Social Network Analysis

Project Round 1

Team: FAT LITTLE BOYS

Members:	Khalid Mehtab Khan	17ucs078
	Abhi Mahajan	17ucs004
	Navdeep Singh	15ucs081

Date: 9th March 2020
Submitted to: Dr Sakthi Balan

Data Set 1: High-energy physics citation network

Description: Covers all the citations within a dataset of 34,546 papers with 421,578 edges. If a paper *i* cites paper *j*, the graph contains a directed edge from *i* to *j*.

Count:

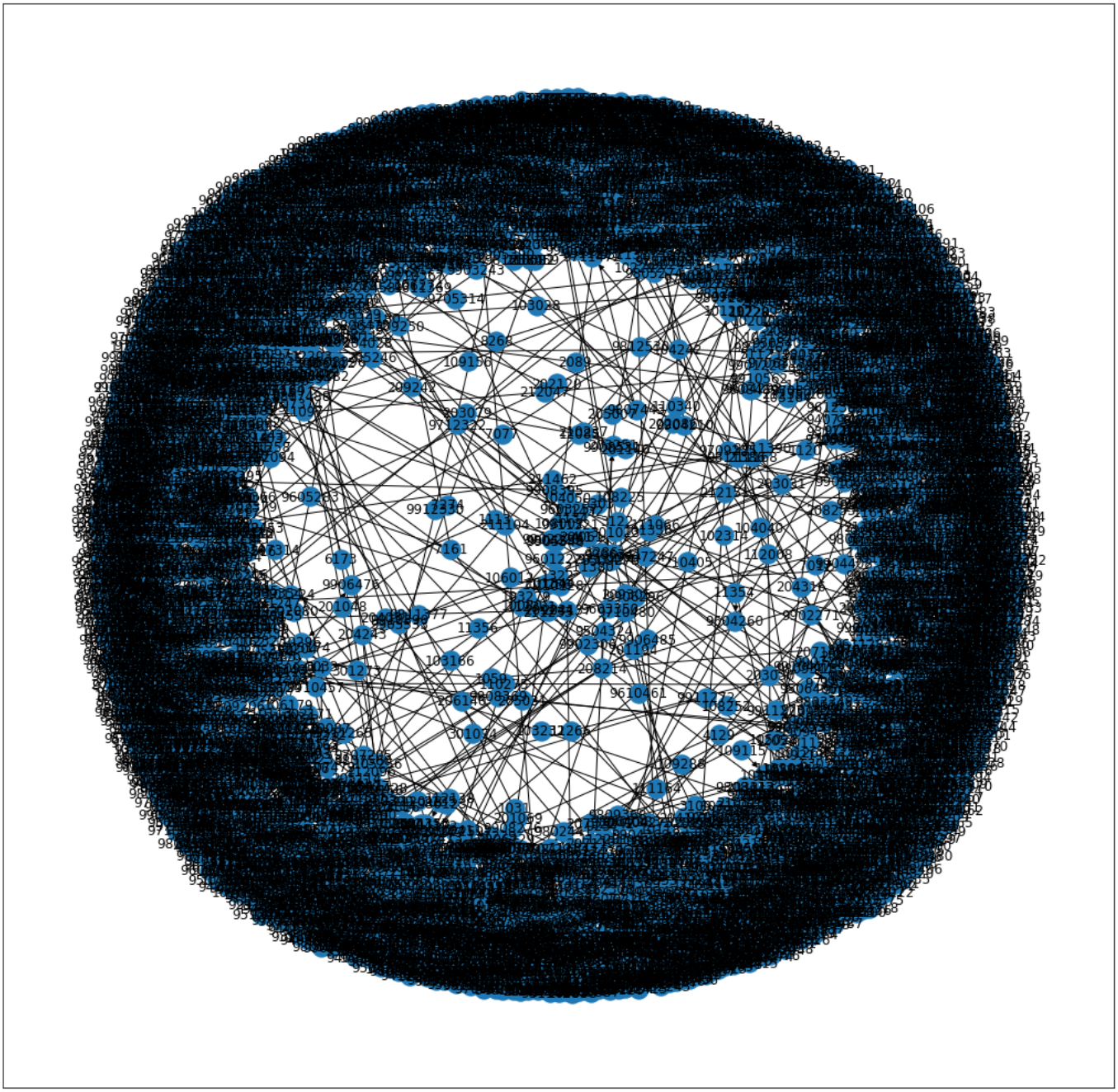
Nodes	34546
Edges	421578

The data contains a huge number of nodes and edges, to plot a graph we will use a small sample of the data say around 3000 of the edges.

```
import pandas as pd
import random
import numpy as np
lis = [random.randint( 0 , 421578 ) for i in range( 0 , 3001)]
pred = lambda x: x not in lis
df=pd.read_csv( "Cit-HepPh.txt" , skiprows=pred, index_col= None , sep= ""
,delimiter= '\t', header = None)
print(df.describe())
np.savetxt('sample-Cit-HepPh.txt' , df.values, fmt= '%d' )
```

```
import networkx as nx
from operator import itemgetter
import matplotlib.pyplot as plt
%matplotlib inline
graph=nx.read_edgelist('sample-Cit-HepPh.txt' ,create_using=nx.DiGraph())
plt.figure(figsize = ( 18 , 18 ))
nx.draw_networkx(graph)
plt.show()
```

Graph: (Small Sample || Size : 18 , 18)



Calculating Transitivity, reciprocity and Clustering Coefficients:

```
import matplotlib.pyplot as plt
import networkx as nx
graph=nx.read_edgelist('Cit-HepPh.txt' ,create_using=nx.DiGraph(),nodetype=int)

print("Transitivity T:")
print(nx.transitivity(graph))
print("Reciprocity R:")
print(nx.reciprocity(graph))
print("Global Clustering Coefficient GCC:")
print(nx.average_clustering(graph))
print("Local Clustering Coefficient of node 103095:")
print(nx.clustering(graph, 103095))
print("Local Clustering Coefficient of node 9902238:")
print(nx.clustering(graph, 9902238))
print("Local Clustering Coefficient of node 9911459:")
print(nx.clustering(graph, 9911459))
```

Transitivity **T**	**0.10118914905026928**
Reciprocity **R**	**0.00311686093676615**
Global Clustering Coefficient **GCC**	**0.14326383421252617**
Local Clustering Coefficient of node **“103095”**	**0.042214912280701754**
Local Clustering Coefficient of node **“9902238”**	**0.152014652014652**
Local Clustering Coefficient of node **“9911459”**	**0.06711344211344211**

As the number of nodes in the graph is still too large we will calculate the local clustering coefficient for some random particular nodes, say 103095, 9902238, 9911459.

