Assignment 1 - Network Graph Sciences Analysis

Florian Barral

March 4, 2018

Question 1

(a) $\mathbf{k} = A.1$
(b) $m = |E| = \frac{1}{2}k^T k = \frac{1}{2}\sum_{i=0}^{n} k_i$
(c) $N = AA^T$,

Question 2

Let V be partitioned into two disjoint sets V_1, V_2. Because there are no internal edges in each set V_1, V_2
:

$$
\begin{gathered}
\{\# \text{ of out edges of } V_1\} = \{\# \text{ of out edges of } V_2\} \\
\sum_{i \in V_1} k_i = \sum_{i \in V_2} k_i \\
|V_1| \frac{\sum_{i \in V_1} k_i}{|V_1|} = |V_2| \frac{\sum_{i \in V_2} k_i}{|V_2|} i \\
|V_1| c_1 = |V_2| c_2 \\
n_1 c_1 = n_1 c_1
\end{gathered}
$$

Question 3

Graph : G=(V,E)
(a) By definition, the element A_{ij} of the square matrix contains the number of paths of length 3 that starts from node i and end at node j. A triangle is designed when the node participates simultaneously as the start node and the end node, so we are interested in diagonal elements A_{ii} : nodes involved in triangles.
Therefore, the trace of A^3 is related to the number of triangles. As :
- 3 nodes are involved in a triangle : we are then triple counting the number of triangles
- the graph is undirected : we are then doubling the number of triangles
So to conclude, the total number of triangles in the graph is

$$\Delta(G) = \frac{1}{6} tr(A^3)$$

(b) As we know the trace of a matrix if the sum of eigen values. We have :

$$\Delta(G) = \frac{1}{6} \sum_{i \in V} \lambda_i^3$$

(c) As we have just shown that the diagonal element A_{ii} is Δ_i the number of triangles that node i participates in, we just need to find a way to derive A_{ii}. The adjacency matrix A being real and symmetric, we have $A = U \Sigma U^T$ with $UU^T = I$. Then $A^3 = U \Sigma^3 U^T$. So with the eigenvectors $(\vec{u_1}, ... , \vec{u_n})$ of the eigenvalues $(\lambda_1, ..., \lambda_n)$, and u_{ij} the ith element of $\vec{u_j}$, we can derive Δ_i as follows:

$$\begin{aligned}
\Delta_i = \frac{1}{2} A_{ii}^3 &= \frac{1}{2} \sum_{k \in V} u_{ik} (\Sigma^3 U)_{ki} \\
&= \frac{1}{2} \sum_{k \in V} u_{ik} (\sum_{l \in V} \Sigma_{kl}^3 u_{li}) \\
&= \frac{1}{2} \sum_{k \in V} u_{ik} (\lambda_k^3 u_{ki}) \\
\Delta_i &= \frac{1}{2} \sum_{k \in V} \lambda_k^3 u_{ki}^2
\end{aligned}$$

Question 4

(a) $G_{n,p}$undirected graph on n nodes and each edge (u,v) appears i.i.d. with probability p **(a)** There are $\binom{n}{3}$ triples of vertices . Each triple has statistically a probability of p of being a triangle. Let Δ_{ijk} be the random variable for the triangle with vertices i, j, and k being present equals to 1 if vertices participate to the same triangle. Then the number of triangles is the expectancy of $\sum_{ijk} \Delta_{ijk}$:

$$\begin{aligned} E[\sum_{ijk} \Delta_{ijk}] = * \sum_{ijk} E[\Delta_{ijk}] = \sum_{ijk} p^3 &= \binom{n}{3} p^3 \\ &= \frac{n(n-1)(n-2)}{3!} p^3 \\ &\underset{\infty}{\sim} \frac{1}{6}(np)^3 = \frac{1}{6}c^3 \end{aligned}$$

*the expected value of a sum of random variables is the sum of the expected values, because the events are identically distributed.

