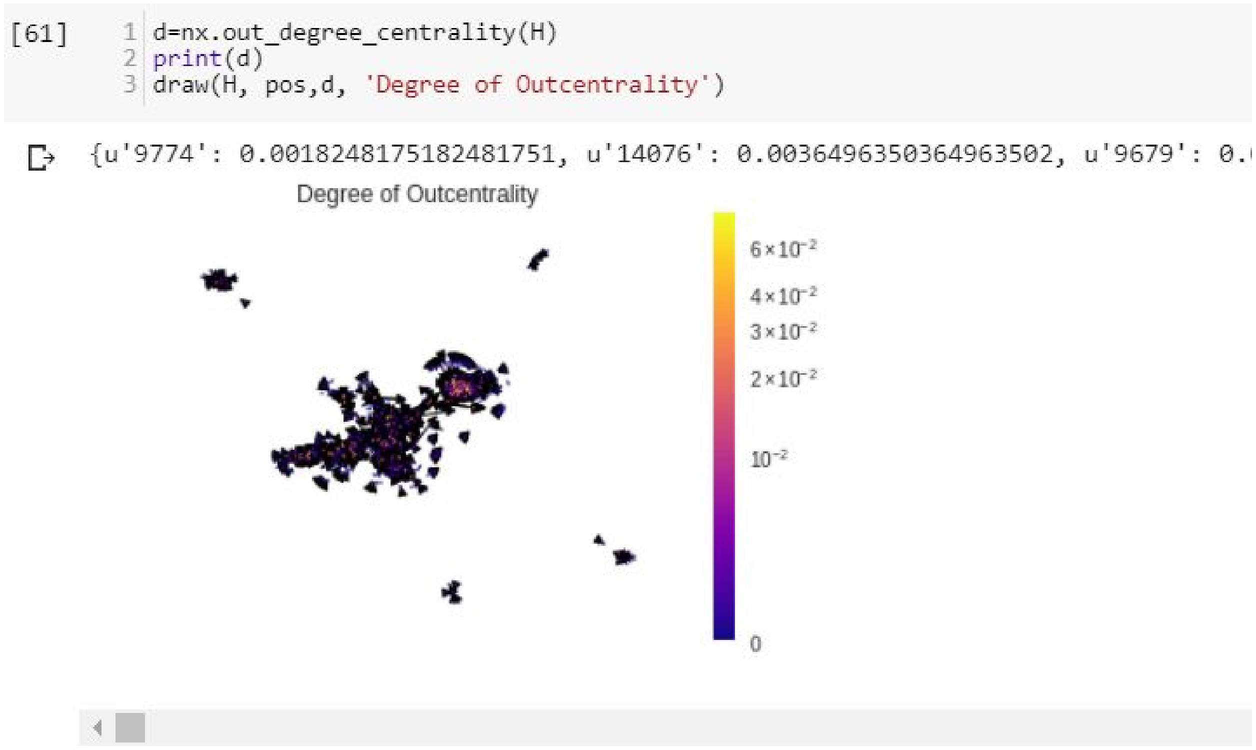
```
{u'9774': 0.0, u'14076': 0.0, u'9679': 0.0, u'7698': 0.0, u'1
```



First five Values are:
0.0
0.0
0.0
0.0018248175182481751
0.003649635036496350

OutDegree Centrality:

Out-degree is the number of outgoing links, or the number of successor nodes.



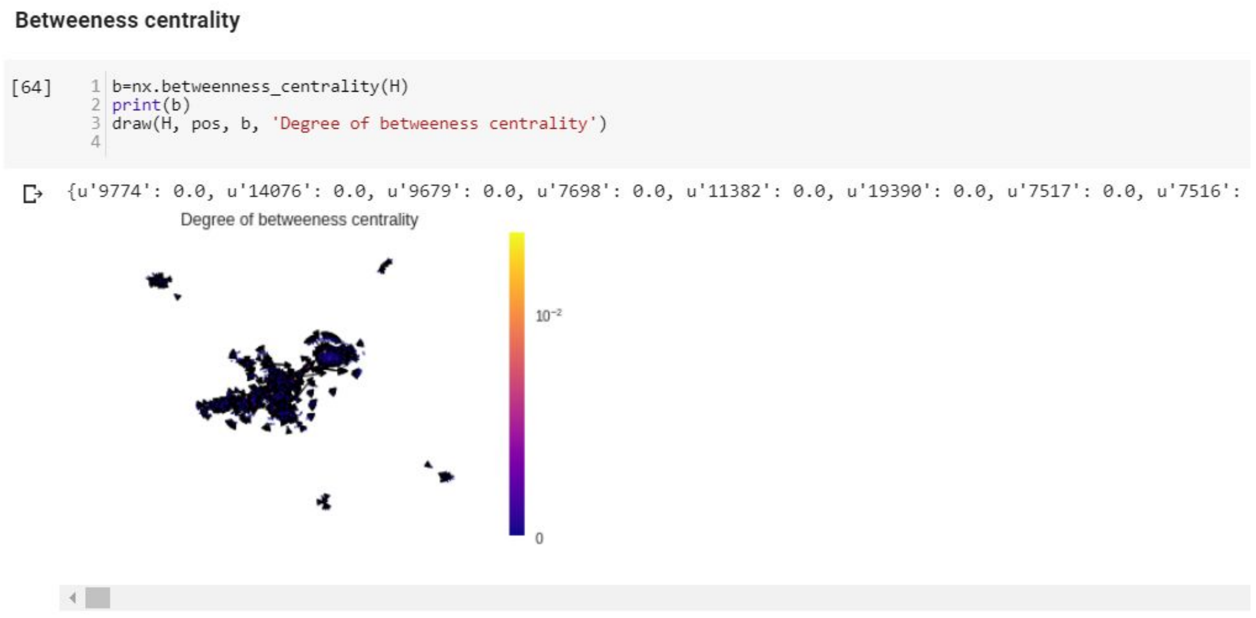
```
[61]  d=nx.out_degree_centrality(H)
      print(d)
      draw(H, pos,d, 'Degree of Outcentrality')
```

```
{u'9774': 0.0018248175182481751, u'14076': 0.0036496350364963502, u'9679': 0.
```

First five Values are:
0.0018248175182481751
0.0036496350364963502
0.0018248175182481751
0.003649635036496350
0.001824817518248175

Betweenness Centrality:

A measure of the influence of a vertex over the flow of information between every pair of vertices under the assumption that information primarily flows over the shortest paths between them.



Betweeness centrality

```
[64]  b=nx.betweenness_centrality(H)
      print(b)
      draw(H, pos, b, 'Degree of betweeness centrality')
```

```
{u'9774': 0.0, u'14076': 0.0, u'9679': 0.0, u'7698': 0.0, u'11382': 0.0, u'19390': 0.0, u'7517': 0.0, u'7516':
```

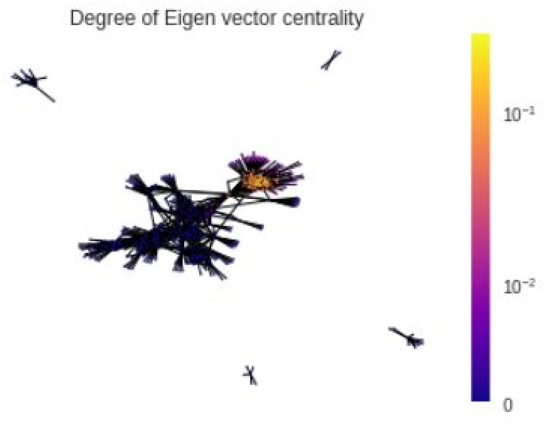
First five Values are:
0.0
0.0
0.0
0.0
0.0

Eigenvector Centrality:

A method of computing the "centrality", or approximate importance, of each node in a graph. The assumption is that each node's centrality is the sum of the centrality values of the nodes that it is connected to. The nodes are drawn with a radius proportional to their centrality.

```
[62]  e= nx.eigenvector_centrality(G)
      print(e)
      draw(G, pos, e, 'Degree of Eigen vector centrality')
```

```
{u'9774': 4.7164268976967324e-20, u'14076': 5.776547110020189e-05, u'9679': 4.716426897696
```

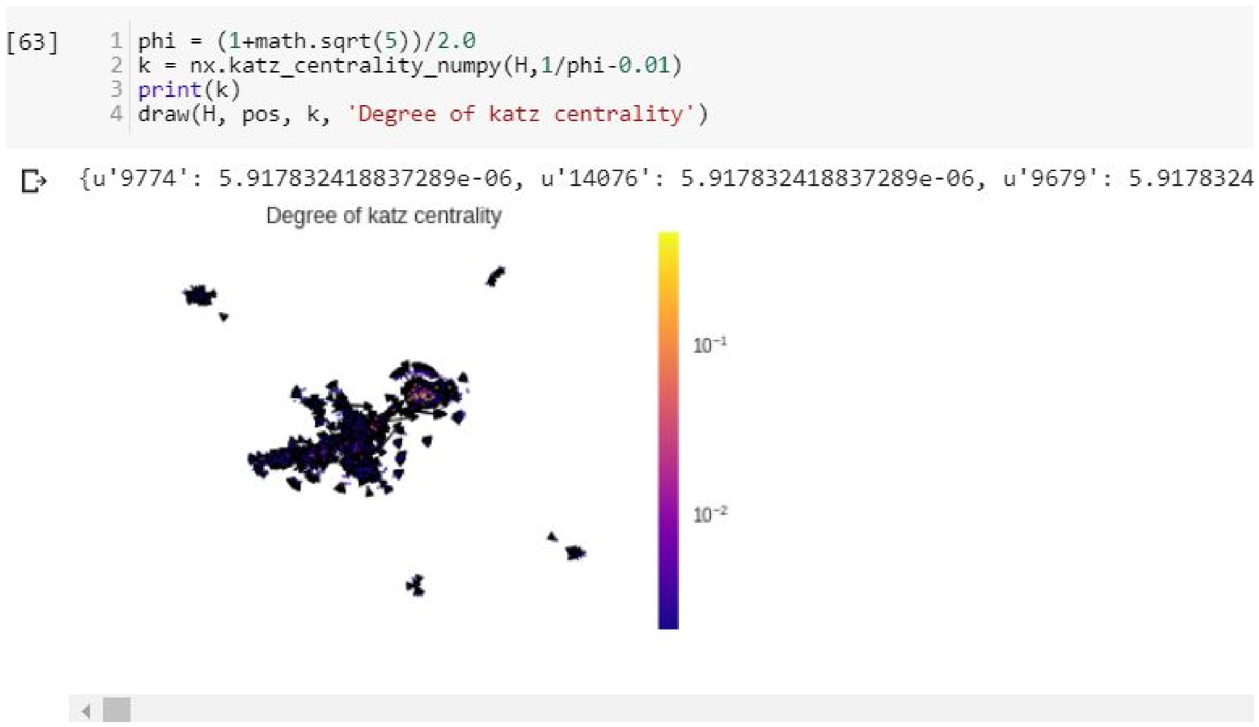


First five Values are:
4.7164268976967324e-20
5.776547110020189e-05
4.7164268976967324e-20
2.1460370924860818e-05
4.841087212369237e-06

Katz Centrality:

A measure of centrality in a network and is used to measure the relative degree of influence of an actor within a social network.

Katz centrality



```
[63]  phi = (1+math.sqrt(5))/2.0
      k = nx.katz_centrality_numpy(H,1/phi-0.01)
      print(k)
      draw(H, pos, k, 'Degree of katz centrality')
```

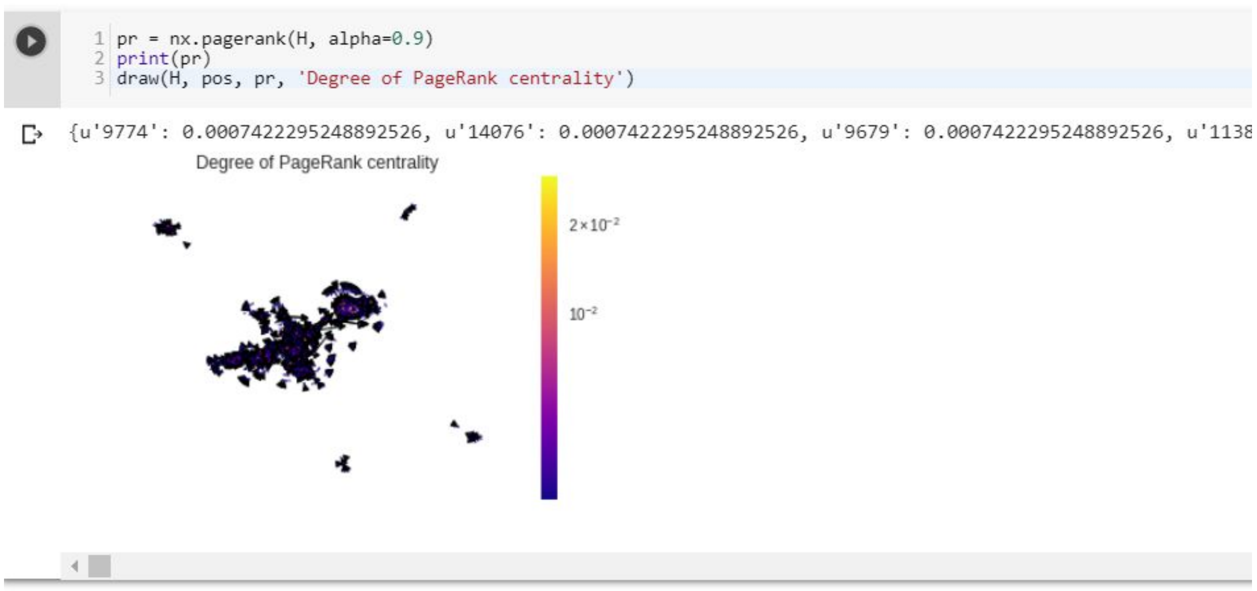
```
{u'9774': 5.917832418837289e-06, u'14076': 5.917832418837289e-06, u'9679': 5.9178324
```

First five Values are:
5.917832418837289e-06
5.917832418837289e-06
5.917832418837289e-06
9.516075669216363e-06
5.917832418837289e-06

PageRank Centrality:

A potential problem with Katz centrality is the following: if a node with high centrality links many others then all those others get high centrality. In many cases, however, it means less if a node is only one among many to be linked.