Centrality Measures:

Degree Centrality:

```
dc = nx.degree_centrality(graph)
sorted_degree = sorted(dc.items(), key=itemgetter(1),
reverse=True)
print("Top 10 nodes by degree:")
for dc in sorted_degree[:10]:
    print(dc)
```

As the number of nodes is too large we produce the nodes with the top 10 nodes of degree centrality.

Top 10 nodes by degree:

Node	Degree Centrality
9803315	0.024489795918367346
9512380	0.018381820813431756
9804398	0.01800550007236937
9407339	0.01644232160949486
9606399	0.016181791865682443
9807344	0.014763352149370386
9306320	0.013865971920683166
201071	0.013547546678245766
9905221	0.013460703430308293
9507378	0.013200173686495875

As the number of nodes is too large we produce the average, maximum and minimum degree centralities for drawing conclusions.

```
dc = nx.degree_centrality(graph)
avgdc = 0
for i in dc.values():
  avgdc += i

print(avgdc/len(dc.values()))
```

```
maxdc = 0
for i in dc.values():
  if i>maxdc:
    maxdc = i
print(maxdc)
```

```
mindc=1
for i in dc.values():
  if i<mindc:
    mindc = i
print(mindc)
```

Average Degree Centrality	0.000706520827862103
Maximum Degree Centrality	0.024489795918367346
Minimum Degree Centrality	2.8947749312490954e-05

Eigenvector Centrality:

As the number of nodes is too large we produce the nodes with the top 10 nodes of Eigenvector centrality.

```
ec = nx.eigenvector_centrality(graph,max_iter=100)
sorted_eigenvector = sorted(ec.items(),
key=itemgetter(1), reverse=True)
print("Top 10 nodes by eigenvector centrality:")
for d in sorted_eigenvector[:10]:
    print(d)
```

Top 10 nodes by eigenvector centrality:

Node	Eigenvector Centrality
9207214	0.2600157979467914
9304225	0.19710824365473087
9210235	0.17116937774922744
9408384	0.16519423183836565
9303230	0.16468877208835062
9209232	0.16430097599703256
9308246	0.16284377743199807
9209268	0.15722150237945953
9304307	0.15263179345713085
9307247	0.14025757168769196

As the number of nodes is too large we produce the average, maximum and minimum Eigenvector centralities for drawing conclusions.

```
ec = nx.eigenvector_centrality(graph,max_iter=100)
avgec = 0
for i in ec.values():
  avgec += i
print(avgec/len(ec.values()))
```

```
maxec = 0
for i in ec.values():
  if i>maxec:
    maxec = i
print(maxec)
```

```
minec=1
for i in ec.values():
  if i<minec:
    minec = i
print(minec)
```

Average Eigenvector Centrality	0.0006434783806018568
Maximum Eigenvector Centrality	0.2600157979467914
Minimum Eigenvector Centrality	4.093736001931497e-46

Betweenness Centrality:

As the number of nodes is too large we produce the nodes with the top 10 nodes of Betweenness centrality.

```
bc = nx.betweenness_centrality(graph,k=100)
sorted_betweenness = sorted(bc.items(), key=itemgetter(1),
reverse=True)
print("Top 10 nodes by betweenness centrality:")
for d in sorted_betweenness[:10]:
    print(d)
```

Top 10 nodes by betweenness centrality:

Node	Betweenness Centrality
206003	0.19181490920946823
101224	0.13751163502663252
9501384	0.09189676611272837
9806301	0.07985286185041306
9806471	0.0702690242863032
103230	0.06920937551072788
9809468	0.0626549538219483
9901409	0.0626337070779338
9910201	0.06262216032172785
9904408	0.06100209353159604

As the number of nodes is too large we produce the average, maximum and minimum Betweenness centralities for drawing conclusions.

```
bc = nx.betweenness_centrality(graph,k=100)
avgbc = 0
for i in bc.values():
  avgbc += i

print(avgbc/len(bc.values()))
```

```
maxbc = 0
for i in bc.values():
  if i>maxbc:
    maxbc = i
print(maxbc)
```

```
minbc=1
for i in bc.values():
  if i<minbc:
    minbc = i
print(minbc)
```

Average Betweenness Centrality	0.00012294678300202777
Maximum Betweenness Centrality	0.19181490920946823
Minimum Betweenness Centrality	0.0

Katz Centrality:

As the number of nodes is too large we produce the nodes with the top 10 nodes of Katz Centrality.

```
kc = nx.katz_centrality(graph)
sorted_katz = sorted(kc.items(), key=itemgetter(1), reverse=True)
print("Top 10 nodes by katz centrality:")
for d in sorted_katz[:10]:
    print(d)
```

Top 10 nodes by katz centrality:

Nodes	Katz Centrality
9207214	0.18619443073520525
9408384	0.15425223251554218
9304225	0.1540721995273013
9304307	0.1362359292936491
9303230	0.12441500804990983
9209232	0.12347558612395348
9308246	0.12008537979164959
9507378	0.1153110608449669
9210235	0.11476234367962208
9311214	0.11361963810536199

As the number of nodes is too large we produce the average, maximum and minimum Katz centralities for drawing conclusions.

```
kc = nx.katz_centrality(graph)
avgkc = 0
for i in kc.values():
  avgkc += i

print(avgkc/len(kc.values()))
```

```
maxkc = 0
for i in kc.values():
  if i>maxkc:
    maxkc = i
print(maxkc)
```

```
minkc=1
for i in kc.values():
  if i<minkc:
    minkc = i
print(minkc)
```

Average Katz Centrality	0.0010975886740570092
Maximum Katz Centrality	0.18619443073520525
Minimum Katz Centrality	2.8408266640934843e-05