(b) Mutatis mutandis :

$$\begin{aligned} E[\sum_{ijk} \Delta'_{ijk}] = * \sum_{ijk} E[\Delta'_{ijk}] = \sum_{ijk} \binom{3}{2} p^2 &= \binom{n}{3}\binom{3}{2} p^2 \\ &= \frac{n(n-1)(n-2)}{3!} \frac{3!}{2!} p^2 \\ &\underset{\infty}{\sim} \frac{1}{2} nc^2 \end{aligned}$$

(c) $C = \frac{\frac{1}{6}c^3 3}{\frac{1}{2}nc^2} = \frac{c}{n} \sim p$

Question 5

(a)

$$x_i = \sum_{k=1}^{\infty} \sum_{j \in V} \alpha^k A_{ij}^k$$

(b) As we know above :

$$A_{ij}^k = \sum_{l \in V} \lambda_l^k u_{li} u_{lj}$$

Question 6

Let's reason by equivalence to prove $\boxed{\dfrac{1}{C_A} + \dfrac{n_A}{n} = \dfrac{1}{C_B} + \dfrac{n_B}{n}}$:

$$\begin{aligned}
& \frac{1}{C_A} + \frac{n_A}{n} = \frac{1}{C_B} + \frac{n_B}{n} \\
\iff & \frac{\sum_j d_{Aj}}{n} + \frac{n_A}{n} = \frac{\sum_j d_{Bj}}{n} + \frac{n_B}{n} \\
\iff & \sum_j d_{Aj} + n_A = \sum_j d_{Bj} + n_B \\
\iff & \sum_{j \in A \cup B} d_{Aj} + n_A = \sum_{j \in A \cup B} d_{Bj} + n_B \\
\iff & \sum_{j \in A} d_{Aj} + \sum_{j \in B} d_{Aj} + n_A = \sum_{j \in A} d_{Bj} + \sum_{j \in B} d_{Bj} + n_B \\
\iff & \sum_{j \in A} d_{Aj} + \sum_{j \in B} (d_{Bj} + 1) + n_A = \sum_{j \in A} (d_{Aj} + 1) + \sum_{j \in B} d_{Bj} + n_B \\
\iff & \sum_{j \in A} d_{Aj} + \sum_{j \in B} d_{Bj} + n_B + n_A = \sum_{j \in A} d_{Aj} + n_A + \sum_{j \in B} d_{Bj} + n_B \quad \text{which is true}
\end{aligned}$$

In [1]:
```
import networkx as nx
import math
import random
```

In [2]:
```
#nx.test()
```

In [3]:
```
with open('/Users/florian/Documents/ETUDES/Etudes Post-Prepa/4. ESS
EC MSc DSBA/T2/5. Networks/assignment1 code/ca-GrQc.txt') as f:
    lines = f.readlines()
myList = [line.strip().split() for line in lines]
# to remove useless text of the file, following :
del myList[0]
del myList[0]
del myList[0]
del myList[0]
#myList
```

In [4]:
```
g = nx.Graph()
g.add_edges_from(myList)
```

In [82]:
```
# Drawing total graph
nx.draw(g)
```

Question 7 (a) - (1), (2)

In [13]:
```
V=len(g.nodes())
E=len(g.edges())
print '(1)'
print 'Number of nodes:', V
print 'Number of edges:', E
print '-----------'
print '(2)'
print 'Is the graph connected? ==>',nx.is_connected(g)
print '(2)(i)'
print 'Number of connected components (CCs)', nx.number_connected_c
omponents(g)
```

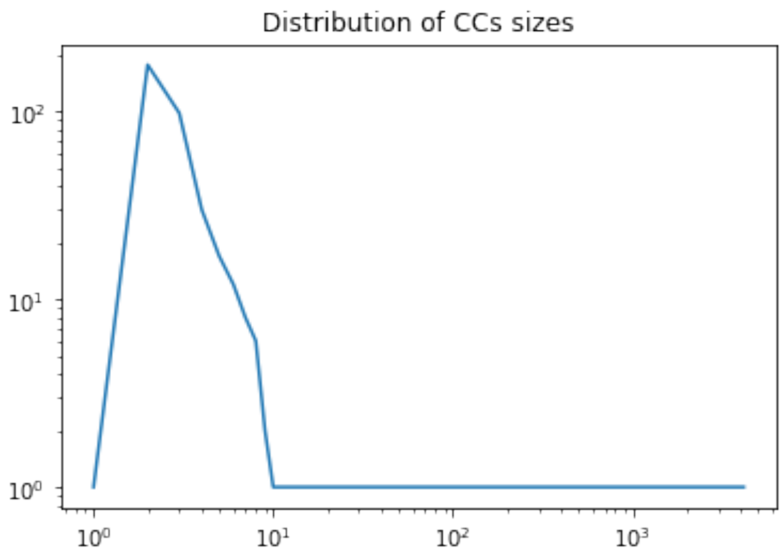
```
 (1)
Number of nodes: 5242
Number of edges: 14490
-----------
(2)
Is the graph connected? ==> False
(2)(i)
Number of connected components (CCs) 355
```