Page Rank Centrality



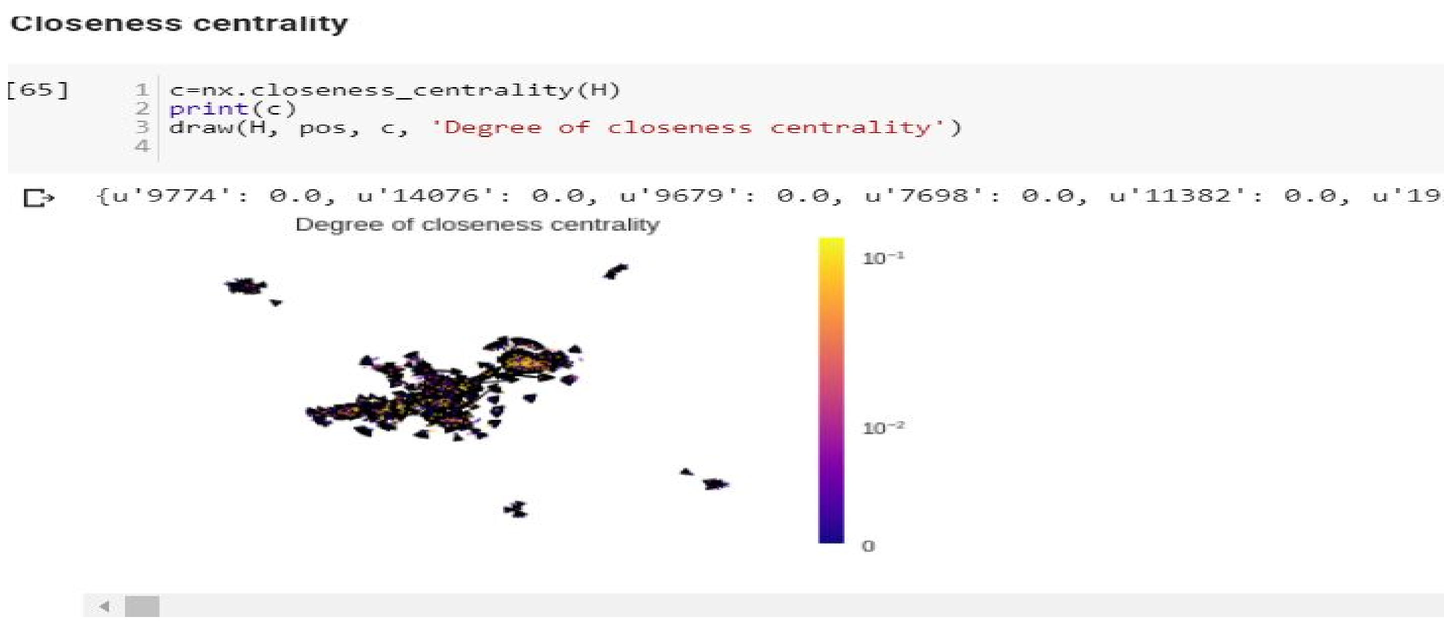
```
pr = nx.pagerank(H, alpha=0.9)
print(pr)
draw(H, pos, pr, 'Degree of PageRank centrality')
```

```
{u'9774': 0.0007422295248892526, u'14076': 0.0007422295248892526, u'9679': 0.0007422295248892526, u'1138
```

First five Values are:
0.0007422295248892526
0.0007422295248892526
0.0007422295248892526
0.0007422295248892526
0.0007422295248892526

Closeness Centrality:

It is a measure of centrality in a network, calculated as the reciprocal of the sum of the length of the shortest paths between the node and all other nodes in the graph.



Closeness centrality

```
[65]
c=nx.closeness_centrality(H)
print(c)
draw(H, pos, c, 'Degree of closeness centrality')
```

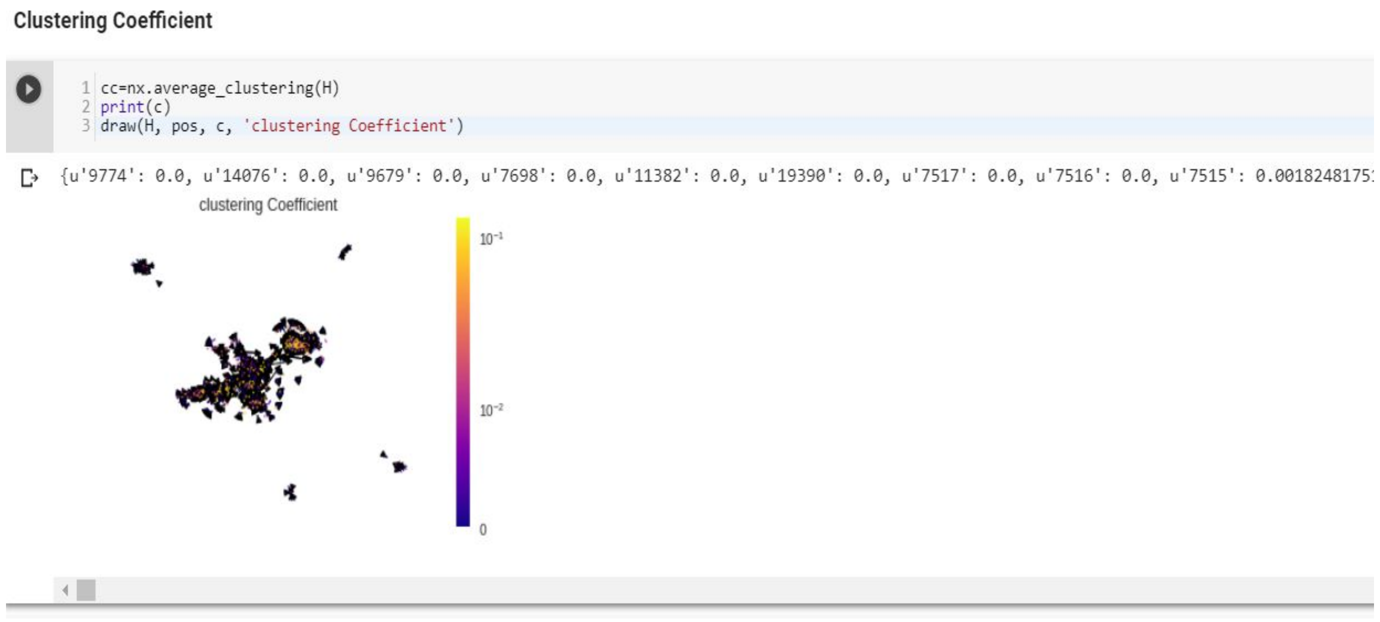
```
{u'9774': 0.0, u'14076': 0.0, u'9679': 0.0, u'7698': 0.0, u'11382': 0.0, u'19
```

First five Values are:
0.0
0.0
0.0
0.001824817518248175

Clustering Coefficient

Global:

First five Values are:



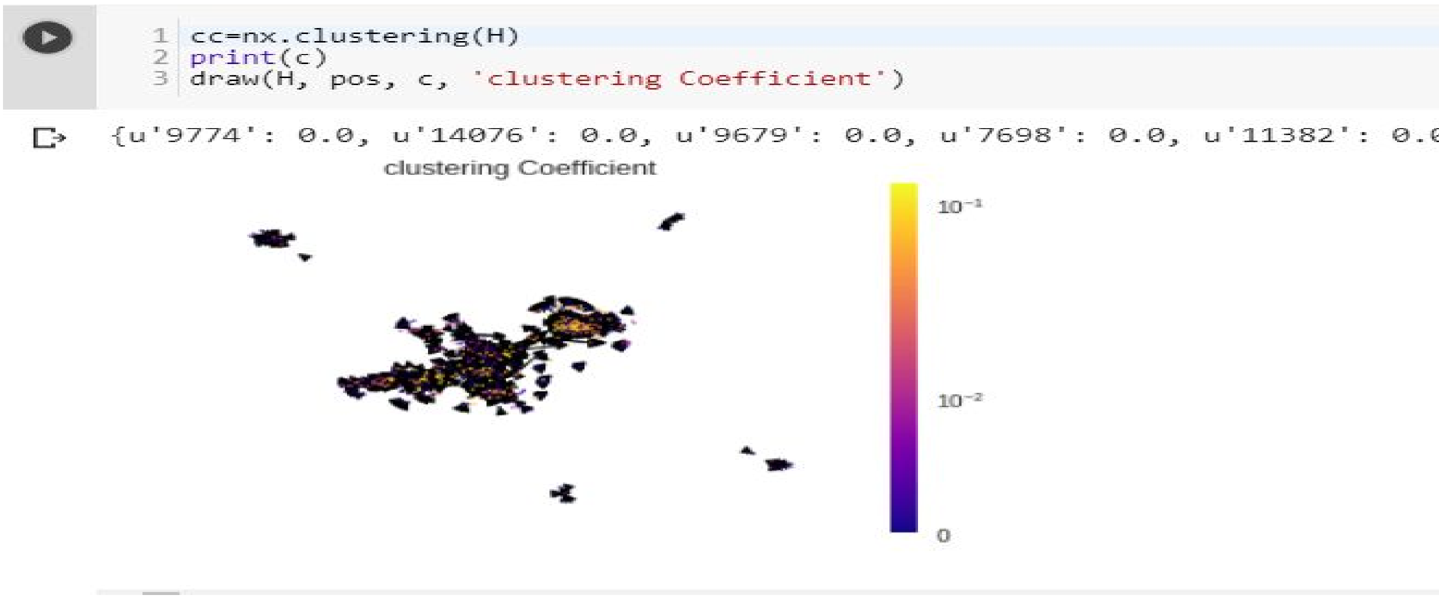
Clustering Coefficient

```
cc=nx.average_clustering(H)
print(c)
draw(H, pos, c, 'clustering Coefficient')
```

```
{u'9774': 0.0, u'14076': 0.0, u'9679': 0.0, u'7698': 0.0, u'11382': 0.0, u'19390': 0.0, u'7517': 0.0, u'7516': 0.0, u'7515': 0.0018248175
```

First five Values are:
0.0
0.0
0.0018248175182481751
0.0036496350364963502
0.0018248175182481751

<u>Local:</u>



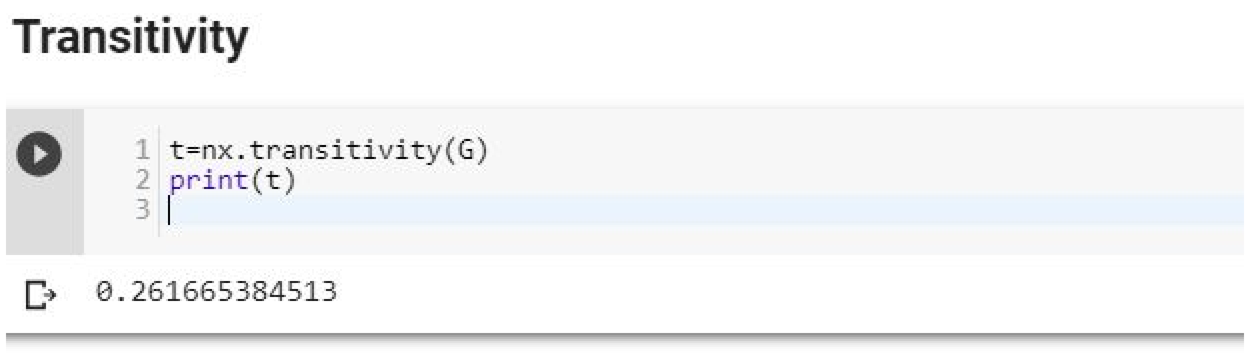
```
cc=nx.clustering(H)
print(c)
draw(H, pos, c, 'clustering Coefficient')
```

```
{u'9774': 0.0, u'14076': 0.0, u'9679': 0.0, u'7698': 0.0, u'11382': 0.0
```

First five Values are:
0.0
0.0
0.0
0.0018248175182481751
0.0024330900243309

Transitivity

In transitivity, we analyze the linking behaviour to determine whether it demonstrates a transitive behaviour



Transitivity

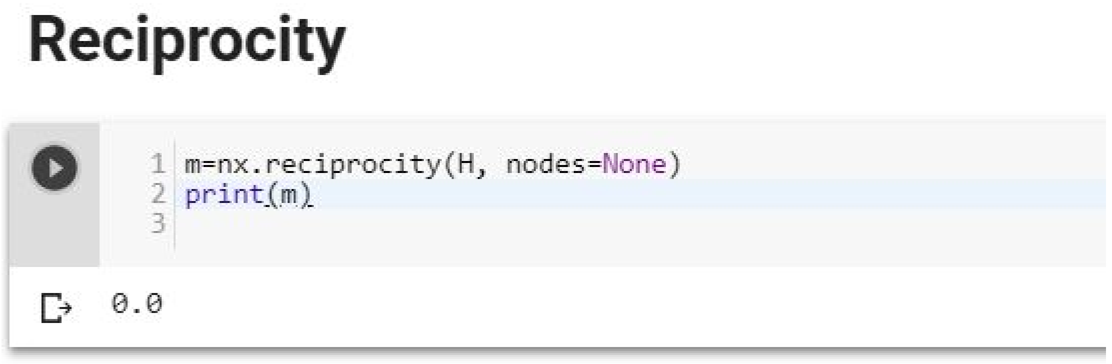
```
t=nx.transitivity(G)
print(t)

0.261665384513
```

Value of transitivity is .261665384513

Reciprocity

It is a measure of the likelihood of vertices in a directed network to be mutually linked. Like the clustering coefficient, scale-free degree derogree distribution, or community structure, reciprocity is a quantitative measure used to study complex networks.