Page Rank:

As the number of nodes is too large we produce the nodes with the top 10 nodes of Page Rank.

```
pr = nx.pagerank(graph)
sorted_rank = sorted(pr.items(), key=itemgetter(1), reverse=True)
print("Top 10 nodes by page rank:")
for d in sorted_rank[:10]:
    print(d)
```

Top 10 nodes by page rank:

Node	Page Rank
9303255	0.0036552156226205505
9209205	0.0027529807126590354
9310316	0.0025038906410070165
9206203	0.00225672820051627
9208254	0.002166528431306827
9803315	0.001835009054615175
9203203	0.0017550673210533234
9206242	0.0017540341319184287
9303230	0.0016660211441866565
9206236	0.001615453710224979

As the number of nodes is too large we produce the average, maximum and minimum Page Rank for drawing conclusions.

```
pr = nx.pagerank(graph)
avgpr = 0
for i in pr.values():
  avgpr += i
print(avgpr/len(pr.values()))
```

```
maxpr = 0
for i in pr.values():
  if i>maxpr:
    maxpr = i
print(maxpr)
```

```
minpr = 1
for i in pr.values():
  if i<minpr:
    minpr = i
print(minpr)
```

Average Page Rank	2.894691136456476e-05
Maximum Page Rank	0.0036552156226205505
Minimum Page Rank	9.42519496134375e-06

Closeness Centrality:

```
cc = nx.closeness_centrality(graph,u=9803315)
print(cc)
```

0.08266499724516543

Inferences:

- Transitivity of the graph and the Global clustering Coefficients have large values as compared to other values.
- In a star with vertices, the betweenness centrality of the central vertex is, therefore, the number of such geodesics which is. The betweenness centrality of each pendant vertex is zero since no pendant vertex lies in between any geodesic. As we can see the minimum betweenness centrality is zero.
- The Maximum Katz centrality of the node attains a moderate value as compared to maximum values of other centralities
- One more important centrality is Page rank as it attains low value as a maximum when compared to other centralities, also the average Page Rank value is low.
- Among all centralities, Eigenvector Centrality attains the highest value for the maximum value of a node.

Conclusion:

The graph is highly populated and not sparse as the transitivity value of the graph is quite high. We see that some nodes have a rather high centrality, while most have very low, indicating that some nodes are quite popular while most others are barely connected.

Data Set 2: Social Network: Reddit Hyperlink Network

Description: The hyperlink network represents the directed connections between two subreddits. We also provide subreddit embeddings. The network is extracted from publicly available Reddit data of 2.5 years from Jan 2014 to April 2017.

Subreddit Hyperlink Network: the subreddit-to-subreddit hyperlink network is extracted from the posts that create hyperlinks from one subreddit to another. We say a hyperlink originates from a post in the source community and links to a post in the target community. Each hyperlink is annotated with three properties: the timestamp, the sentiment of the source community post towards the target community post, and the text property vector of the source post. The network is directed, signed, temporal, and attributed.

Count:

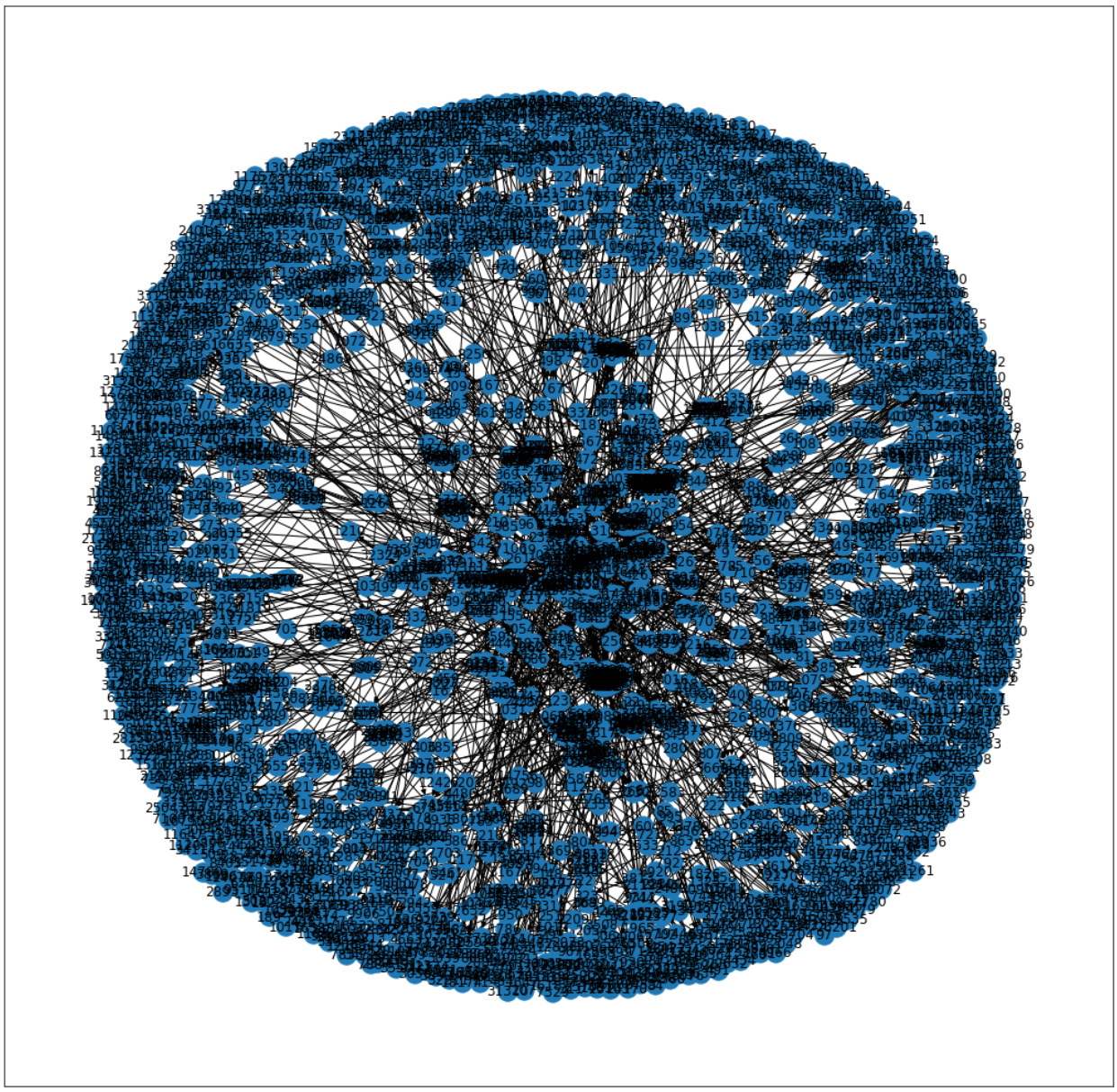
Number of nodes (subreddits)	55,863
Number of edges (hyperlink between subreddits)	858,490