In [14]:
```
list_of_components = sorted(nx.connected_components(g), key = len,
reverse=True)

# count the number of components for each size
def values_distribution_of_components(numerical_list):
    numerical_list = sorted(numerical_list)
    dico = {}
    for i in numerical_list:
        if len(i) not in dico:
            dico[len(i)] = 1
        else:
            dico[len(i)] += 1
    return(dico)

sizes_dico = values_distribution_of_components(list_of_components)
sizes_dico
```

Out[14]:
```
{1: 1,
 2: 177,
 3: 98,
 4: 30,
 5: 17,
 6: 12,
 7: 8,
 8: 6,
 9: 2,
 10: 1,
 12: 1,
 14: 1,
 4158: 1}
```

In [15]:
```
print '(2)(ii)'
import matplotlib.pylab as plt
%matplotlib inline
plt.plot(*zip(*sorted(sizes_dico.items())))
plt.xscale('log')
plt.yscale('log')
plt.title('Distribution of CCs sizes')
plt.show()
```

```
(2)(ii)
```



In [17]:
```
GCC = max(nx.connected_component_subgraphs(g),key=len)
GCC_nodes=len(GCC.nodes())
GCC_edges=len(GCC.edges())

print 'The largest connected component (GCC) has', GCC_nodes,'nodes
and', GCC_edges, 'edges.'
print 'This represents', round(float(GCC_nodes)/V*100,2), '% of tot
al nodes and' , round(float(GCC_edges)/E*100,2),'% of total edges.'
print 'This GCC is so really important!! It has relatevely more edg
es than nodes, this means the others CCs might be very isolated.'
```

```
The largest connected component (GCC) has 4158 nodes and 13425 edg
es.
This represents 79.32 % of total nodes and 92.65 % of total edges.
This GCC is so really important!! It has relatevely more edges tha
n nodes, this means the others CCs might be very isolated.
```

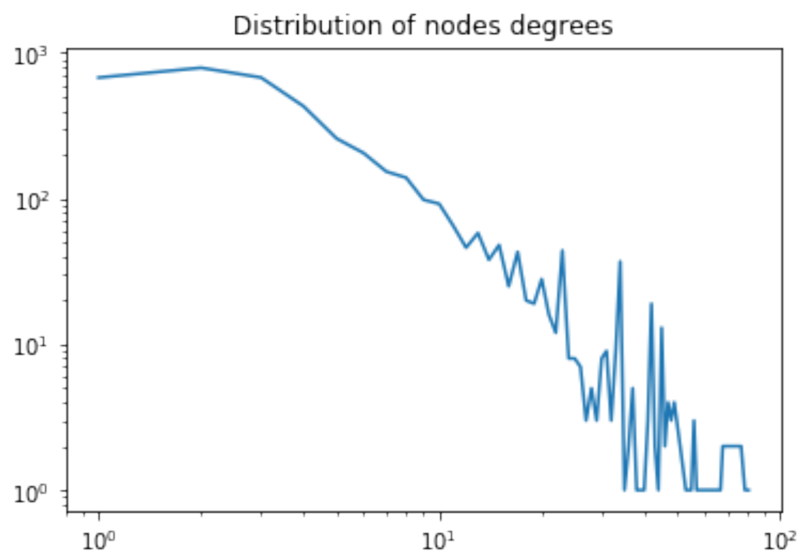
Question 7 (b)

