SNA Final Report

Data Set 1 - Wikipedia Talk network

Team Members :

Apurv Jain 16UCC019

Aakirti Agarwal 16UCS002

Parul Shandilya 16UCS126

Priyansi Singh 16UCS143

Dataset Description :

The network contains all the users and discussion from the inception of Wikipedia till January 2008.

Nodes in the network represent Wikipedia users and a directed edge from node i to node j represents that user i at least once edited a talk page of user j.

Finding Random Sample

In the original network, there are 2394385 nodes and 5021410 edges. Since it is a very huge network , we used a random sample in order to get the inferences.

In [7]:

```
#finding random sample
import os
import random

filename = 'wiki-Talk1.txt'

result = []
f = open("wiki-Talk.txt", "a")

with open(filename) as fh:
    data = fh.readlines()
    for line in data:
        if line:
            words = line.split()
            result.append(words)

sample = random.sample(result, 1000)

for item in sample:
    f.write(item[0] + "    " + item[1] + "\n")
```

Now using the above random sample , we will read the graph G from the list of edges ('wiki-Talk.txt') and find the general information about the graph.

In [1]:

```
import networkx as nx

#reading graph from list of edges
G=nx.read_edgelist('wiki-Talk.txt',nodetype=int,create_using=nx.DiGraph())

print(" ")

#general information about the graph
```

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

```
Name:
Type: DiGraph
Number of nodes: 1745
Number of edges: 1004
Average in degree:   0.5754
Average out degree:   0.5754
```

Basic Information about graph:

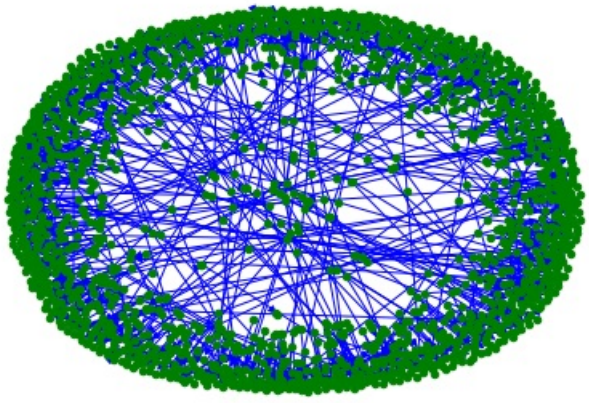
From here , we can see that the graph generated has 1745 nodes and 1004 edges. It is a directed graph and average in degree & out degree are 0.5771 (fraction) .

Graph Representation

In [4]:

```
import matplotlib.pyplot as plt

# drawing graph
nx.draw(G, pos=None, node_color='g', edge_color='b',node_size=20)
plt.show()
#print (nx.is_connected(G))
```



In the graph , nodes are represented by green colour and directed edges are represented by blue colour.

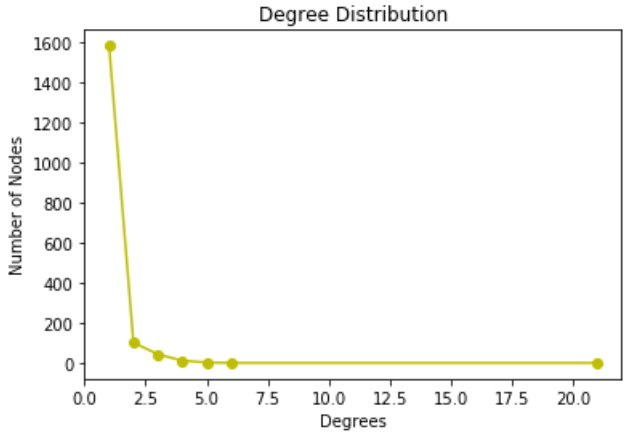
Degree Distribution

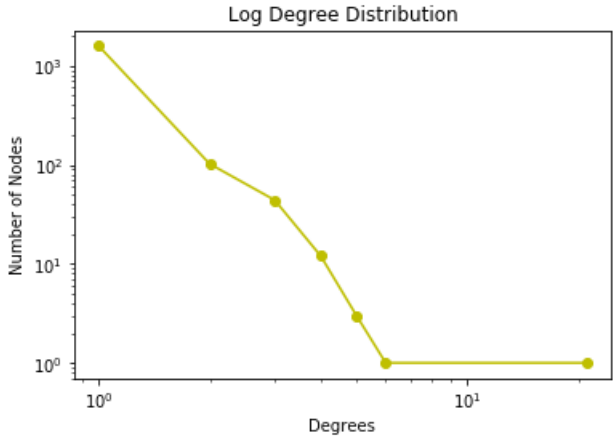
We will now try to find how

In [5]:

```
d=dict()
for x,y in nx.degree(G):
    if y not in d:
        d[y]=1
    else:
        d[y]+=1
#for x in d:
    #print(x,d[x])
plt.plot(d.keys(),d.values(),'yo-')
plt.xlabel("Degrees")
plt.ylabel("Number of Nodes")
plt.title("Degree Distribution")
#print(list(set(all_degrees)))
#print (nx.degree(G))
plt.show()
plt.loglog(d.keys(),d.values(),'yo-')
```

```
plt.xlabel("Degrees")
plt.ylabel("Number of Nodes")
plt.title("Log Degree Distribution")
plt.show()
```





Power Law

According to the Power Law ,very less nodes have high degree and as the number of nodes increases , the degree keeps on decreasing. So, when we draw the log-log graph of this data , we will get a straight line.