The data contains a huge number of nodes and edges, to plot a graph we will use a small sample of the data say around 3000 of the edges.

```
import pandas as pd
import random
import numpy as np
lis = [random.randint( 0 , 286561 ) for i in range( 0 , 3001)]
pred = lambda x: x not in lis
df=pd.read_csv( "redditHyperlinks.txt" , skiprows=pred, index_col= None ,
sep= "" ,delimiter= ' ', header = None)
print(df.describe())
np.savetxt('sample-redditHyperlinks.txt' , df.values, fmt= '%d' )
```

```
import networkx as nx
from operator import itemgetter
import matplotlib.pyplot as plt
%matplotlib inline
graph=nx.read_edgelist('sample-redditHyperlinks.txt'
,create_using=nx.DiGraph())
plt.figure(figsize = ( 18 , 18 ))
nx.draw_networkx(graph)
plt.show()
```

Graph: (Small Sample || Size : 18 , 18)



Calculating Transitivity, reciprocity and Clustering Coefficients:

```
import matplotlib.pyplot as plt
import networkx as nx
g=nx.read_edgelist('redditHyperlinks.txt' ,create_using=nx.DiGraph(),nodetype=int)

print("Transitivity T:")
print(nx.transitivity(g))
print("Reciprocity R:")
print(nx.reciprocity(g))
print("Global Clustering Coefficient GCC:")
print(nx.average_clustering(g))
print("Local Clustering Coefficient of node 28:")
print(nx.clustering(g, 28))
print("Local Clustering Coefficient of node 400:")
print(nx.clustering(g, 400))
print("Local Clustering Coefficient of node 612:")
print(nx.clustering(g, 612))
```

Transitivity T	0.05669671995230048
Reciprocity R	0.1957756800487589
Global Clustering Coefficient GCC	0.139306128836154
Local Clustering Coefficient of node 28	0.04090000382005017
Local Clustering Coefficient of node 400	0.11870967741935484
Local Clustering Coefficient of node 612	0.13253012048192772

As the number of nodes in the graph is still too large we will calculate the local clustering coefficient for some random particular nodes, say 103095, 9902238, 9911459.

Centrality Measures:

Degree Centrality:

```
dc = nx.degree_centrality(g)
sorted_degree = sorted(dc.items(), key=itemgetter(1), reverse=True)
print("Top 10 nodes by degree:")
for dc in sorted_degree[:10]:
    print(dc)
```

As the number of nodes is too large we produce the nodes with the top 10 nodes of degree centrality.

Top 10 nodes by degree:

Node	Degree Centrality
59	0.07055206149545772
41	0.056743535988819004
122	0.05042627533193571
57	0.031949685534591196
225	0.030719776380153736
36	0.026750524109014674
166	0.024654088050314465
0	0.02328441649196366
42	0.02280922431865828
224	0.022641509433962263

As the number of nodes is too large we produce the average, maximum and minimum degree centralities for drawing conclusions.

```
dc = nx.degree_centrality(g)
avgdc = 0
for i in dc.values():
  avgdc += i

print(avgdc/len(dc.values()))
```

```
maxdc = 0
for i in dc.values():
  if i>maxdc:
    maxdc = i
print(maxdc)
```

```
mindc=1
for i in dc.values():
  if i<mindc:
    mindc = i
print(mindc)
```

Average Degree Centrality	0.00021536442609267805
Maximum Degree Centrality	0.07055206149545772
Minimum Degree Centrality	2.795248078266946e-05

Eigenvector Centrality:

As the number of nodes is too large we produce the nodes with the top 10 nodes of Eigenvector centrality.

```
ec = nx.eigenvector_centrality(g,max_iter=100)
sorted_eigenvector = sorted(ec.items(), key=itemgetter(1), reverse=True)
print("Top 10 nodes by eigenvector centrality:")
for d in sorted_eigenvector[:10]:
    print(d)
```

Top 10 nodes by eigenvector centrality:

Node	Eigenvector Centrality
59	0.2651849714317706
41	0.25348488091361604
166	0.18016430348316215
36	0.17390465153200324
42	0.16648951718218455
190	0.14377204939239926
55	0.13870523888975217
97	0.1365494961612271
224	0.1350220379131744
157	0.12484434665059566

As the number of nodes is too large we produce the average, maximum and minimum Eigenvector centralities for drawing conclusions.

```
ec = nx.eigenvector_centrality(g,max_iter=100)
avgec = 0
for i in ec.values():
  avgec += i


print(avgec/len(ec.values()))
```

```
maxec = 0
for i in ec.values():
  if i>maxec:
    maxec = i
print(maxec)
```

```
minec=1
for i in ec.values():
  if i<minec:
    minec = i
print(minec)
```

Average Eigenvector Centrality	0.000928584237902804
Maximum Eigenvector Centrality	0.2651849714317706
Minimum Eigenvector Centrality	8.56907943307509e-15

Betweenness Centrality:

As the number of nodes is too large we produce the nodes with the top 10 nodes of Betweenness centrality.

```
bc = nx.betweenness_centrality(g,k=100)
sorted_betweenness = sorted(bc.items(), key=itemgetter(1), reverse=True)
print("Top 10 nodes by betweenness centrality:")
for d in sorted_betweenness[:10]:
    print(d)
```

Top 10 nodes by betweenness centrality:

Node	Betweenness Centrality
59	0.05963330883120977
41	0.053819185246275526
122	0.044764628656930316
225	0.022363662953319676
123	0.018303894190329413
0	0.015037689798577714
233	00.014690763065653728
57	0.013375229619827518
451	0.01316086200027518
369	0.01065676972374559

As the number of nodes is too large we produce the average, maximum and minimum Betweenness centralities for drawing conclusions.

```
bc = nx.betweenness_centrality(g,k=100)
avgbc = 0
for i in bc.values():
  avgbc += i

print(avgbc/len(bc.values()))
```

```
maxbc = 0
for i in bc.values():
  if i>maxbc:
    maxbc = i
print(maxbc)
```

```
minbc=1
for i in bc.values():
  if i<minbc:
    minbc = i
print(minbc)
```

Average Betweenness Centrality	3.708982592286775e-05
Maximum Betweenness Centrality	0.05963330883120977
Minimum Betweenness Centrality	0.0

Page Rank:

As the number of nodes is too large we produce the nodes with the top 10 nodes of Page Rank.

```
pr = nx.pagerank(g)
sorted_rank = sorted(pr.items(), key=itemgetter(1), reverse=True)
print("Top 10 nodes by page rank:")
for d in sorted_rank[:10]:
    print(d)
```

Top 10 nodes by page rank:

Node	Page Rank
59	0.0036552156226205505
41	0.0027529807126590354
36	0.0025038906410070165
166	0.00225672820051627
57	0.002166528431306827
7180	0.001835009054615175
0	0.0017550673210533234
42	0.0017540341319184287
225	0.0016660211441866565
55	0.001615453710224979

As the number of nodes is too large we produce the average, maximum and minimum Page Rank for drawing conclusions.

```
pr = nx.pagerank(g)
avgpr = 0
for i in pr.values():
  avgpr += i
print(avgpr/len(pr.values()))
```

```
maxpr = 0
for i in pr.values():
  if i>maxpr:
    maxpr = i
print(maxpr)
```

```
minpr = 1
for i in pr.values():
  if i<minpr:
    minpr = i
print(minpr)
```

Average Page Rank	2.795169946333658e-05
Maximum Page Rank	0.011211335014428519
Minimum Page Rank	7.591266448091285e-06

Closeness Centrality:

```
cc = nx.closeness_centrality(g,u=400)
print(cc)
```

0.22677002927118745

Inferences:

- Average Degree Centrality of the network is low.
- Reciprocity of the graph and the Global clustering Coefficients have large values as compared to other values while transitivity of the network is moderate.
- The minimum Betweenness Centrality of the network is zero
- One more important centrality is Page rank as it attains low value as a maximum when compared to other centralities, also the average Page Rank value is low.
- The Closeness Centrality is important for this network.
- Among all centralities, Eigenvector Centrality attains the highest value for the maximum value of a node.

Conclusion:

The graph is highly dense as the reciprocity value of the graph is quite high. We see that some nodes have a rather a high centrality, while most have very low, indicated that some nodes are quite popular while most others are barely connected

Visualization of Number of nodes in Giant Component w.r.t Total Number of nodes for a random Graph

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

```
k=0
list=[]
klist=[]
while k<5 :
  gph=nx.erdos_renyi_graph(500,k/499)
  giant = max(nx.connected_components(gph),key = len)
  ng = gph.size()
  list.append(ng/500)
  klist.append(k)
  k=k+0.1
  plt.figure(figsize=( 20 , 8 ))
  plt.plot(klist,list,color= 'red' ,linewidth= 3 ,marker= 'o' ,markersize= 10
,markerfacecolor= 'green')
  plt.xlabel( 'K values' )
  plt.ylabel( 'Corresponding Ng/N values' )
  plt.title( 'Plot:Ng/N values corresponding to average degree(k) values of random graph'
)
  plt.show()
```

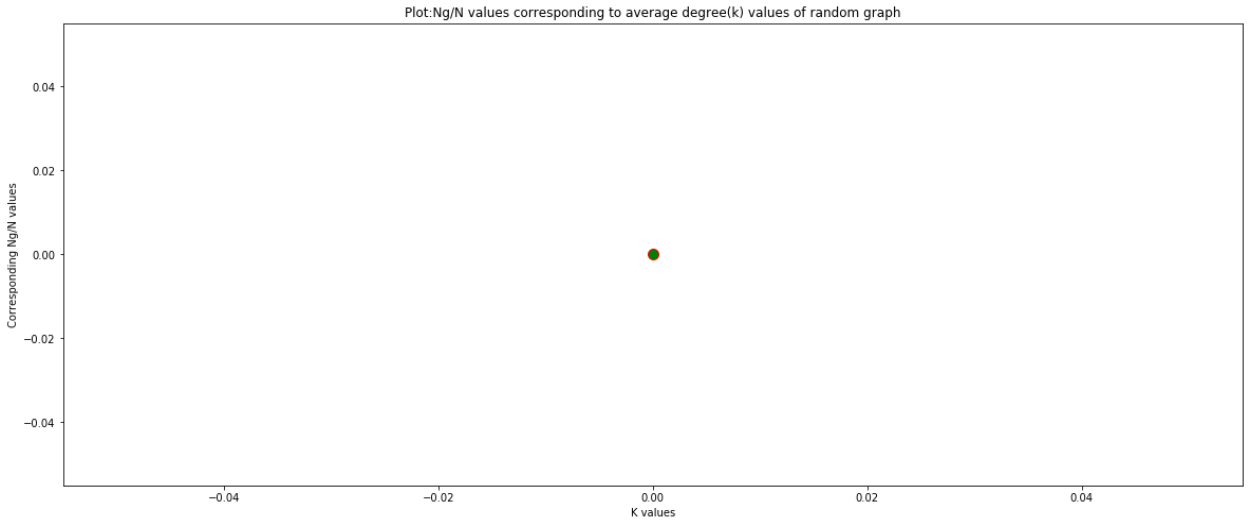
OBSERVATION:

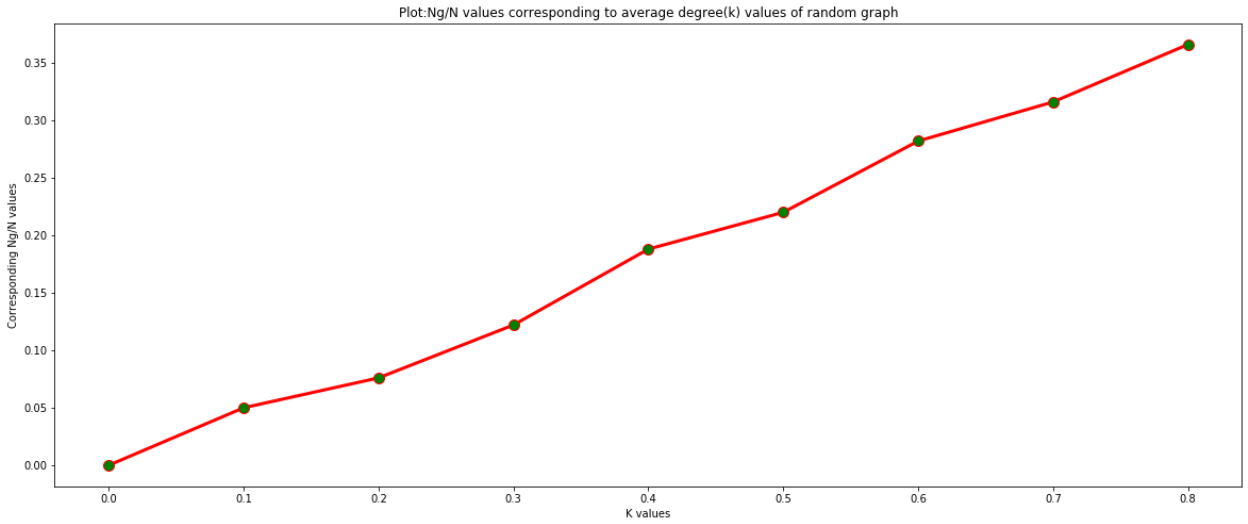
We plotted and watched different irregular charts with 500 hubs however the outcomes have a comparative example

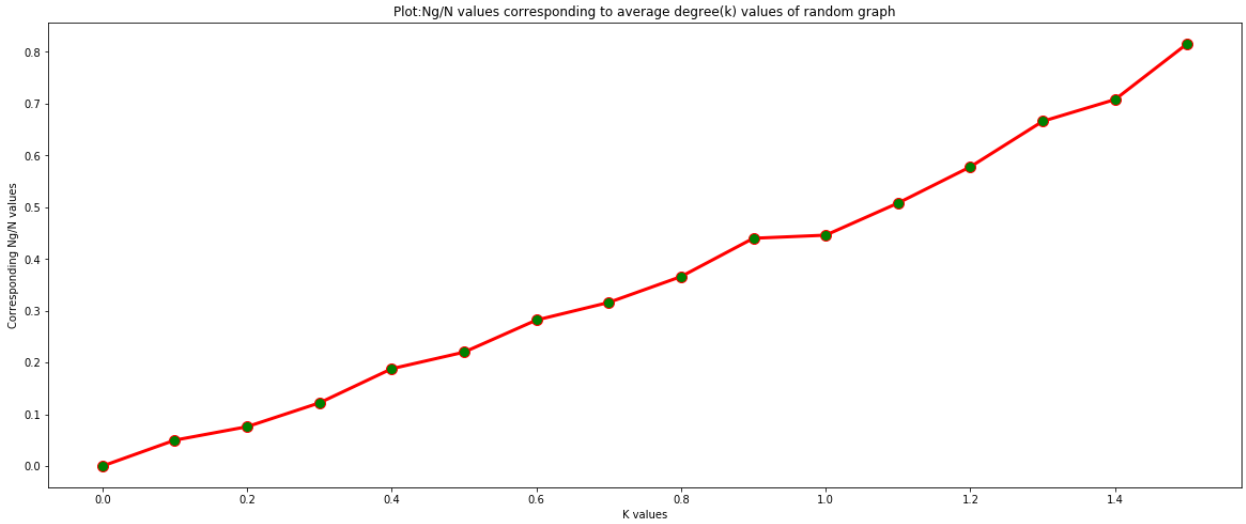
At first, the Ng/N esteem first stays near zero at that point has a fast increment and afterwards increments straightly as we bit by bit increment

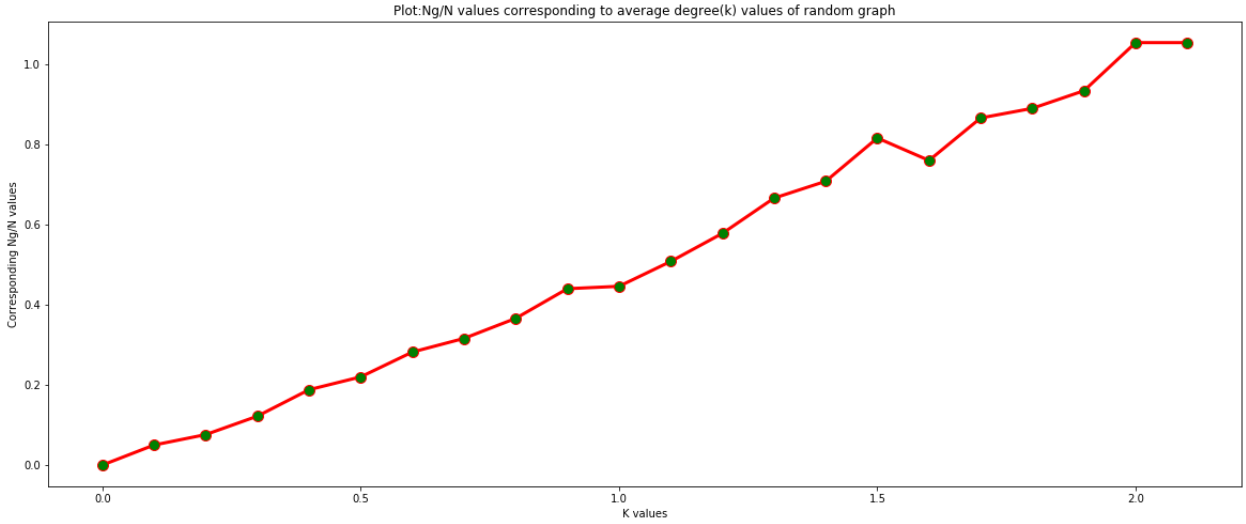
<k>.