Reciprocity

```
m=nx.reciprocity(H, nodes=None)
print(m)

0.0
```

Best Centrality Measure for Dataset #1:

Out of all the centrality measures, **EigenCentrality is the best measure to measure the centrality of Orkut**. This is because EigenCentrality is a good ‘all-round’ SNA score, handy for understanding human social networks, but also for understanding networks like malware propagation.

Dataset #2 Chosen from Twitter (WWW) Database

Properties of Edges in the Dataset:

- Edges are directed. That is, there is a difference between the edge (u, v) and the edge (u, v).
- Edges that are unweighted, and only a single edge is allowed between any two nodes.
- Diameter: 23 edges

Complete Dataset
Link to Dataset: http://konect.uni-koblenz.de/networks/twitter
Size: 41,652,230 Vertices
Volume: 1,468,365,182 Edges
Average Degree: 70.506 edges / vertex
Each vertex represents a user.

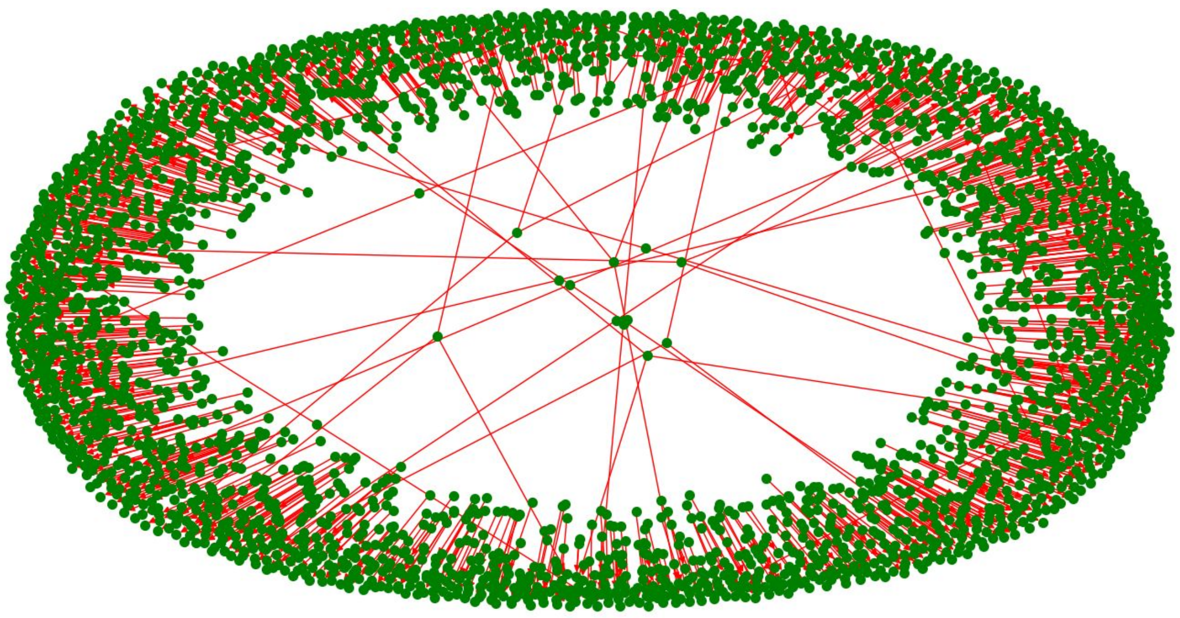
Sampled Dataset
Size: 2919 Vertices
Volume: 1500 Edges
Average Degree: 1.027 edges / vertex
About the Network: This is the follower network from Twitter, containing 1.4 billion directed follow edges between 41 million Twitter users.

Source of Dataset: Proceedings of the 19th International World Wide Web (WWW) Conference, April 26-30, 2010

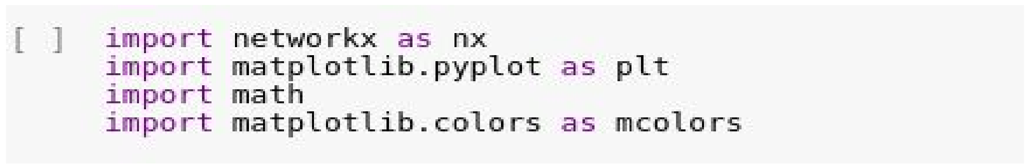
Process of Sampling: We’ve randomly selected 1000 vertices, from a dataset of 41,652,230 vertices. We used the following command to extract them:

```
shuf -n 1000 old_filename.tsv > new_filename.txt
```

Directed Graph Image



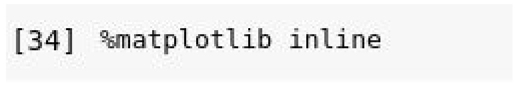
Firstly, we imported all the required Python libraries and dependencies. The NetworkX library for studying graphs and networks. Matplotlib is mainly used for plotting graphs



```
[ ]  import networkx as nx
     import matplotlib.pyplot as plt
     import math
     import matplotlib.colors as mcolors
```

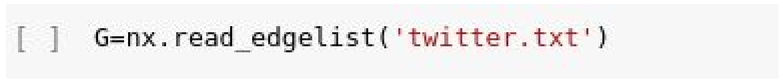
.

This command sets the backend of matplotlib to the 'inline' backend.

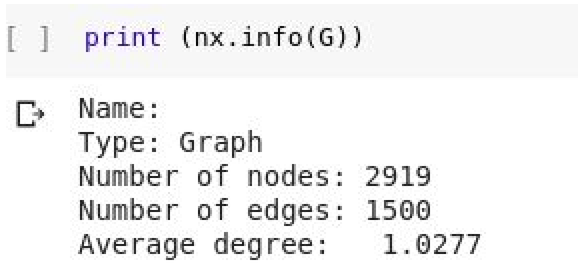


```
[34] %matplotlib inline
```

This command reads a graph from a list of edges.



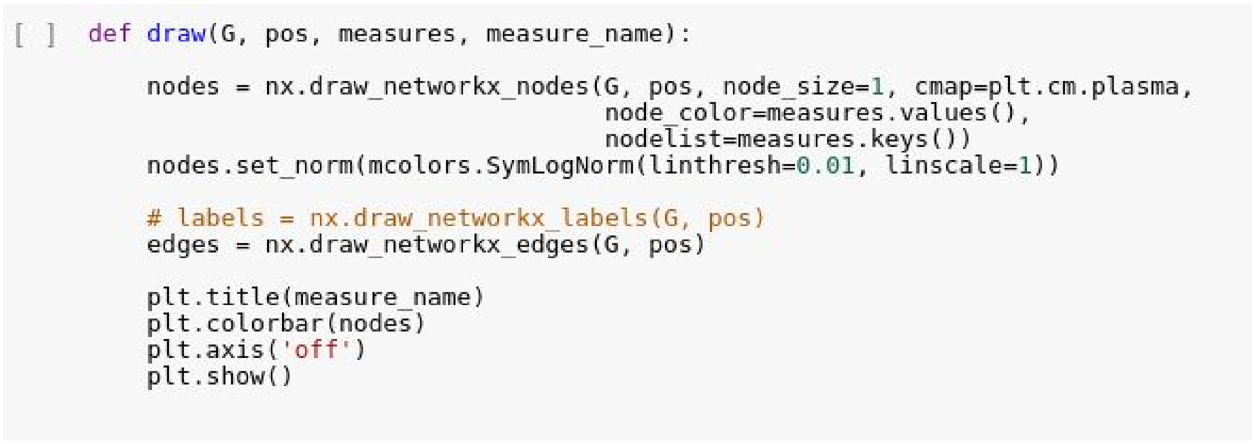
```
[ ]  G=nx.read_edgelist('twitter.txt')
```



```
[ ]  print (nx.info(G))
```

```
Name:
Type: Graph
Number of nodes: 2919
Number of edges: 1500
Average degree:   1.0277
```

This command is used to draw the nodes of the graph G. This draws only the nodes of the graph G.



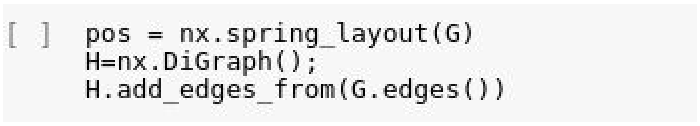
```
[ ]  def draw(G, pos, measures, measure_name):

         nodes = nx.draw_networkx_nodes(G, pos, node_size=1, cmap=plt.cm.plasma,
                                        node_color=measures.values(),
                                        nodelist=measures.keys())
         nodes.set_norm(mcolors.SymLogNorm(linthresh=0.01, linscale=1))

         # labels = nx.draw_networkx_labels(G, pos)
         edges = nx.draw_networkx_edges(G, pos)

         plt.title(measure_name)
         plt.colorbar(nodes)
         plt.axis('off')
         plt.show()
```

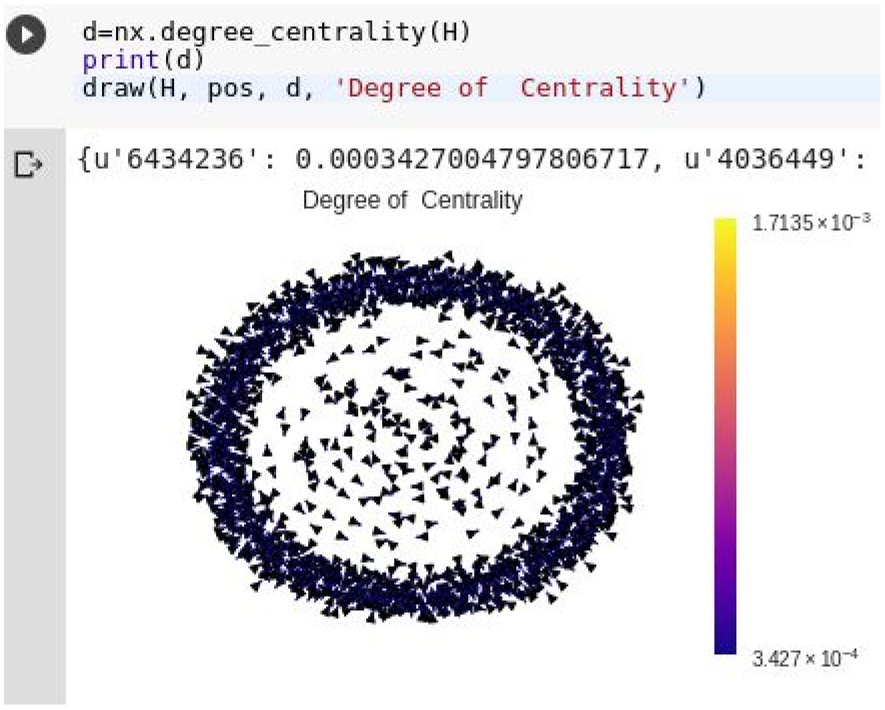
A DiGraph stores nodes and edges with optional data, or attributes. This is a Base class for directed graphs.



```
[ ]  pos = nx.spring_layout(G)
     H=nx.DiGraph();
     H.add_edges_from(G.edges())
```

Degree Centrality:

It is defined as the number of links incident upon a node (i.e., the number of ties that a node has).