In [18]:
```
import numpy as np
degrees=dict(GCC.degree())
degrees_list=sorted(degrees.values())
print 'Max degree of the nodes of the graph:', np.max(degrees_list)
print 'Min degree of the nodes of the graph:', np.min(degrees_list)
print 'Median degree of the nodes of the graph:', np.median(degrees
_list)
print 'Mean degree of the nodes of the graph:', np.mean(degrees_lis
t)
print '-----'
print 'Considering the min and median degree, we can conclude there
are many nodes isolated as leaves of a tree.'
```

```
Max degree of the nodes of the graph: 81
Min degree of the nodes of the graph: 1
Median degree of the nodes of the graph: 3.0
Mean degree of the nodes of the graph: 6.45887445887
-----
Considering the min and median degree, we can conclude there are m
any nodes isolated as leaves of a tree.
```

In [32]:
```
# get the number of nodes with same degree
degrees_list = degrees.values()

degrees_distribution_dico = {}
for i in degrees_list:
    if i not in degrees_distribution_dico:
        degrees_distribution_dico[i] = 1
    else:
        degrees_distribution_dico[i] += 1
#degrees_distribution_dico
```

In [33]:
```
# and plot it
plt.plot(*zip(*sorted(degrees_distribution_dico.items())))
plt.xscale('log')
plt.yscale('log')
plt.title('Distribution of nodes degrees')
plt.show()
```



Question 7 (c)

Question 7 (c) (1)

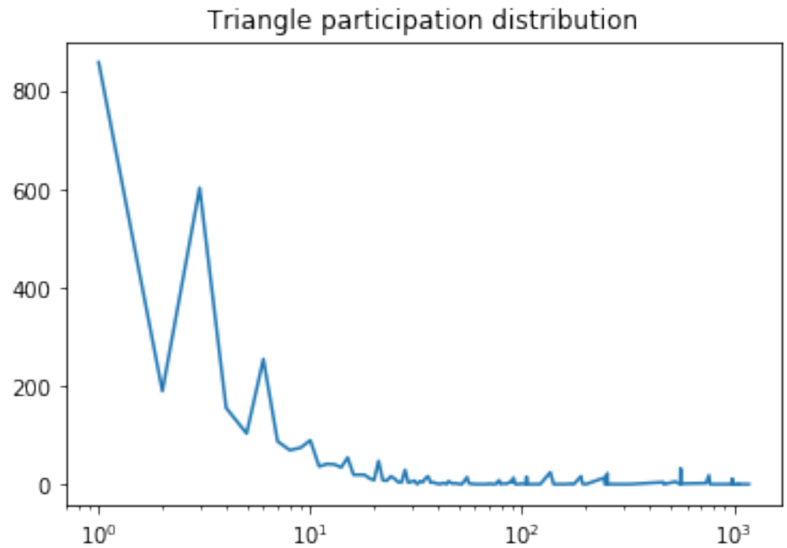
In [34]:
```
# for each node get its triangle participation
nodes_participation = nx.triangles(GCC)
# derive the number of triangles
nbr_triangles_GCC = sum(list(nodes_participation.values()))/3
print 'The total number of triangles in the GCC of the network is',
nbr_triangles_GCC
```

```
The total number of triangles in the GCC of the network is 47779
```

Question 7 (c) (2)

In [35]:
```
# count the number of nodes having same triangle participation
triangles_participation_distribution = {}
for key, value in nodes_participation.iteritems():
    if value not in triangles_participation_distribution:
        triangles_participation_distribution[value] = 1
    else:
        triangles_participation_distribution[value] += 1
```

In [36]:
```
plt.plot(*zip(*sorted(triangles_participation_distribution.items())
))
plt.xscale('log')
plt.title('Triangle participation distribution')
plt.show()
```



Question 7 (d)

Computing the whole spectrum of the adjacency matrix can be a computational bottleneck because the underlying algorithm complexity is at least $O(n^2)$ or $O(n^3)$

In [37]:
```
from numpy import linalg as LA

#Generating adjacency matrix
GCC_adj_matrix = nx.to_numpy_matrix(GCC)

#Isolating eigen_values
sorted_eigen_values = LA.eigvals(GCC_adj_matrix).real
```