We can observe the same here in this network. So, we can conclude that a large number of users edited very less talk pages and the number of users decreased as the number of edited pages increased.

Density Calculation

A graph is more dense if the number of edges is close to the number of maximal edges possible. Density is calculated as the ratio of edges in network to maximal edges possible.

In [6]:

```
#calculating density
print (nx.density(G))
```

```
0.000329907205384
```

The density is 0.0033 . Thus , it is a sparse graph

Local Clustering Coeffecient

It measures the immediate neighbour connectivity of a node.

In [7]:

```
#getting average local clustering coeffecient
print "Average Local Clustering Coeffecient is:"
print nx.average_clustering(G)
```

```
print nx.average_clustering(G)

#printing local clustering coeffecient of some sample nodes
w=1

print("Node    Local Clustering Coeffecient")
for p,r in nx.clustering(G).items():
        if(w<=5):
            print(str(p)+"\t"+str(r))
            w=w+1
```

```
Average Local Clustering Coeffecient is:
0.0
Node    Local Clustering Coeffecient
1 0
2 0
3 0
4 0
5 0
```

In the dictionary , first element gives node number and 2nd element gives We get local clustering for all nodes as 0. This means that if user A is connected to a group of users, then no 2 users in that group are connected to each other.

Global Clustering Coeffecient

It measures the fraction of all possible triangles present in G.

In [8]:

```
#printing global clustering coeffecient of nodes
print "The Golbal Clustering Coeffecient is:"
print(nx.transitivity(G))
```

```
The Golbal Clustering Coeffecient is:
0
```

The Global Transitivity is 0 (Graph is intransitive). It means that no triangle is present in Graph G.That means , if user A edited talk page of user B , and user B edited talk page of user C , then user A did not edit talk page of user C.

Reciprocity

The reciprocity of a directed graph is defined as the ratio of the number of edges pointing in both directions to the total number of edges in the graph.

In [9]:

```
print "The reciprocity is :"
print nx.reciprocity(G)
```

```
The reciprocity is :
0.00199203187251
```

The reciprocity is 0.00199. This means that if user A edits the talk page of B , then it is very rare that B also edits the talk page of A.

Centrality Measures

Centrality measures indicate how important a node is in a graph. Following are the different Centrality Measures:

1. Degree Cenrality

There are two degrees involoved in directed graph - in_ degree and out_degree

In Degree

It computes the in-degree of all nodes in graph

It computes the in-degree of all nodes in graph

In [10]:

```
#calculating in-degree
dict1=G.in_degree()
print "The nodes with maximum in-degree are :"
i= max(nx.in_degree_centrality(G),key=(nx.in_degree_centrality(G)).get)
val=dict1[i]

print("Node    inDegree")
for p,r in dict1:
    if (r==val):
        print (str(p)+"\t"+str(r))

print (" ")
print "List of some nodes with their in-degree:"
w=1
for p,r in dict1:
    if(w<=5):
        print(str(p)+"\t"+str(r))
        w=w+1
```

```
The nodes with maximum in-degree are :
Node    inDegree
754 2
907 2
923 2
18935 2
2889 2
2893 2
2920 2
601149 2
4447 2

List of some nodes with their in-degree:
1 1
2 1
3 1
4 1
5 0
```

We observe that the highest in-degree is "2" which is associated with all the listed nodes. This means that all these listed users' talk pages are edited the maximum number of times. Also , we can conclude that the talk pages of these users are much popular.

Out Degree

It computes the out-degree of all nodes in graph

In [11]:

```
#calculating out-degree
dict1=G.out_degree()
print "The nodes with maximum out-degree are :"
i= max(nx.out_degree_centrality(G),key=(nx.out_degree_centrality(G)).get)

val=dict1[i]
print("Node    outDegree")
for p,r in dict1:
    if (r==val):
        print (str(p)+"\t"+str(r))

print (" ")
print "List of some nodes with their out-degree:"
w=1
for p,r in dict1:
    if(w<=5):
        print(str(p)+"\t"+str(r))
        w=w+1
```

```
The nodes with maximum out-degree are :
Node    outDegree
3215 21
```

```
List of some nodes with their out-degree:
1 1
2 2
3 2
4 0
5 1
```

The maximum out-degree is "21" which is associated with node 3125.This means that this user has contributed a lot in wikipedia.He has edited talk pages of most of the users.

Eigen Vector Centrality :

Eigenvector centrality computes the centrality for a node based on the centrality of its neighbors. The eigenvector centrality for node i is $Ax=\lambda x$ where A is the adjacency matrix of the graph G with eigenvalue λ

In [12]:

```
#calculating eigen vector centrality
import numpy as np
dict=nx.eigenvector_centrality_numpy(G)
i=max(nx.eigenvector_centrality_numpy(G),key=(nx.eigenvector_centrality_numpy(G).get))

print "Some of the nodes with their Eigen Vector Centrality are:"
#print i,dict[i]
print("Node    Eigen Vector Centrality")

# sorting based on values
dict_sorted_keys = sorted(dict, key=dict.get, reverse=True)
w=1
for r in dict_sorted_keys:
    if(w<=5):
        w=w+1
        print (str(r)+"\t"+str(dict[r]))
```

```
Some of the nodes with their Eigen Vector Centrality are:
Node    Eigen Vector Centrality
1163519 0.4472135954999581
2 0.447213595499958
4 0.447213595499958
1 0.44721359549995787
3 0.44721359549995787
```

Node 3 has the maximum eigen vector centrality measure (have sorted the list and displayed top 5 entries). This means that several influential users have edited this user's talk page.