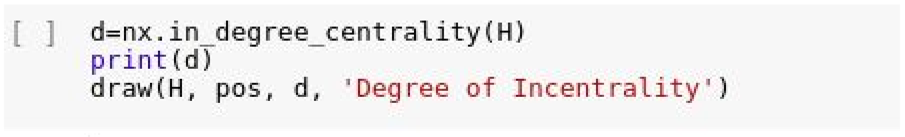
```
d=nx.degree_centrality(H)
print(d)
draw(H, pos, d, 'Degree of  Centrality')
```

```
{u'6434236': 0.0003427004797806717, u'4036449':
```

Five Five Values:
0.0003427004797806717,
0.0003427004797806717,
0.0003427004797806717,
0.0003427004797806717

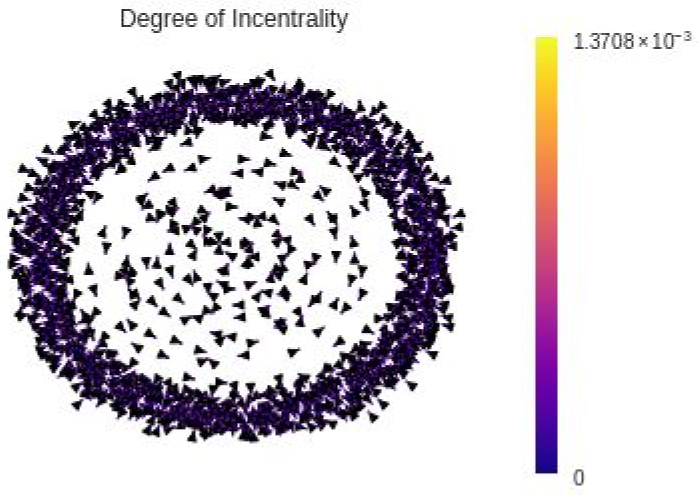
InDegree Centrality:

In-degree is the number of incoming links, or the number of predecessor nodes



```
[ ]  d=nx.in_degree_centrality(H)
     print(d)
     draw(H, pos, d, 'Degree of Incentrality')
```

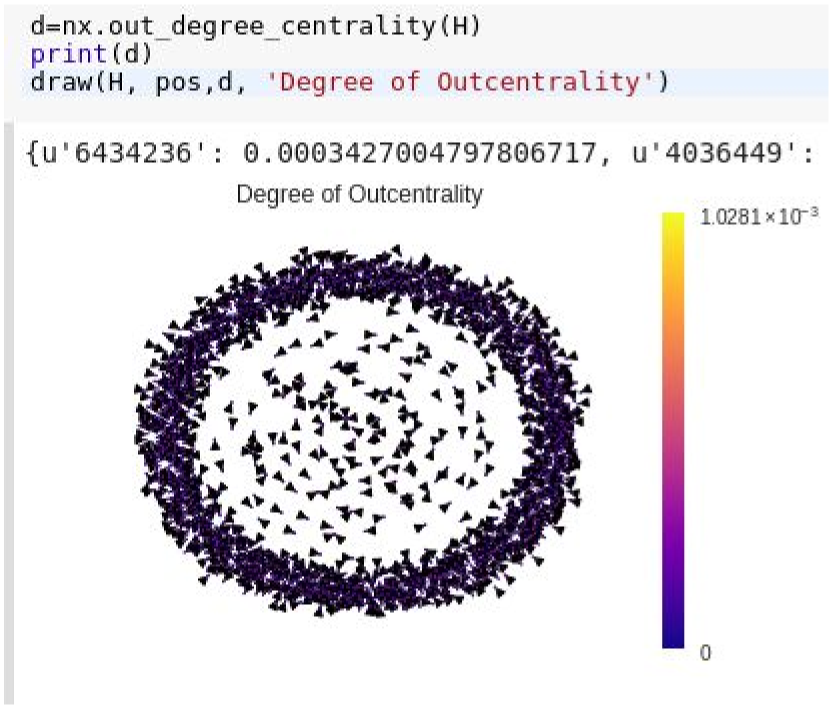
```
{u'6434236': 0.0, u'4036449': 0.0, u'8683828': 0
```



Five Five Values:
0.0,
0.0,
0.0,
0.0018248175182481751
0.0024330900243309

OutDegree Centrality:

Out-degree is the number of outgoing links, or the number of successor nodes.



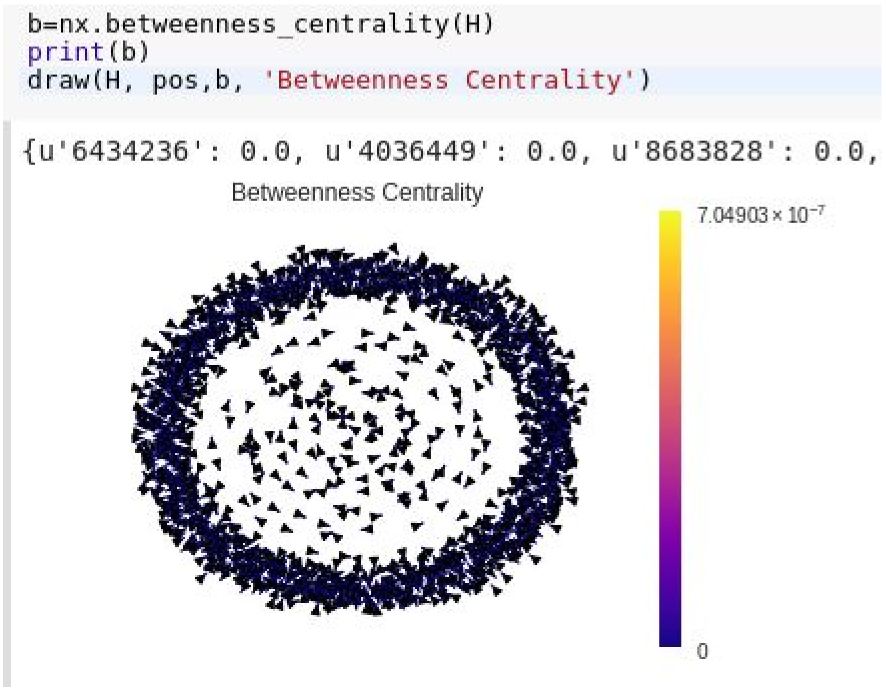
```
d=nx.out_degree_centrality(H)
print(d)
draw(H, pos,d, 'Degree of Outcentrality')
```

```
{u'6434236': 0.0003427004797806717, u'4036449':
```

Five Five Values:
0.0003427004797806717,
0.0003427004797806717,
0.0003427004797806717,
0.0003427004797806717,
0.0003427004797806717

Betweenness Centrality:

A measure of the influence of a vertex over the flow of information between every pair of vertices under the assumption that information primarily flows over the shortest paths between them.



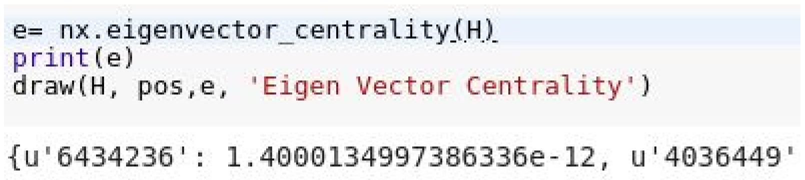
```
b=nx.betweenness_centrality(H)
print(b)
draw(H, pos,b, 'Betweenness Centrality')
```

```
{u'6434236': 0.0, u'4036449': 0.0, u'8683828': 0.0,
```

Five Five Values:
0.0,
0.0,
0.0,
0.0018248175182481751
0.0024330900243309

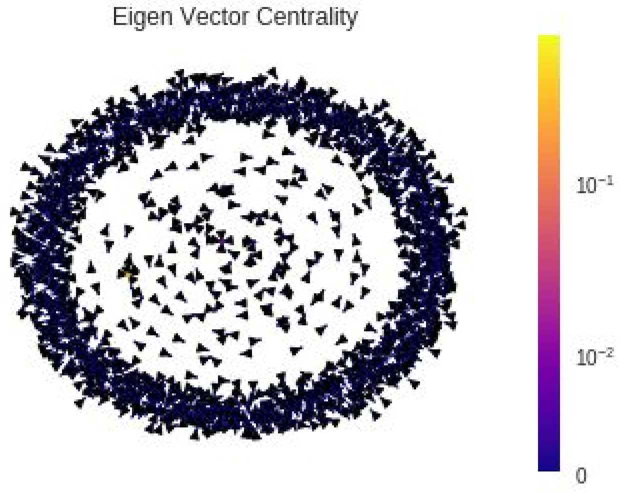
Eigenvector Centrality:

A method of computing the "centrality", or approximate importance, of each node in a graph. The assumption is that each node's centrality is the sum of the centrality values of the nodes that it is connected to. The nodes are drawn with a radius proportional to their centrality.



```
e= nx.eigenvector_centrality(H)
print(e)
draw(H, pos,e, 'Eigen Vector Centrality')
```

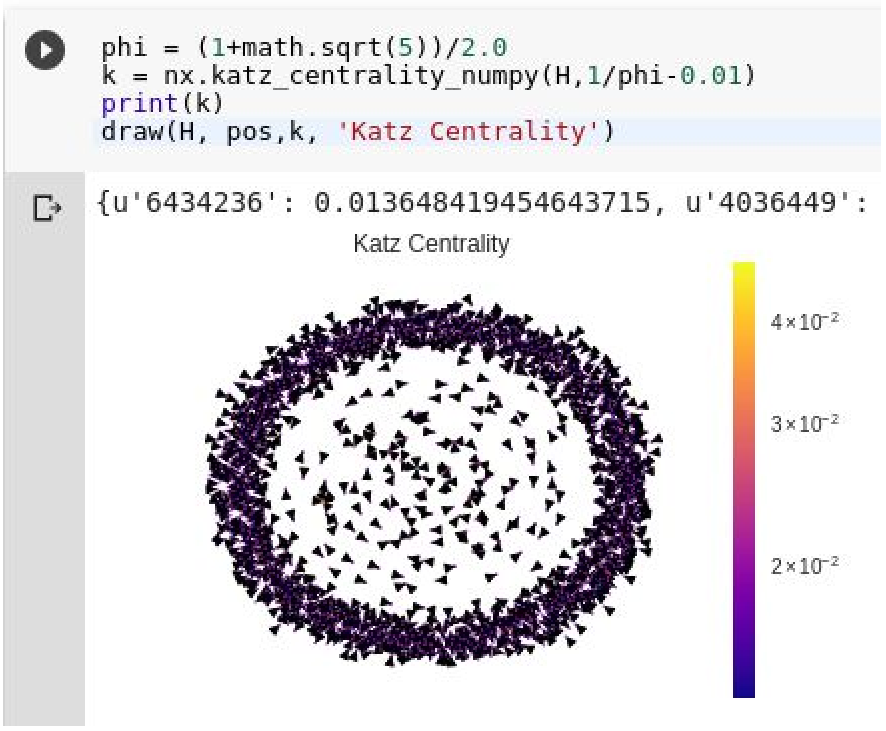
```
{u'6434236': 1.4000134997386336e-12, u'4036449'
```



Five Five Values:
1.4000134997386336e-12,
1.4000134997386336e-12,
1.4000134997386336e-12,
1.4000134997386336e-12,
1.4000134997386336e-12

Katz Centrality:

A measure of centrality in a network and is used to measure the relative degree of influence of an actor within a social network.



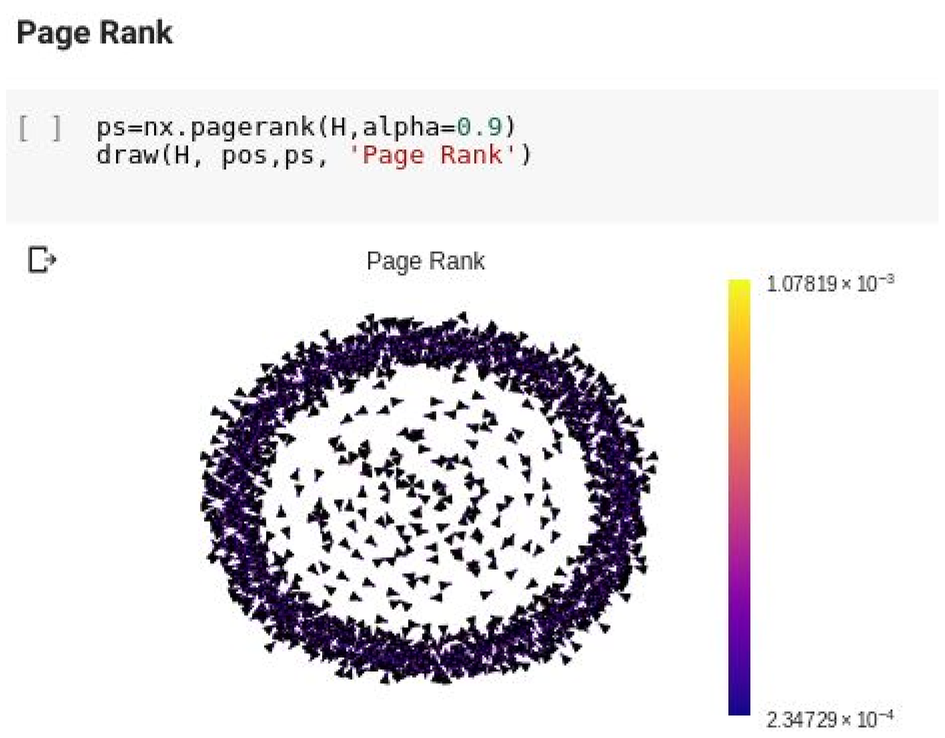
```
phi = (1+math.sqrt(5))/2.0
k = nx.katz_centrality_numpy(H,1/phi-0.01)
print(k)
draw(H, pos,k, 'Katz Centrality')
```

```
{u'6434236': 0.013648419454643715, u'4036449':
```

Five Five Values:
0.013648419454643715,
0.013648419454643715,
0.013648419454643715,
0.013648419454643715,
0.013648419454643715

PageRank Centrality:

A potential problem with Katz centrality is the following: if a node with high centrality links many others then all those others get high centrality. In many cases, however, it means less if a node is only one among many to be linked.