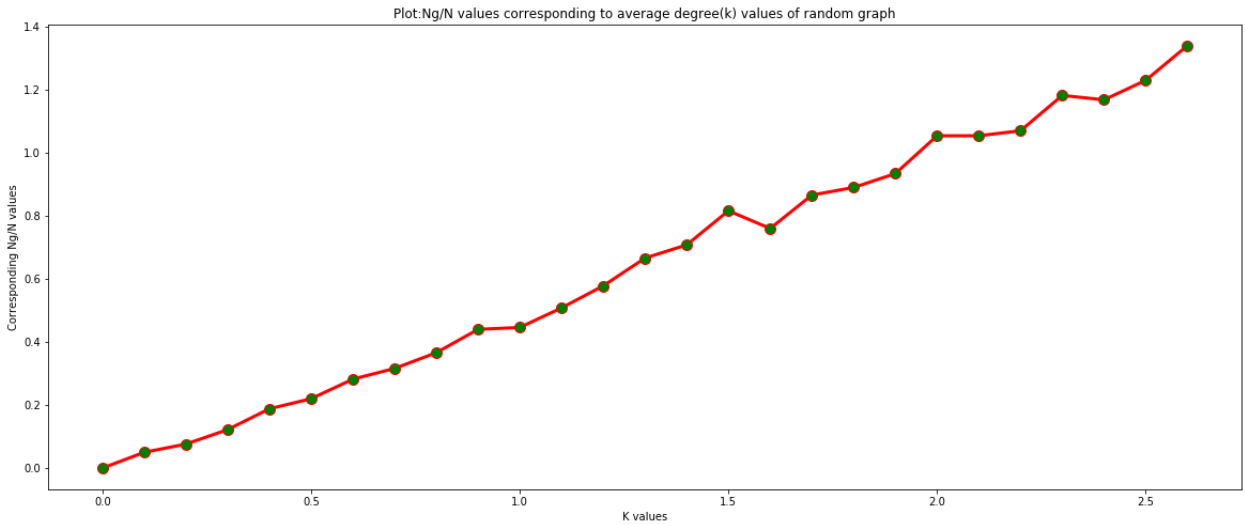
As the value of k is incremented from 0 to 5 by 0.1 at each iteration a number of graphs will be present, say a series of 50 graphs of which some are shown below in order.