Betweenness Centrality:

Betweenness centrality of a node v is the sum of the fraction of all-pairs shortest paths that pass through v.

Since path is irrelevant in context of this data , this centrality measure is not apt for our analysis.

Closeness Centrality

Closeness centrality of a node u is the reciprocal of the average shortest path distance to u over all n-1 reachable nodes.

Since path is irrelevant in context of this data , this centrality measure is not apt for our analysis.

Conclusion

1. In the original network, there are 2394385 nodes and 5021410 edges. Since it is a very huge network , we used a random sample in order to get the inferences.
2. Nodes in the network represent Wikipedia users and a directed edge from node i to node j represents that user i at least once edited a talk page of user j.
3. It is a directed,unweighted graph with 1745 nodes and 1004 edges.
4. This network roughly follows Power Law Distribution (Degree Distribution). It is a sparse graph.

5. It can be observed that if user A edits the talk page of B , then it is very rare that B also edits the talk page of A.
6. The graph is observed to be intransitive.
7. The user with the maximum out-degree has edited most of the talk pages.
8. The user with the maximum in-degree has the most popular talk pages.
9. The user with maximum eigen vector centrality measures has his talk pages edited by most influential users.

Data Set 2 - Texas road network

Dataset Description :

This is a road network of Texas. Intersections and endpoints are represented by nodes, and the roads connecting these intersections or endpoints are represented by edges.

Finding Random Sample

In the original network, there are 1379917 nodes and 1921660 edges. Since it is a very huge network , we used a random sample in order to get the inferences.

In [1]:

```
#finding random sample
import os
import random

filename = 'wiki-Talk.txt'

result = []
f = open("data.txt", "a")

with open(filename) as fh:
    data = fh.readlines()
    for line in data:
        if line:
            words = line.split()
            result.append(words)

sample = random.sample(result, 1000)

for item in sample:
    f.write(item[0] + "    " + item[1] + "\n")
```

Now using the above random sample , we will read the graph G from the list of edges ('data.txt') and find the general information about the graph.

In [27]:

```
import networkx as nx

#reading graph from list of edges
G=nx.read_edgelist('data.txt',nodetype=int,create_using=nx.DiGraph())

#general information about the graph
print (nx.info(G))
```

```
Name:
Type: DiGraph
Number of nodes: 1746
Number of edges: 3623
Average in degree:   2.0750
Average out degree:   2.0750
```

Basic Information about graph:

From here , we can see that the graph generated has 1746 nodes and 3623 edges. It is a directed graph and average in degree & out degree are 2.0750 (fraction) .

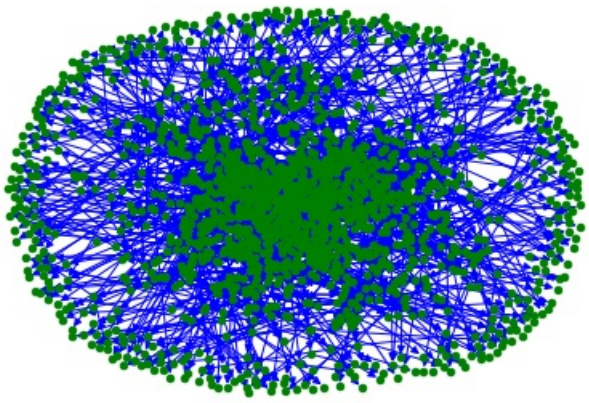
Graph Representation

In [28]:

In [28]:

```
import matplotlib.pyplot as plt

# drawing graph
nx.draw(G, pos=None, node_color='g', edge_color='b',node_size=20)
plt.show()
```



In the graph , nodes are represented by green colour and directed edges are represented by blue colour.

Degree Distribution

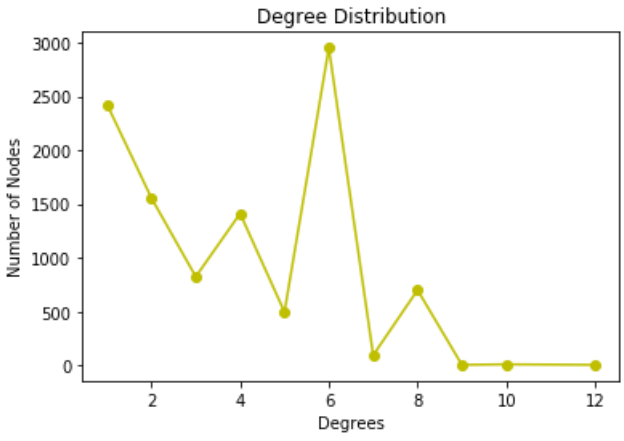
We will now try to find how

In [81]:

```
#d=dict()
for x,y in nx.degree(G):
    if y not in d:
        d[y]=1
    else:
        d[y]+=1
#for x in d:
    #print(x,d[x])
plt.plot(d.keys(),d.values(),'yo-')
plt.xlabel("Degrees")
plt.ylabel("Number of Nodes")
plt.title("Degree Distribution")
```

Out[81]:

```
Text(0.5,1,'Degree Distribution')
```



It can be observed that most of the nodes have degree '6' . This means that 6 roads passes through most of the roads.