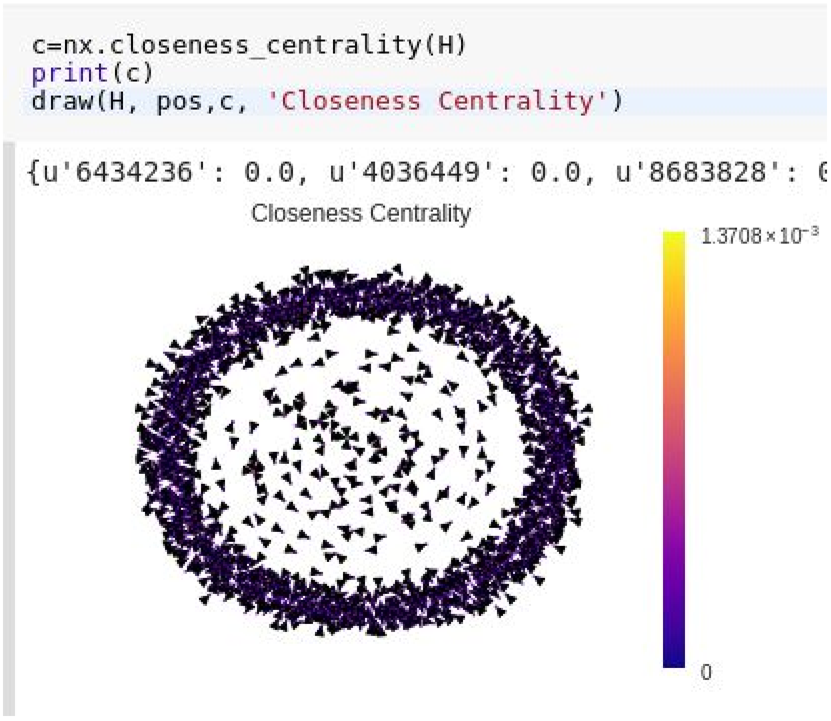
Page Rank

```
[ ]  ps=nx.pagerank(H,alpha=0.9)
     draw(H, pos,ps, 'Page Rank')
```

```
0.00023472888061959758,
0.00023472888061959758,
0.00023472888061959758,
0.00023472888061959758,
0.00023472888061959758
```

Closeness Centrality:

It is a measure of centrality in a network, calculated as the reciprocal of the sum of the length of the shortest paths between the node and all other nodes in the graph.



```
c=nx.closeness_centrality(H)
print(c)
draw(H, pos,c, 'Closeness Centrality')
```

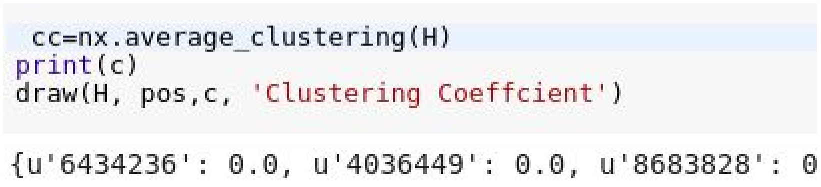
```
{u'6434236': 0.0, u'4036449': 0.0, u'8683828': 0
```

First five Values are:

0.0
0.0
0.0
0.0018248175182481751
0.0024330900243309

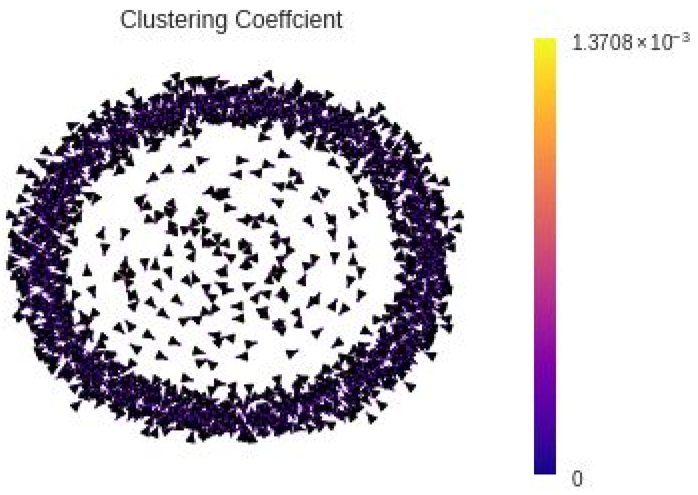
Clustering Coefficient

Global:



```
 cc=nx.average_clustering(H)
print(c)
draw(H, pos,c, 'Clustering Coeffcient')
```

```
{u'6434236': 0.0, u'4036449': 0.0, u'8683828': 0
```



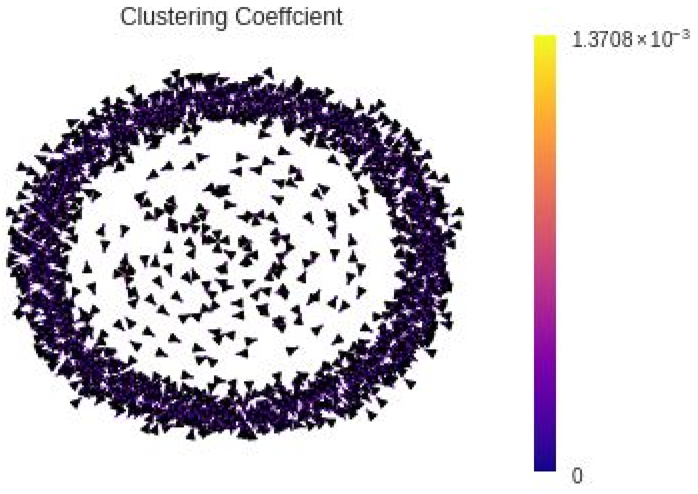
First five Values are:

0.0,
0.0,
0.0,
0.0018248175182481751
0.0024330900243309

Local:

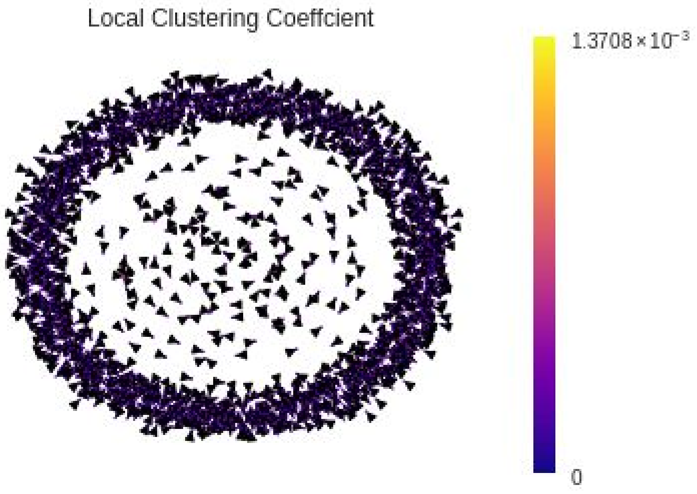
```
 cc=nx.average_clustering(H)
print(c)
draw(H, pos,c, 'Clustering Coeffcient')
```

```
{u'6434236': 0.0, u'4036449': 0.0, u'8683828': 0
```



```
cc=nx.clustering(H)
print(c)
draw(H, pos,c, 'Local Clustering Coeffcient')
```

```
{u'6434236': 0.0, u'4036449': 0.0, u'8683828': 0
```

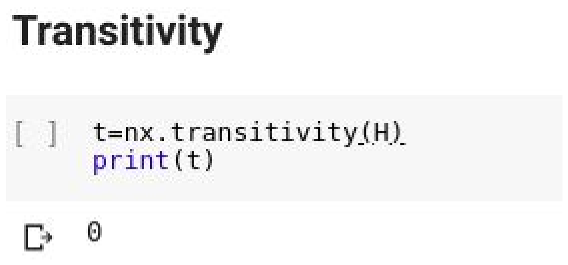


First five Values are:

0.0,
0.0,
0.0,
0.0018248175182481751
0.0024330900243309

Transitivity

In transitivity, we analyze the linking behavior to determine whether it demonstrates a transitive behaviour



Transitivity

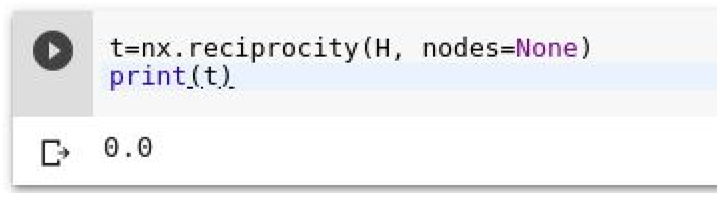
```
t=nx.transitivity(H)
print(t)
```

```
0
```

Value of transitivity is 0

Reciprocity

It is a measure of the likelihood of vertices in a directed network to be mutually linked. Like the clustering coefficient, scale-free degree distribution, or community structure, reciprocity is a quantitative measure used to study complex networks.



```
t=nx.reciprocity(H, nodes=None)
print(t)
```

```
0.0
```

<u>Best Centrality Measure for Dataset #2:</u>

Out of all the centrality measures, **Degree Centrality is the best measure to measure the centrality of Twitter**. This is because Degree Centrality is used for finding very connected individuals, popular individuals, individuals who are likely to hold most information or individuals who can quickly connect with the wider network.

SECOND ROUND

Q1. Try to get an algorithm package in Python to find the maximum connected component (called as a giant component in the class) in a given graph G. Let us denote the number of nodes in the giant component of a graph G as NG.

```
Gc = max(nx.connected_component_subgraphs(G), key=len)
nx.draw(Gc, with_labels=True)
```

Here, Gc is the largest connected component of graph G.

Q2. For the problem given in Homework - 5, vary ⟨k⟩ from 0 to 5 with an increment of 0.1. For each value of ⟨k⟩ find the ratio NGN where N is the number of nodes in the graph. Plot this ratio with respect to ⟨k⟩. Take ⟨k⟩ as x-axis and ratio NG/N as the y-axis.

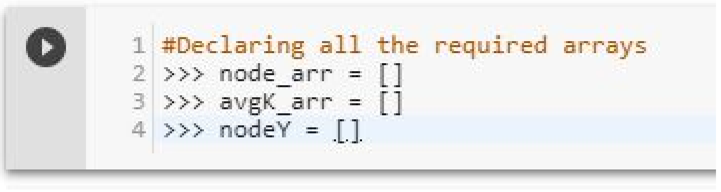
Firstly, we imported the required Python Dependencies: networkx and matplotlib.pyplot

```
#importing the networkx library
>>> import networkx as nx

#importing the matplotlib library for plotting the graph
>>> import matplotlib.pyplot as plt
```

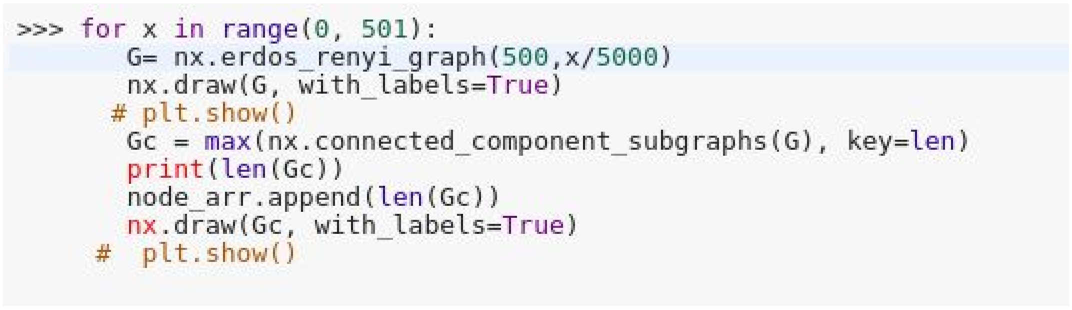
Then, we declared all the required arrays.

- node_arr: stores the number of nodes in the maximum connected components of all the arrays.
- avgK_arr: stores the values of average degrees from 0 to 5, with an increment of 0.1
- nodeY: stores the value of Ng/n.