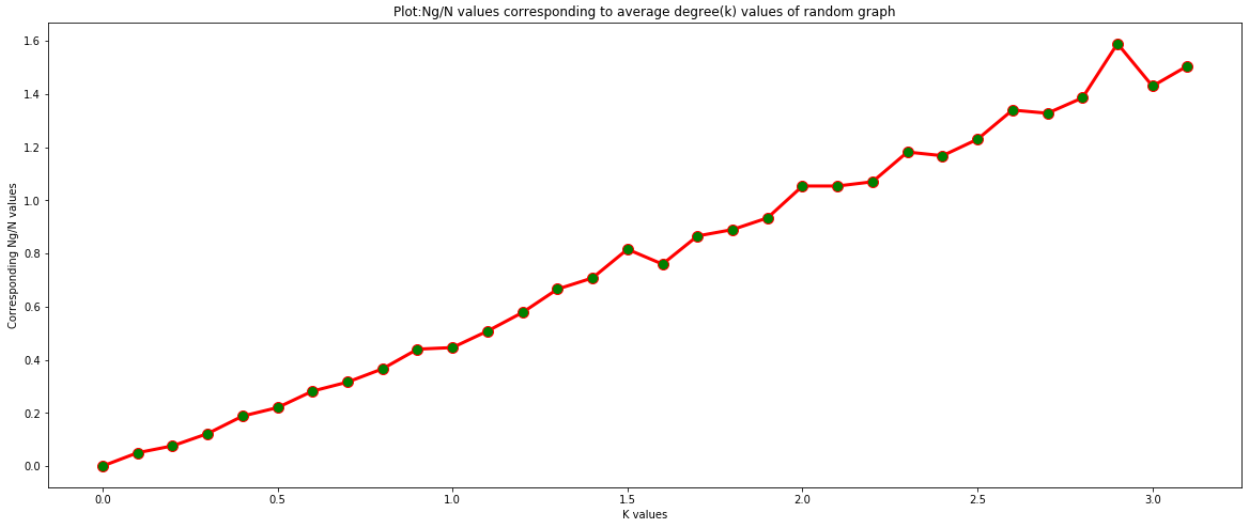


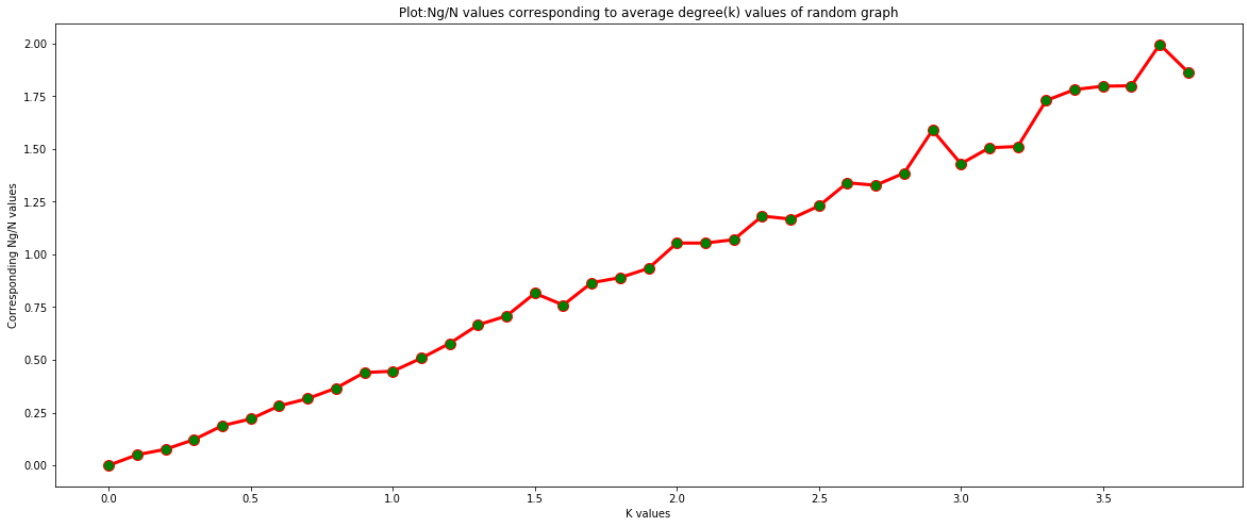


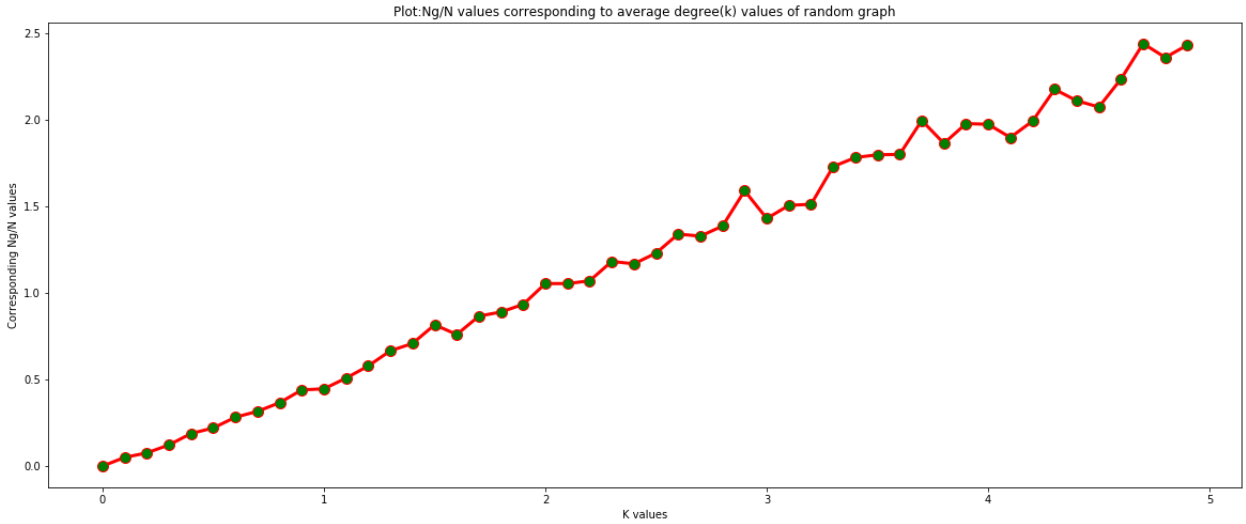

Plot:Ng/N values corresponding to average degree(k) values of random graph
Corresponding Ng/N values
K values
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4

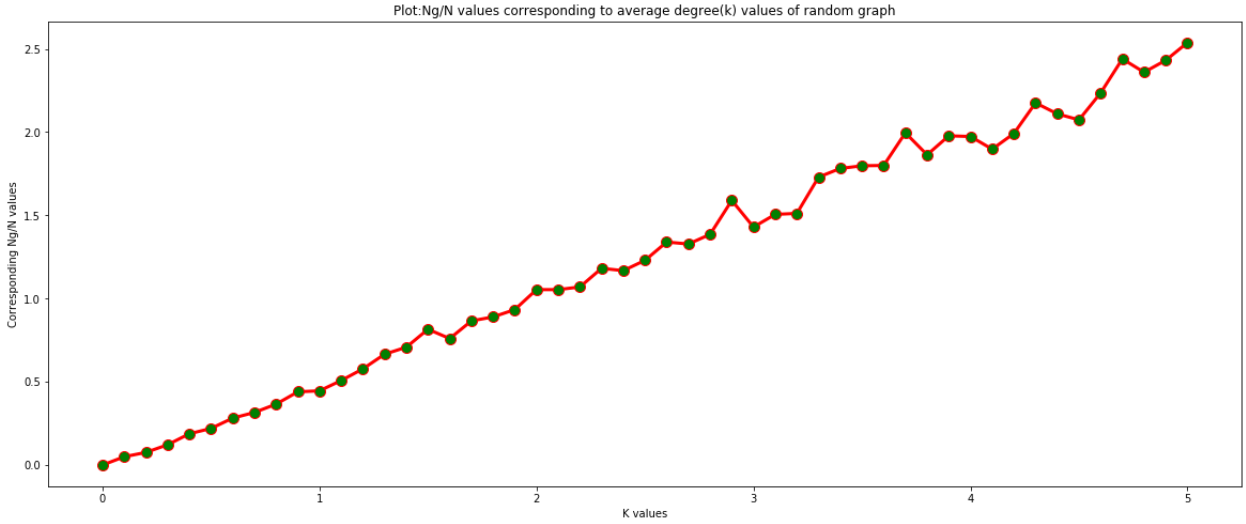

Plot:Ng/N values corresponding to average degree(k) values of random graph
Corresponding Ng/N values
K values
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.5
1.0
1.5
2.0


Plot:Ng/N values corresponding to average degree(k) values of random graph
Corresponding Ng/N values
K values
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0.0
0.5
1.0
1.5
2.0
2.5


Plot:Ng/N values corresponding to average degree(k) values of random graph
Corresponding Ng/N values
K values
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0.5
1.5
2.0
2.5
3.0


Plot:Ng/N values corresponding to average degree(k) values of random graph
Corresponding Ng/N values
K values
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5


Plot:Ng/N values corresponding to average degree(k) values of random graph
Corresponding Ng/N values
K values
0.0
0.5
1.0
1.5
2.0
2.5
0
1
2
3
4
5


Plot:Ng/N values corresponding to average degree(k) values of random graph
Corresponding Ng/N values
K values
0.0
0.5
1.0
1.5
2.0
2.5
0
1
2
3
4
5