Density Calculation

A graph is more dense if the number of edges is close to the number of maximal edges possible. Density is calculated as the ratio of edges in network to maximal edges possible.

In [67]:

```
#calculating density
print (nx.density(G))
```

```
0.00118912815867
```

The density is 0.001189 . Thus , it is a sparse graph

Local Clustering Coeffecient

It measures the immediate neighbour connectivity of a node.

In [78]:

```
#getting average local clustering coeffecient
print "Average Local Clustering Coeffecient is:"
print nx.average_clustering(G)

#printing local clustering coeffecient of all nodes
w=1
#for i in (nx.clustering(G).items()):
print "Some of the nodes with their clustering coeffecient are:"
print("Node    Local Clustering Coeffecient")
for p,r in nx.clustering(G).items():
    if(p==88):
        w=w+1
        print (str(p)+"\t"+str(r))
    elif(w>=2 and w<7):
        w=w+1
        print (str(p)+"\t"+str(r))
    elif(w>7):
        break
```

```
Average Local Clustering Coeffecient is:
0.0318649464955
Some of the nodes with their clustering coeffecient are:
Node    Local Clustering Coeffecient
88 0
89 0
90 0.166666666667
91 0.333333333333
92 0
93 0.166666666667
```

In the dictionary , first element gives node number and 2nd element gives We get average local clustering as 0.03186. This means that if node A has roads to a group of nodes, then no 2 nodes are connected by a road most of the time.

Global Clustering Coeffecient

It measures the fraction of all possible triangles present in G.

In [69]:

```
#printing global clustering coeffecient of nodes
print "The Golbal Clustering Coeffecient is:"
print(nx.transitivity(G))
```

```
The Golbal Clustering Coeffecient is:
0.0464488252768
```

The Global Transitivity is 0.0464. It means that some triangles are present in Graph G.That means, if node A is connected to node B

by a road and node B is directly connected to node C by a road , then it happens sometimes that node A has a road to node C.

Reciprocity

The reciprocity of a directed graph is defined as the ratio of the number of edges pointing in both directions to the total number of edges in the graph.

In [70]:

```
print nx.reciprocity(G)
```

```
0.874965498206
```

The reciprocity is 0.8749. This means that most of the roads are bidirectional.

Centrality Measures

Centrality measures indicate how important a node is in a graph. Following are the different Centrality Measures:

1. Degree Cenrality

It computes degree of all nodes in graph.

We observe that the highest in-degree is "6". This means that this node acts as a junction with maximum connectivity.

In [76]:

```
dict1=G.degree()
print "The nodes with maximum degree is :"
i= max(nx.out_degree_centrality(G),key=(nx.out_degree_centrality(G)).get)

val=dict1[i]
print("Node    Degree")
for p,r in dict1:
    if (r==val):
        print (str(p)+"\t"+str(r))

print (" ")
print "List of some nodes with their degree:"
w=1
for p,r in dict1:
    if(w<=5):
        print(str(p)+"\t"+str(r))
        w=w+1
```

```
The nodes with maximum degree is :
Node    Degree
219 12

List of some nodes with their degree:
0 6
1 6
2 6
3 6
4 6
```

The maximum degree is "12" which is associated with node 216.This mean this node acts as a junction and has maximum connectivity.

Eigen Vector Centrality :

Eigenvector centrality computes the centrality for a node based on the centrality of its neighbors. The eigenvector centrality for node i is $Ax=\lambda x$ where A is the adjacency matrix of the graph G with eigenvalue λ.

Since "influential" node is irrelevant to this network, we won't calculate it.

Betweenness Centrality:

Betweenness centrality of a node v is the sum of the fraction of all-pairs shortest paths that pass through v.

In [60]:

```
dict =nx.betweenness_centrality(G)
i= max(nx.betweenness_centrality(G),key=(nx.betweenness_centrality(G)).get)
#print i,dict[i]
print("Node    Betweenness Centrality")

dict_sorted_keys = sorted(dict, key=dict.get, reverse=True)
w=1
for r in dict_sorted_keys:
    if(w<=5):
        w=w+1
        print (str(r)+"\t"+str(dict[r]))
```

```
Node    Closeness Centrality
436 0.0792040523436
476 0.0782095053678
565 0.0767475903726
564 0.0767265604312
563 0.0767009870419
```

Node 436 has highest Between Centrality meausure. It means that most of the shortest paths passes through this node(considering weight of all edges as 1).

Closeness Centrality

Closeness centrality of a node u is the reciprocal of the average shortest path distance to u over all n-1 reachable nodes.

Since path is irrelevant in context of this data , this centrality measure is not apt for our analysis.