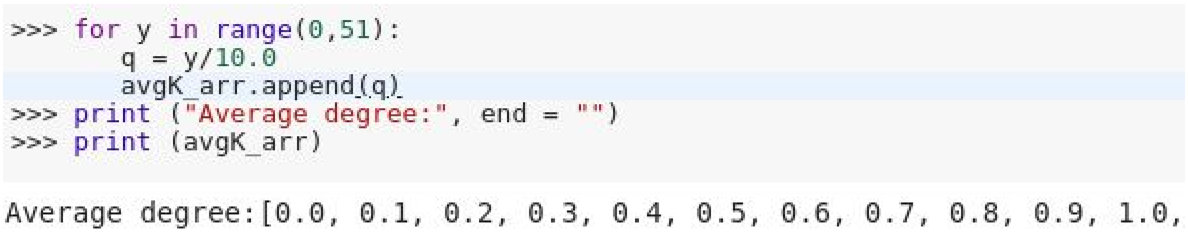
```
#Declaring all the required arrays
>>> node_arr = []
>>> avgK_arr = []
>>> nodeY = []
```

Then we plotted the graphs for all values of k between 0 to 5, incremented with 0.1 using the **erdos_renyi_graph function** of Networkx library and found the maximum connected components of each graph (denoted by Gc).



```
>>> for x in range(0, 501):
        G= nx.erdos_renyi_graph(500,x/5000)
        nx.draw(G, with_labels=True)
       # plt.show()
        Gc = max(nx.connected_component_subgraphs(G), key=len)
        print(len(Gc))
        node_arr.append(len(Gc))
        nx.draw(Gc, with_labels=True)
      #  plt.show()
```

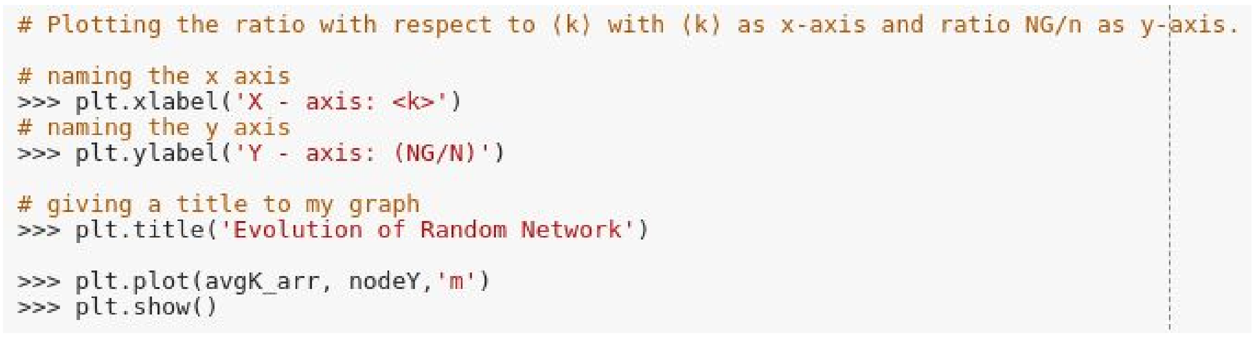
After that, we stored average degree values from 0 to 5, with an increment of 0.1 and printed them to check if they were correct.



```
>>> for y in range(0,51):
        q = y/10.0
        avgK_arr.append(q)
>>> print ("Average degree:", end = "")
>>> print (avgK_arr)
```

```
Average degree:[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0,
```

Finally, we plotted the graph with Average degree k on the X-axis and the value of Ng/N on the Y-axis. Also, given the title of the plot as ‘Evolution of Random Network'.



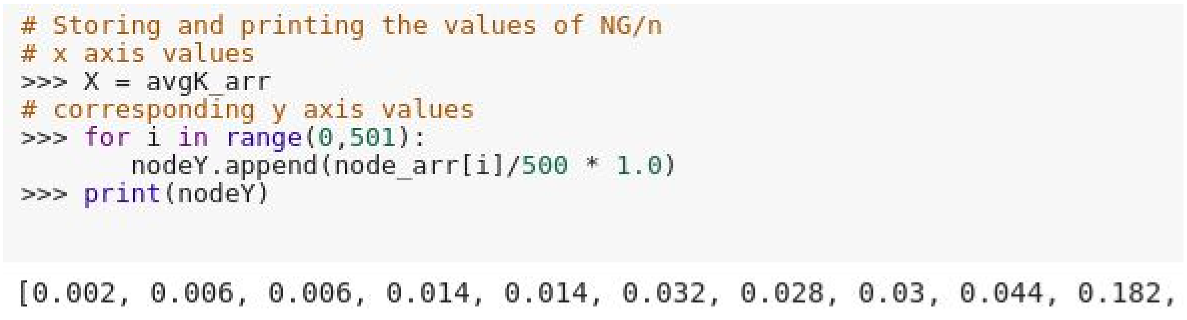
```
# Plotting the ratio with respect to ⟨k⟩ with ⟨k⟩ as x-axis and ratio NG/n as y-axis.

# naming the x axis
>>> plt.xlabel('X - axis: <k>')
# naming the y axis
>>> plt.ylabel('Y - axis: (NG/N)')

# giving a title to my graph
>>> plt.title('Evolution of Random Network')

>>> plt.plot(avgK_arr, nodeY,'m')
>>> plt.show()
```

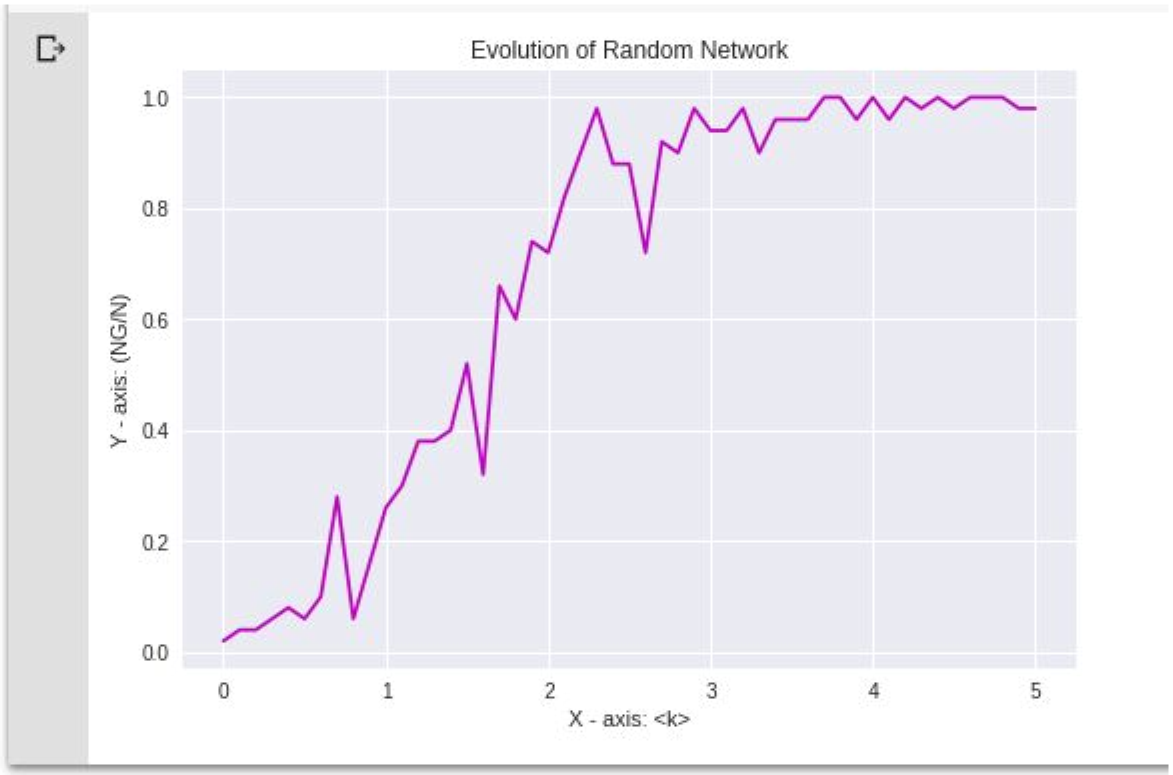
We calculate the value of Ng/N here.



```
# Storing and printing the values of NG/n
# x axis values
>>> X = avgK_arr
# corresponding y axis values
>>> for i in range(0,501):
        nodeY.append(node_arr[i]/500 * 1.0)
>>> print(nodeY)
```

```
[0.002, 0.006, 0.006, 0.014, 0.014, 0.032, 0.028, 0.03, 0.044, 0.182,
```

Here is the final plot obtained.



THIRD ROUND

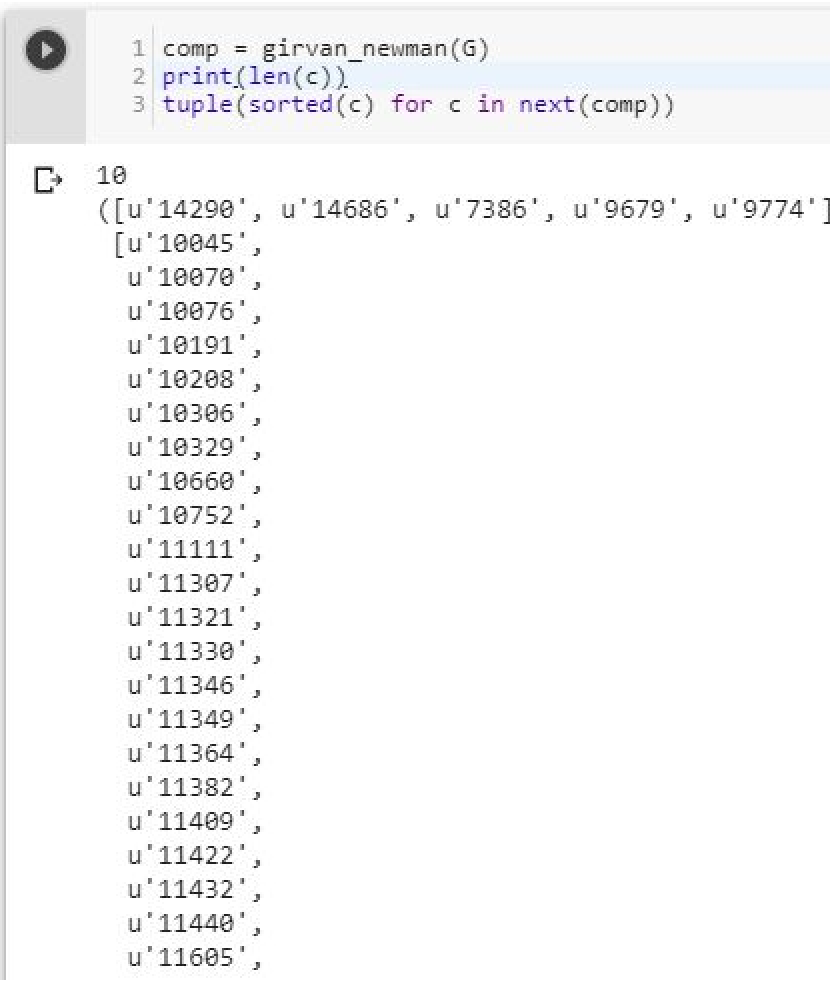
Q. Applying community detection algorithms on the original network formed using your original dataset.

In this project, we have used-

Group-Based Community Detection

1. Hierarchical Community Detection

a. **Girvan-Newman Algorithm-** Divisive algorithms require a centrality measure that is high for nodes that belong to different communities and is low for node pairs in the same community. The divisive hierarchical algorithm of Girvan and Newman uses link betweenness and centrality.



```
comp = girvan_newman(G)
print(len(c))
tuple(sorted(c) for c in next(comp))
```

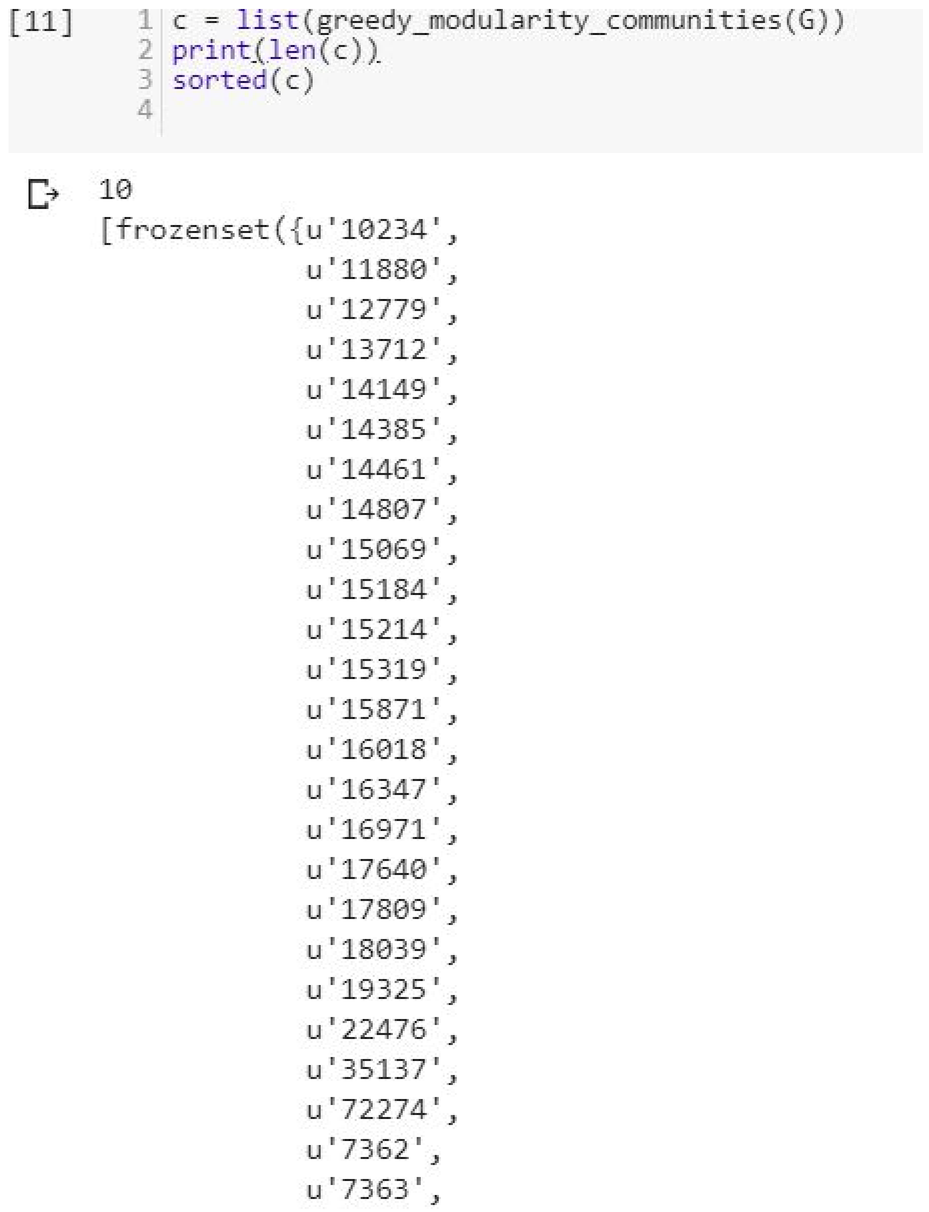
```
10
([u'14290', u'14686', u'7386', u'9679', u'9774']
 [u'10045',
  u'10070',
  u'10076',
  u'10191',
  u'10208',
  u'10306',
  u'10329',
  u'10660',
  u'10752',
  u'11111',
  u'11307',
  u'11321',
  u'11330',
  u'11346',
  u'11349',
  u'11364',
  u'11382',
  u'11409',
  u'11422',
  u'11432',
  u'11440',
  u'11605',
```

The Girvan Newman Algorithm is giving 10 communities.

2. Modularity Maximization

a. **Greedy Modularity Algorithm-** Find communities in the graph using Clauset-Newman-Moore greedy modularity maximization. This method currently supports the Graph class and does not consider edge weights.

Greedy modularity maximization begins with each node in its own community and joins the pair of communities that increases modularity most until no such pair exists**.**



```
[11]
c = list(greedy_modularity_communities(G))
print(len(c))
sorted(c)
```

```
10
[frozenset({u'10234',
            u'11880',
            u'12779',
            u'13712',
            u'14149',
            u'14385',
            u'14461',
            u'14807',
            u'15069',
            u'15184',
            u'15214',
            u'15319',
            u'15871',
            u'16018',
            u'16347',
            u'16971',
            u'17640',
            u'17809',
            u'18039',
            u'19325',
            u'22476',
            u'35137',
            u'72274',
            u'7362',
            u'7363',
```

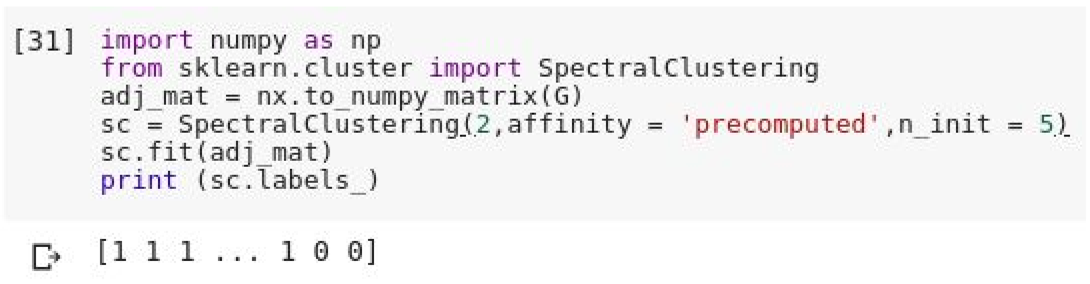
Modularity Maximization community detection is giving 10 community.

3. Balanced Based Detection-

a. **Spectral Clustering Algorithm-** Spectral clustering algorithms for community detection under a general bipartite stochastic block model (SBM). A modern spectral clustering algorithm consists of three steps:

- Regularization of an appropriate adjacency or Laplacian matrix
- A form of spectral truncation
- A k-means type algorithm in the reduced spectral domain.

We focus on the adjacency-based spectral clustering and for the first step, propose a new data-driven regularization that can restore the concentration of the adjacency matrix even for the sparse networks. This result is based on recent work on regularization of random binary matrices but avoids using unknown population level parameters, and instead estimates the necessary quantities from the data.



```
[31] import numpy as np
     from sklearn.cluster import SpectralClustering
     adj_mat = nx.to_numpy_matrix(G)
     sc = SpectralClustering(2,affinity = 'precomputed',n_init = 5)
     sc.fit(adj_mat)
     print (sc.labels_)
```

```
[1 1 1 ... 1 0 0]
```

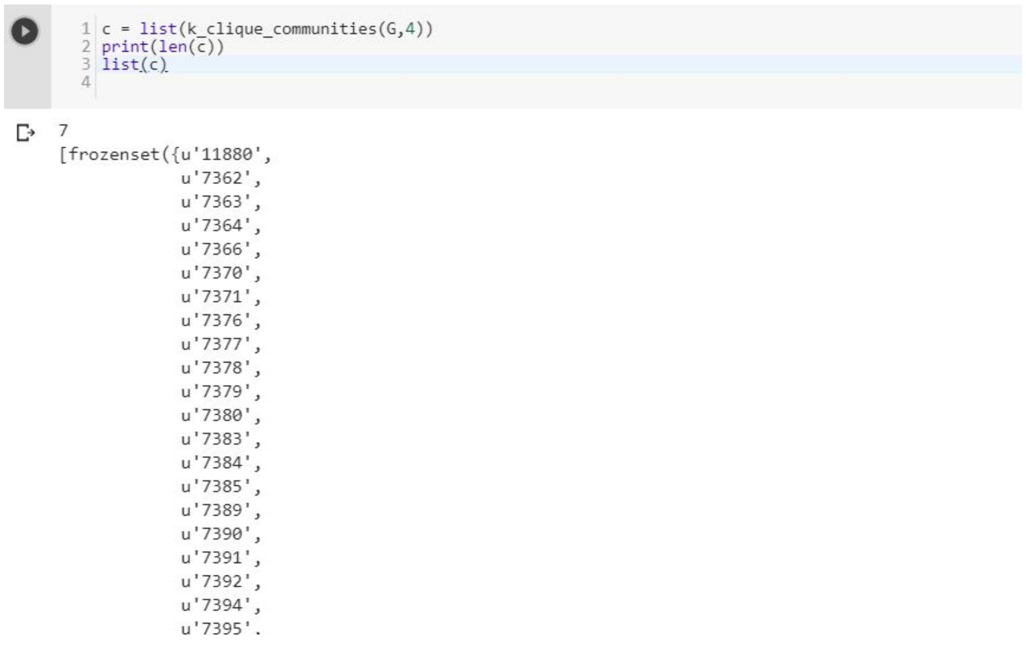
0 belongs to one community and all the others belong to other communities.

The Spectral Clustering Method is giving 10 communities.

Member Based Community Detection

4. Node Degree

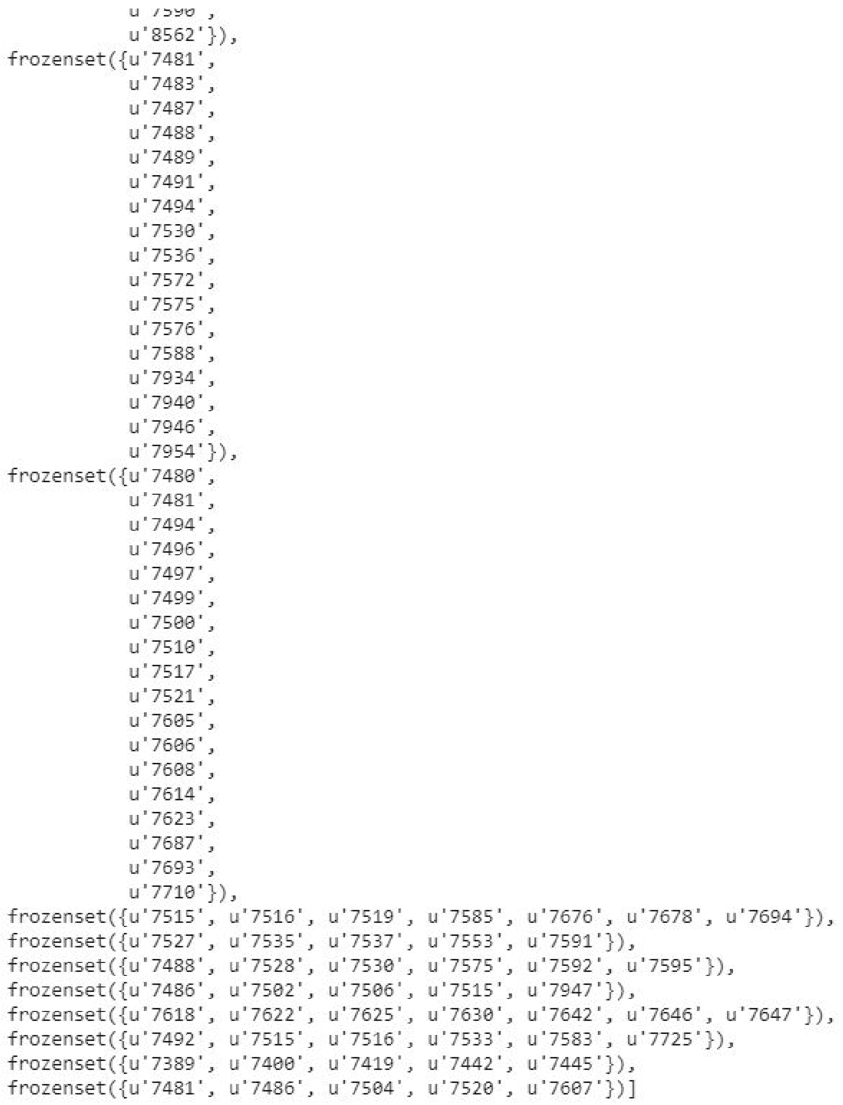
- **Clique Percolation Algorithm-** We assume communities are formed from a set of cliques (small or large) in addition to edges that connect these cliques. A Clique well-known algorithm in this area is the clique percolation method (CPM). Given parameter k, the method starts by finding all cliques of size k. Then a graph is generated (clique graph) where all cliques are represented as nodes, and cliques that share k - 1 vertex are connected via edges. Communities are then found by reporting the connected components of this graph. The algorithm searches for all cliques of size k and is therefore computationally intensive. In practice, when using the CP Malgorithm, we often solve CPM for a small k. Relaxations discussed for cliques are desirable to enable the algorithm to perform faster. Lastly, CPM can return to overlapping communities.



```
c = list(k_clique_communities(G,4))
print(len(c))
list(c)

```

```
7
[frozenset({u'11880',
            u'7362',
            u'7363',
            u'7364',
            u'7366',
            u'7370',
            u'7371',
            u'7376',
            u'7377',
            u'7378',
            u'7379',
            u'7380',
            u'7383',
            u'7384',
            u'7385',
            u'7389',
            u'7390',
            u'7391',
            u'7392',
            u'7394',
            u'7395'.
```



```
           u'7590',
           u'8562'}),
 frozenset({u'7481',
           u'7483',
           u'7487',
           u'7488',
           u'7489',
           u'7491',
           u'7494',
           u'7530',
           u'7536',
           u'7572',
           u'7575',
           u'7576',
           u'7588',
           u'7934',
           u'7940',
           u'7946',
           u'7954'}),
 frozenset({u'7480',
           u'7481',
           u'7494',
           u'7496',
           u'7497',
           u'7499',
           u'7500',
           u'7510',
           u'7517',
           u'7521',
           u'7605',
           u'7606',
           u'7608',
           u'7614',
           u'7623',
           u'7687',
           u'7693',
           u'7710'}),
 frozenset({u'7515', u'7516', u'7519', u'7585', u'7676', u'7678', u'7694'}),
 frozenset({u'7527', u'7535', u'7537', u'7553', u'7591'}),
 frozenset({u'7488', u'7528', u'7530', u'7575', u'7592', u'7595'}),
 frozenset({u'7486', u'7502', u'7506', u'7515', u'7947'}),
 frozenset({u'7618', u'7622', u'7625', u'7630', u'7642', u'7646', u'7647'}),
 frozenset({u'7492', u'7515', u'7516', u'7533', u'7583', u'7725'}),
 frozenset({u'7389', u'7400', u'7419', u'7442', u'7445'}),
 frozenset({u'7481', u'7486', u'7504', u'7520', u'7607'})]
```

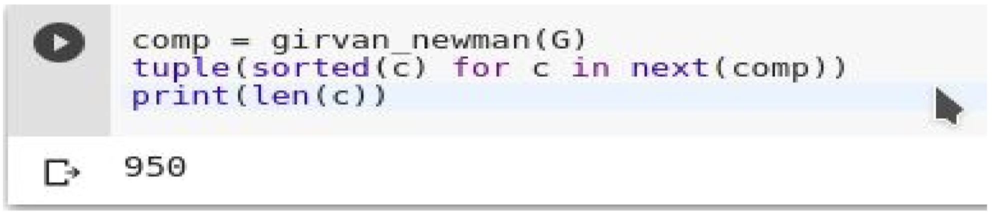
The k clique CPM is giving 12 communities.