In [77]:

```
dict= nx.closeness_centrality(G, u=None, distance=None, wf_improved=True, reverse=False)
i= max(nx.closeness_centrality(G, u=None, distance=None, wf_improved=True, reverse=False),key=(nx.c
loseness_centrality(G, u=None, distance=None, wf_improved=True, reverse=False)).get)
#print i,dict[i]
print("Node    Closeness Centrality")

dict_sorted_keys = sorted(dict, key=dict.get, reverse=True)
w=1
for r in dict_sorted_keys:
    if(w<=5):
        w=w+1
        print (str(r)+"\t"+str(dict[r]))
```

```
Node    Closeness Centrality
476 0.0186759310163
565 0.0186402047016
564 0.018609690759
566 0.0185957385645
436 0.0185843386839
```

Node 476 has the highest Closeness Centrality measure.It means that from this node , we can reach other nodes very quickly.

Conclusion

1. In the original network, there are 1379917 nodes and 1921660 edges edges. Since it is a very huge network , we used a random sample in order to get the inferences.
2. Intersections and endpoints are represented by nodes, and the roads connecting these intersections or endpoints are represented by edges.
3. It can be observed that if node A has roads to a group of nodes, then no 2 nodes are connected by a road most of the time.
4. Most of the roads are bidirectional.
5. Most of the shortest paths passes through the node having highest betweenness centrality measure(considering weight of all

edges as 1).

Round 2

We will generate graphs using G(n,p) model where n(no. of nodes) = 500. We will vary ⟨k⟩ from 0 to 5 with increment of 0.1. For each value of ⟨k⟩,we will find the ratio Ng/N where N is the number of nodes in the graph. Later, we will plot this ratio with respect to ⟨k⟩ taking ⟨k⟩ as x-axis and ratio Ng/N as y-axis.

In [9]:

```
import networkx as nx

a=[]
n=500

for k in range (51):
    #calculating probability
    p= (1.0)*(k)/(10.0*(n-1))
    #generating graph based on G(n,p) model
    G1=(nx.gnp_random_graph(n, p, seed=None, directed=False))
    #finding the giant component Ng
    giant = max(nx.connected_component_subgraphs(G1), key=len)
    #calculating Ng/N ratio
    a.append(1.0*(giant.number_of_nodes())/(1.0*n))

#List containing all the ratios Ng/N for each average degree <k>
print ("Ratios Ng/N :")
print a
```

```
Ratios Ng/N :
[0.002, 0.006, 0.008, 0.008, 0.018, 0.01, 0.014, 0.084, 0.03, 0.186, 0.132, 0.312, 0.216, 0.158, 0
.456, 0.644, 0.662, 0.71, 0.666, 0.792, 0.8, 0.806, 0.83, 0.87, 0.886, 0.89, 0.882, 0.948, 0.942,
0.936, 0.944, 0.948, 0.962, 0.96, 0.958, 0.97, 0.942, 0.984, 0.964, 0.976, 0.972, 0.976, 0.988, 0.
984, 0.974, 0.986, 0.986, 0.976, 0.99, 0.994, 0.99]
```

Here, we have calculated ratio Ng/N for each of the above mentioned degree.

In [8]:

```
import matplotlib.pyplot as plt
b=[]

#storing all the average degrees <k> in list b
for i in range(0,51):
    b.append(i/10.0)

print ("Average degrees :")
print b
```

```
Average degrees :
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.
9, 2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8,
3.9, 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 5.0]
```

Here, we have made a list of all the required average degree values

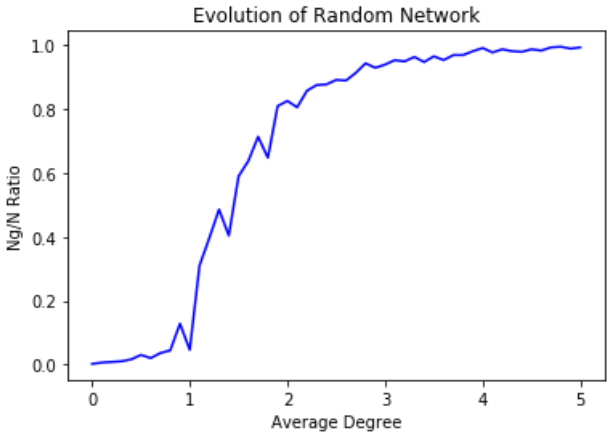
In [5]:

```
#Plotting graph between <k> and ratio Ng/N
plt.plot(b,a,'-b')

#X-axis label
plt.xlabel("Average Degree")
#Y-axis label
plt.ylabel("Ng/N Ratio")
#Plot title
plt.title("Evolution of Random Network")
```

Out[5]:

```
Text(0.5,1,'Evolution of Random Network')
```



The plot obtained is almost exponential in nature.Ng/N remains zero for small ‹k›, indicating the lack of a large cluster. Once ‹k› exceeds a critical value (k=1), Ng/N increases, signaling the rapid emergence of a large cluster that we call the giant component.

Round 3

Data Set - Facebook network

Dataset Description :

The facebook network contains 4039 users. An edge exists between 2 users if they are friend of each other. This is an undirected graph with a total of 88234 edges.

Finding Random Sample

Since the original network is huge , we used a random sample in order to get the inferences.

In [8]:

```
#finding random sample
import os
import random

filename = 'facebook.txt'

result = []
f = open("facebook1.txt", "a")

with open(filename) as fh:
    data = fh.readlines()
    for line in data:
        if line:
            words = line.split()
            result.append(words)

sample = random.sample(result, 5000)

for item in sample:
    f.write(item[0] + "    " + item[1] + "\n")
```

Now using the above random sample , we will read the graph G from the list of edges ('facebook1.txt') and find the general information about the graph.

In [1]:

```
import networkx as nx

#reading graph from list of edges
G=nx.read_edgelist('facebook1.txt',nodetype=int,create_using=nx.Graph())

print(" ")

#general information about the graph
print (nx.info(G))
```

```
Name:
Type: Graph
Number of nodes: 3127
Number of edges: 6812
Average degree:   4.3569
```

Basic Information about graph:

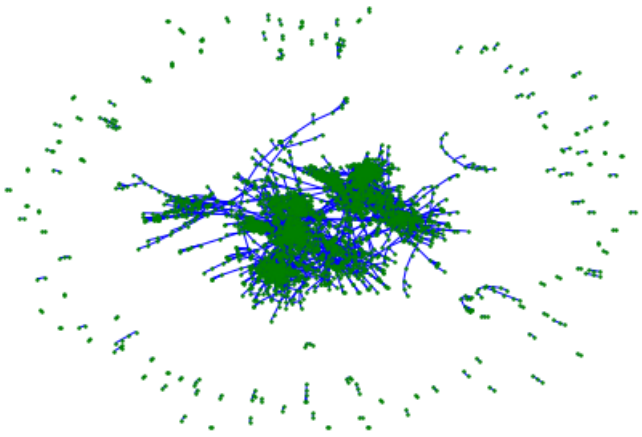
From here , we can see that the graph generated has 3127 nodes and 6812 edges. It is a directed graph and average degree is 4.3569.

Graph Representation

In [10]:

```
import matplotlib.pyplot as plt

# drawing graph
nx.draw(G, pos=None, node_color='g', edge_color='b',node_size=2)
plt.show()
#print (nx.is_connected(G))
```



In the graph , nodes are represented by green colour and directed edges are represented by blue colour.

Community detection

There are a variety of community detection algorithms. When detecting communities, we are interested in detecting communities with either (1) specific members or (2) specific forms of communities. We denote the former as member-based community detection and the latter as group-based community detection.

Hierarchial Clustering

Hierarchical clustering generates community hierarchies. Initially, n nodes are considered as either 1 or n communities in hierarchical clustering.These communities are gradually merged or split (agglomerative or divisive hierarchical clustering algorithms), depending on the type of algorithm,until the desired number of communities are reached.

We have used Girvan-Newman Algorithm here which is a divisive clustering algorithm. Since we have total 3124 nodes in our sampled graph , we have taken k=200 , i.e., we want to have 200 clusters in total.