Inference:

We have found from the clustering algorithms that the average communities are 11. We have used K as 5 in CPM. The output shows the list of nodes.

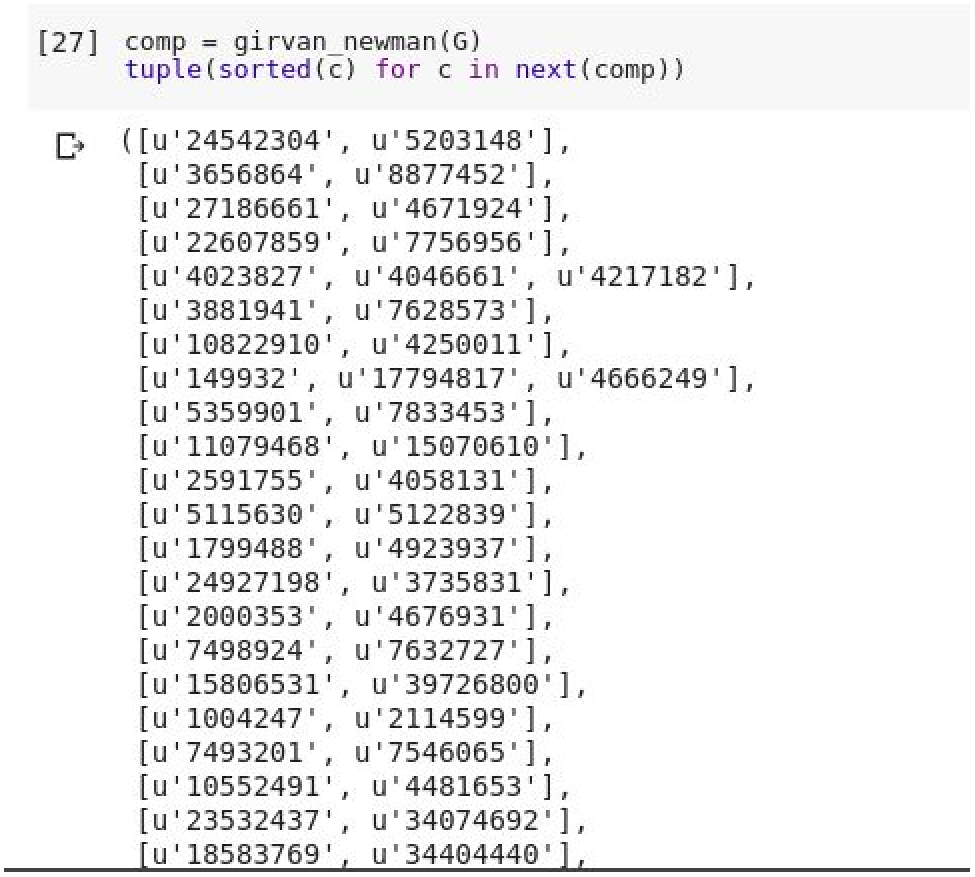
Second Data set Community Detection

- **Girvan- Newman**



```
comp = girvan_newman(G)
tuple(sorted(c) for c in next(comp))
print(len(c))
```

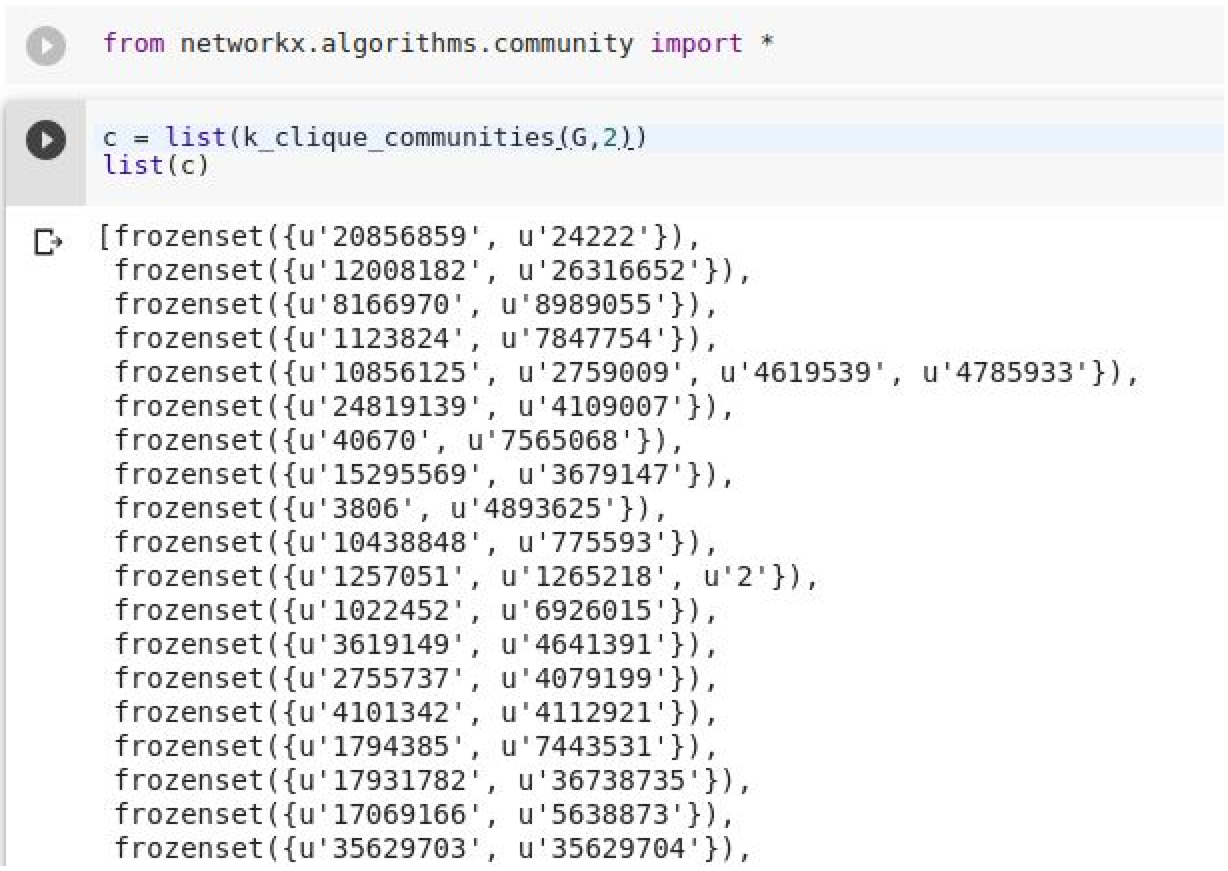
```
950
```



```
[27] comp = girvan_newman(G)
     tuple(sorted(c) for c in next(comp))
```

```
([u'24542304', u'5203148'],
 [u'3656864', u'8877452'],
 [u'27186661', u'4671924'],
 [u'22607859', u'7756956'],
 [u'4023827', u'4046661', u'4217182'],
 [u'3881941', u'7628573'],
 [u'10822910', u'4250011'],
 [u'149932', u'17794817', u'4666249'],
 [u'5359901', u'7833453'],
 [u'11079468', u'15070610'],
 [u'2591755', u'4058131'],
 [u'5115630', u'5122839'],
 [u'1799488', u'4923937'],
 [u'24927198', u'3735831'],
 [u'2000353', u'4676931'],
 [u'7498924', u'7632727'],
 [u'15806531', u'39726800'],
 [u'1004247', u'2114599'],
 [u'7493201', u'7546065'],
 [u'10552491', u'4481653'],
 [u'23532437', u'34074692'],
 [u'18583769', u'34404440'],
```

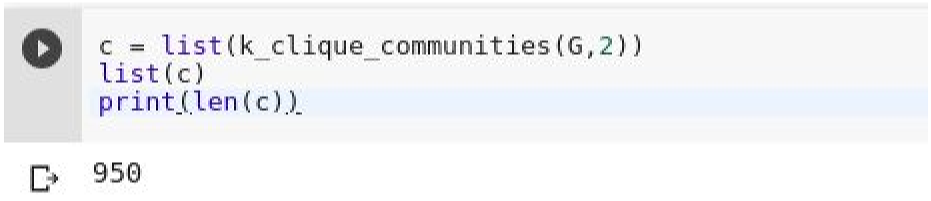
● K- Clique Algorithm



```
from networkx.algorithms.community import *
```

```
c = list(k_clique_communities(G,2))
list(c)
```

```
[frozenset({u'20856859', u'24222'}),
 frozenset({u'12008182', u'26316652'}),
 frozenset({u'8166970', u'8989055'}),
 frozenset({u'1123824', u'7847754'}),
 frozenset({u'10856125', u'2759009', u'4619539', u'4785933'}),
 frozenset({u'24819139', u'4109007'}),
 frozenset({u'40670', u'7565068'}),
 frozenset({u'15295569', u'3679147'}),
 frozenset({u'3806', u'4893625'}),
 frozenset({u'10438848', u'775593'}),
 frozenset({u'1257051', u'1265218', u'2'}),
 frozenset({u'1022452', u'6926015'}),
 frozenset({u'3619149', u'4641391'}),
 frozenset({u'2755737', u'4079199'}),
 frozenset({u'4101342', u'4112921'}),
 frozenset({u'1794385', u'7443531'}),
 frozenset({u'17931782', u'36738735'}),
 frozenset({u'17069166', u'5638873'}),
 frozenset({u'35629703', u'35629704'}),
```



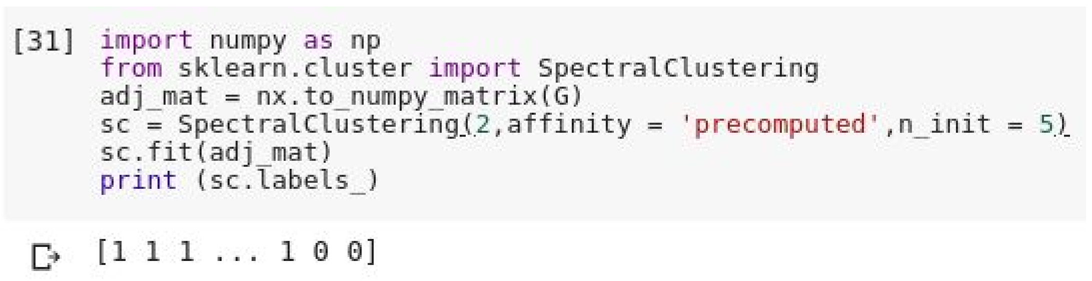
```
c = list(k_clique_communities(G,2))
list(c)
print(len(c))
```

```
950
```

We have chosen the value of k as 2 because we have randomly selected the nodes. The community are of size 2.

● Spectral Clustering

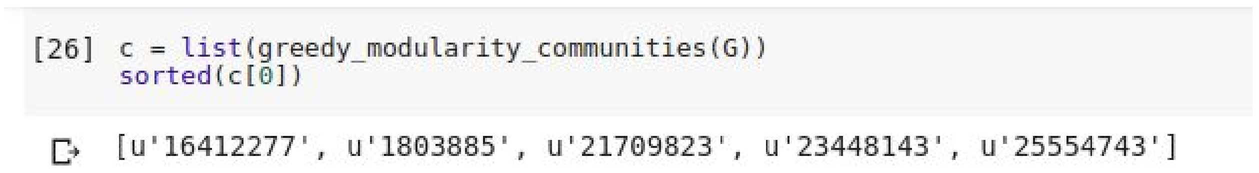
0 belongs to one community and all the others belong to other communities.



```
[31] import numpy as np
     from sklearn.cluster import SpectralClustering
     adj_mat = nx.to_numpy_matrix(G)
     sc = SpectralClustering(2,affinity = 'precomputed',n_init = 5)
     sc.fit(adj_mat)
     print (sc.labels_)
```

```
[1 1 1 ... 1 0 0]
```

- **Greedy Modularity**



```
[26] c = list(greedy_modularity_communities(G))
     sorted(c[0])
```

```
[u'16412277', u'1803885', u'21709823', u'23448143', u'25554743']
```

Inference:
The communities are mostly of size 2 because the graph developed is spare are only two nodes are there.