In [ ]:

```
##comp = nx.algorithms.community.centrality.girvan_newman(G)
import itertools
k = 200
#q=[]
comp = nx.algorithms.community.centrality.girvan_newman(G)
limited = itertools.takewhile(lambda c: len(c) <= k, comp)
for communities in limited:
    print(tuple(sorted(c) for c in communities))
   # print "\n"
```

```
([0, 1, 3, 5, 9, 13, 21, 24, 25, 26, 29, 30, 34, 40, 45, 48, 53, 55, 56, 57, 58, 59, 62, 65, 66,
67, 69, 72, 75, 77, 88, 92, 94, 96, 98, 101, 102, 104, 107, 109, 112, 113, 118, 119, 122, 123, 127
, 128, 131, 133, 134, 136, 142, 143, 149, 156, 158, 161, 166, 169, 170, 172, 173, 181, 185, 186, 1
88, 190, 191, 194, 197, 199, 200, 203, 204, 205, 211, 212, 213, 217, 221, 222, 223, 224, 227, 228,
231, 232, 234, 236, 238, 239, 240, 241, 244, 246, 248, 249, 251, 252, 253, 257, 261, 265, 266, 268
, 271, 272, 276, 277, 280, 281, 282, 284, 285, 290, 291, 292, 295, 297, 299, 302, 303, 304, 313, 3
14, 315, 317, 318, 320, 322, 323, 325, 328, 329, 330, 332, 334, 342, 344, 345, 346, 348, 350, 352,
```

```
84, 685], [579, 588, 611, 618, 619, 631, 633, 639, 679], [587, 645], [601, 976], [649, 1023], [686,
688, 690, 693, 694, 695, 696, 697, 698, 701, 702, 703, 705, 706, 708, 709, 711, 712, 713, 714, 715
, 716, 717, 718, 719, 721, 722, 724, 726, 727, 728, 730, 731, 732, 733, 736, 737, 738, 739, 740, 7
41, 742, 744, 745, 747, 748, 751, 752, 754, 755, 756, 760, 761, 762, 764, 770, 771, 772, 773, 774,
778, 780, 781, 782, 783, 784, 789, 790, 793, 794, 795, 797, 799, 800, 805, 807, 809, 810, 811, 813
, 814, 815, 816, 817, 819, 820, 823, 824, 825, 826, 827, 828, 829, 830, 831, 833, 834, 835, 837, 8
38, 839, 840, 841, 842, 844, 845, 847, 848, 849, 851, 853, 855, 856, 858, 880, 882, 885, 887], [69
1, 746], [720, 787], [734, 846], [735, 769], [758, 808], [763, 836], [788, 806], [791, 796], [804,
889], [818, 822], [857, 862, 900, 928, 1031, 1061, 1085, 1098, 1136, 1140, 1144, 1165, 1174, 1251,
1307, 1317, 1325, 1328, 1360, 1410, 1421, 1555, 1567, 1588, 1593, 1615, 1687, 1698, 1705, 1719, 180
3, 1825, 1847, 3437, 3439, 3440, 3442, 3443, 3445, 3448, 3449, 3450, 3456, 3458, 3460, 3463, 3464,
3466, 3468, 3469, 3470, 3471, 3473, 3474, 3475, 3479, 3480, 3484, 3486, 3488, 3491, 3492, 3493, 349
7, 3498, 3499, 3500, 3501, 3506, 3508, 3509, 3510, 3511, 3514, 3516, 3518, 3521, 3523, 3525, 3528,
3529, 3531, 3532, 3534, 3536, 3538, 3540, 3541, 3542, 3543, 3545, 3546, 3547, 3548, 3550, 3553, 355
6, 3557, 3558, 3561, 3562, 3563, 3564, 3567, 3568, 3569, 3572, 3573, 3575, 3576, 3577, 3581, 3582,
3583, 3584, 3586, 3587, 3590, 3591, 3594, 3596, 3599, 3601, 3602, 3604, 3605, 3608, 3609, 3611, 361
2, 3613, 3614, 3615, 3616, 3617, 3618, 3619, 3620, 3621, 3625, 3627, 3629, 3631, 3633, 3635, 3636,
3639, 3640, 3642, 3646, 3647, 3648, 3649, 3651, 3653, 3654, 3655, 3656, 3657, 3659, 3660, 3662, 366
3, 3664, 3667, 3668, 3670, 3672, 3674, 3677, 3678, 3680, 3681, 3683, 3684, 3687, 3690, 3691, 3692,
3693, 3696, 3697, 3698, 3702, 3703, 3704, 3705, 3706, 3707, 3710, 3712, 3713, 3714, 3715, 3717, 371
8, 3720, 3721, 3722, 3724, 3725, 3728, 3730, 3731, 3734, 3735, 3736, 3737, 3738, 3739, 3740, 3741,
3743, 3747, 3750, 3751, 3753, 3755, 3756, 3758, 3761, 3762, 3763, 3765, 3767, 3770, 3772, 3776, 377
7, 3778, 3779, 3780, 3782, 3783, 3790, 3791, 3792, 3793, 3796, 3800, 3803, 3804, 3810, 3815, 3817,
3819, 3821, 3822, 3823, 3824, 3826, 3828, 3830, 3835, 3838, 3841, 3842, 3845, 3851, 3852, 3857, 385
8, 3859, 3860, 3862, 3863, 3866, 3867, 3868, 3869, 3870, 3871, 3873, 3877, 3881, 3883, 3884, 3886,
3889, 3892, 3896, 3900, 3901, 3902, 3903, 3906, 3907, 3911, 3912, 3915, 3917, 3921, 3924, 3926, 392
8, 3929, 3930, 3931, 3933, 3937, 3938, 3940, 3943, 3944, 3945, 3947, 3948, 3951, 3956, 3962, 3964,
3966, 3968, 3971, 3973, 3978], [863, 876, 886], [865, 868], [867, 873], [909, 1045, 1368, 1384], [9
29, 1838], [931, 1478, 1743, 1829], [955, 1254], [956, 979, 1889], [991, 1088, 1641], [1030,
1111], [1053, 1896], [1064, 1094], [1073, 1670, 1908], [1077, 1202], [1105, 1890], [1240, 1394], [
1297, 1486], [1433, 1708], [1477, 1792, 1860], [1490, 1631], [1548, 1648], [1654, 1762], [1657, 17
76, 1808], [1679, 1773], [1956, 2301, 2570], [1988, 2170], [1992, 2207, 2650], [2034, 2421], [2106
, 2437], [2208, 2530], [2255, 2466], [2320, 2490], [2382, 2456], [2426, 2444], [2432, 2486], [2670
, 2699, 2959, 2982, 3283, 3309, 3318, 3325, 3382], [2671, 3091], [2682, 3334], [2760, 3432],
[2805, 2997], [2886, 3322], [2957, 3061, 3398], [3059, 3300], [3105, 3251], [3180, 3181], [3197, 3
216], [3207, 3241], [3268, 3407], [3314, 3423], [3444, 3742], [3455, 3472, 3526, 3598], [3465, 351
9, 3571, 3632, 3675, 3701, 3757, 3759, 3836, 3899, 3910, 3918, 3957, 3975], [3481, 3919], [3482, 36
85], [3483, 3539, 3878, 3895], [3485, 3559], [3489, 3652], [3490, 3549, 3580], [3504, 3723], [3505
, 3607, 3644, 3773, 3850], [3512, 3760], [3515, 3593, 3768, 3794], [3554, 3726, 3829], [3579,
3708], [3592, 3626, 3916], [3610, 3888], [3630, 3634], [3641, 3960, 3967], [3643, 3855], [3665, 36
76], [3669, 3864], [3695, 3898], [3711, 3969], [3719, 3927], [3847, 3953], [3872, 3950], [3980, 39
81, 3987, 3991, 3997, 3998, 3999, 4001, 4018, 4019, 4022, 4028, 4029, 4030], [3982, 3986], [3985, 3
988, 4014])
```

This shows set of all clusters starting from 1 cluster till 200 clusters are made

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 vertices are connected via edges. Communities are then found by reporting the connected components of this graph.

In [3]:

```
#fixing k=4
from networkx.algorithms.community import k_clique_communities
c = list(k_clique_communities(G, 3))

#printing all the communities
print c
```

```
[frozenset([1017, 1793, 1538, 1539, 1668, 1267, 1004, 1391, 1491, 1717, 1782, 1559, 1721, 1211, 118
1, 1663]), frozenset([2410, 2395, 2140, 2109]), frozenset([67, 40, 169, 334]), frozenset([1984, 22
75, 2532, 2348, 2124, 2253, 2190, 2095, 2257, 2578, 2115, 2261, 2244, 2266, 2043, 2526]), frozenset
([2384, 2344, 2102, 2390, 1912, 1577, 2509]), frozenset([2560, 2306, 2307, 2564, 2309, 2433, 2059,
2064, 2578, 2323, 2324, 2073, 2074, 2331, 2590, 2593, 2339, 2340, 2600, 2602, 2347, 2093, 2607, 235
2, 2354, 2611, 2615, 2619, 2229, 2624, 2370, 2630, 2631, 2122, 2123, 2381, 2638, 2131, 2489, 2532,
2395, 2140, 2654, 2655, 2404, 2407, 2410, 2414, 2165, 2623, 2118, 2430, 2112, 2374, 2184, 1938, 194
1, 2200, 2201, 1946, 2460, 2586, 1953, 2210, 2467, 2469, 2213, 2216, 2218, 2220, 2290, 1971, 2484,
2485, 2233, 2237, 1985, 1986, 2244, 1989, 1993, 2507, 2253, 2005, 2521, 2526, 2271, 2043, 2276, 253
9, 2542, 2543, 2033, 2546, 2037, 2088, 2553, 2554, 2299, 2090, 2046]), frozenset([1120, 1576, 1404]
), frozenset([1288, 1809, 1621]), frozenset([456, 537, 431]), frozenset([1184, 1888, 1603, 1287, 1
```

```
frozenset([2676, 3047, 3098, 2877]), frozenset([2030, 2381, 2542]), frozenset([352, 409, 492]),
frozenset([1752, 1491, 1572]), frozenset([3185, 3419, 3051]), frozenset([1352, 1891, 1659]),
frozenset([400, 513, 444]), frozenset([2153, 2475, 2047]), frozenset([2520, 2324, 2229]),
frozenset([896, 989, 1647]), frozenset([2112, 2308, 2188]), frozenset([2600, 1941, 1918]),
frozenset([1826, 1293, 1605]), frozenset([3104, 3026, 2873, 3084]), frozenset([2130, 2091, 2341]),
frozenset([3089, 3252, 3188]), frozenset([1800, 1331, 1547]), frozenset([2561, 2323, 2109]),
frozenset([2273, 2498, 2328, 2010]), frozenset([665, 658, 662]), frozenset([2492, 2109, 2206]),
frozenset([697, 726, 745]), frozenset([1912, 2190, 2631]), frozenset([1409, 1571, 1535]),
frozenset([1352, 1600, 1471]), frozenset([993, 526, 1110]), frozenset([1683, 1420, 1509]),
frozenset([2329, 2604, 2601]), frozenset([3296, 3049, 3247]), frozenset([1689, 1126, 1198]),
frozenset([1925, 2492, 2602, 2124]), frozenset([919, 942, 1871]), frozenset([1584, 1813, 1669]),
frozenset([1912, 2224, 2042]), frozenset([2083, 2331, 2108, 1983]), frozenset([2226, 2379, 2612]),
frozenset([504, 426, 546]), frozenset([1136, 1061, 1687]), frozenset([665, 670, 583]),
frozenset([2458, 2491, 2396]), frozenset([1912, 1718, 2239]), frozenset([3008, 2889, 1684]),
frozenset([2560, 2630, 2551]), frozenset([3184, 2905, 3226]), frozenset([1209, 1339, 1804]),
frozenset([3611, 3845, 3790]), frozenset([2049, 2314, 2173]), frozenset([713, 851, 828]),
frozenset([2290, 1918, 2430]), frozenset([3336, 3066, 1684]), frozenset([2528, 2313, 2558]),
frozenset([3049, 3267, 2853]), frozenset([1536, 1299, 1366]), frozenset([553, 506, 475]),
frozenset([2624, 1946, 2549]), frozenset([1376, 1269, 934, 1467]), frozenset([2863, 3101, 2901]),
frozenset([1024, 1619, 1126]), frozenset([1954, 2571, 2143]), frozenset([3448, 3437, 3677]),
frozenset([1377, 1620, 1557]), frozenset([1250, 1902, 1390]), frozenset([3132, 2869, 3397]),
frozenset([2712, 3035, 3198]), frozenset([2643, 2246, 2047]), frozenset([2043, 2500, 2334]),
frozenset([2945, 2757, 2927]), frozenset([1912, 2065, 2592]), frozenset([107, 1774, 1639]),
frozenset([2226, 2419, 2038])]
```

Hered,we can observe that all cliques of size k = 3 have been identified and cliques that share k−1 = 2 nodes are connected. All the obtained communities are given above.