In [38]:
```
sorted_eigen_values
```

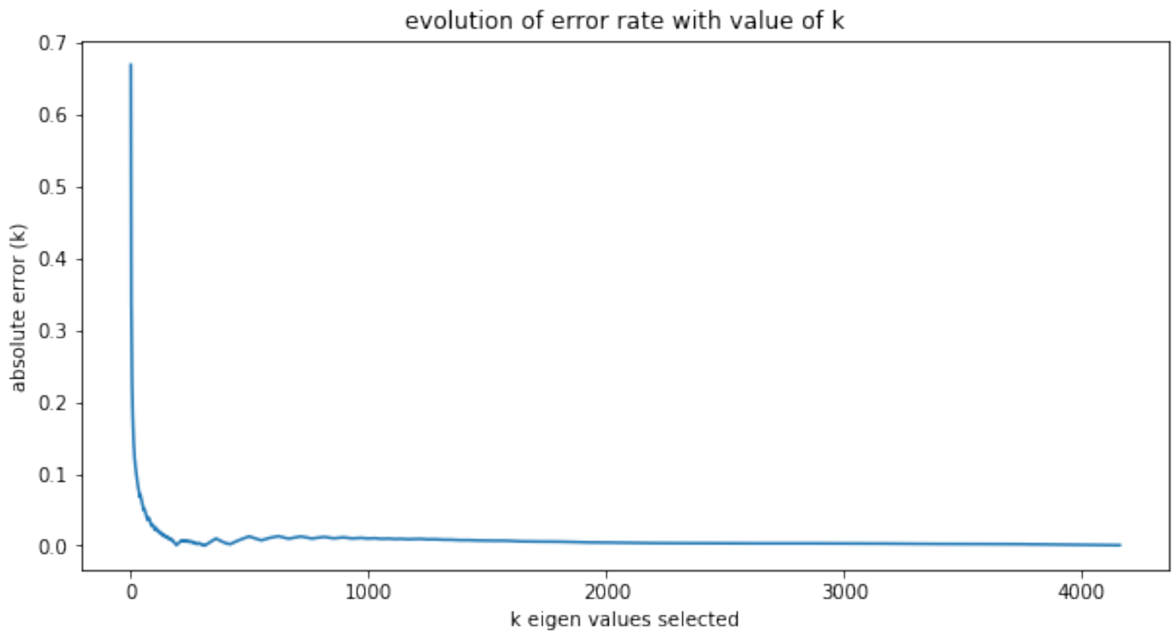
Out[38]:
```
array([ 45.61666218,  38.12196449,  34.00715914, ...,  -1.
,
        -1.        ,  -1.        ])
```

In [39]:
```
#Raising the eigenvalues to power of 3
power_3_eigen = [number**3 for number in sorted_eigen_values]
nbr_triangles_GCC_eig_total = sum(power_3_eigen)/6
nbr_triangles_GCC_eig_total
```

Out[39]:
```
47808.500000000997
```

In [40]:

```
#Error calculator
def error_calculator(k):
    prediction = sum(power_3_eigen[0:k]) / 6
    error = abs((prediction - nbr_triangles_GCC)/nbr_triangles_GCC)
    return error

#Calculating the errors
error_dico={}
for k in range(1,4159):
    error_dico[k] = error_calculator(k)

#plotting the evolution of the error
plt.figure(figsize=(10,5))
plt.plot(*zip(*sorted(error_dico.items())))
plt.xlabel("k eigen values selected")
plt.ylabel("absolute error (k)")
plt.title("evolution of error rate with value of k")
plt.show()
```



Question 8

Erdos-Renyi random graph $G_{n,p}$

In [41]:
```
n = 1000
p = 0.009
GERR=nx.fast_gnp_random_graph(n,p)
theor_mean_degree_GERR=(n-1)*p
```

Question 8 (a)

In [42]:
```
print 'The mean degree of the graph is', theor_mean_degree_GERR
```

```
The mean degree of the graph is 8.991
```

The mean degree formula being : $c =< k >= \sum_{m=0}^{\binom{n}{2}} \frac{2m}{n} Pr(m) = \frac{2}{n}\binom{n}{2}p = (n-1)p$

Question 8 (b)

In [43]:
```
print 'Is the graph connected? ==>',nx.is_connected(GERR)
```

```
Is the graph connected? ==> True
```

A random graph is connected if its mean degree c is higher than $ln(n)$. Which is indeed the case here.

In [44]:
```
theor_mean_degree_GERR > math.log(1000)
```

Out[44]:
```
True
```

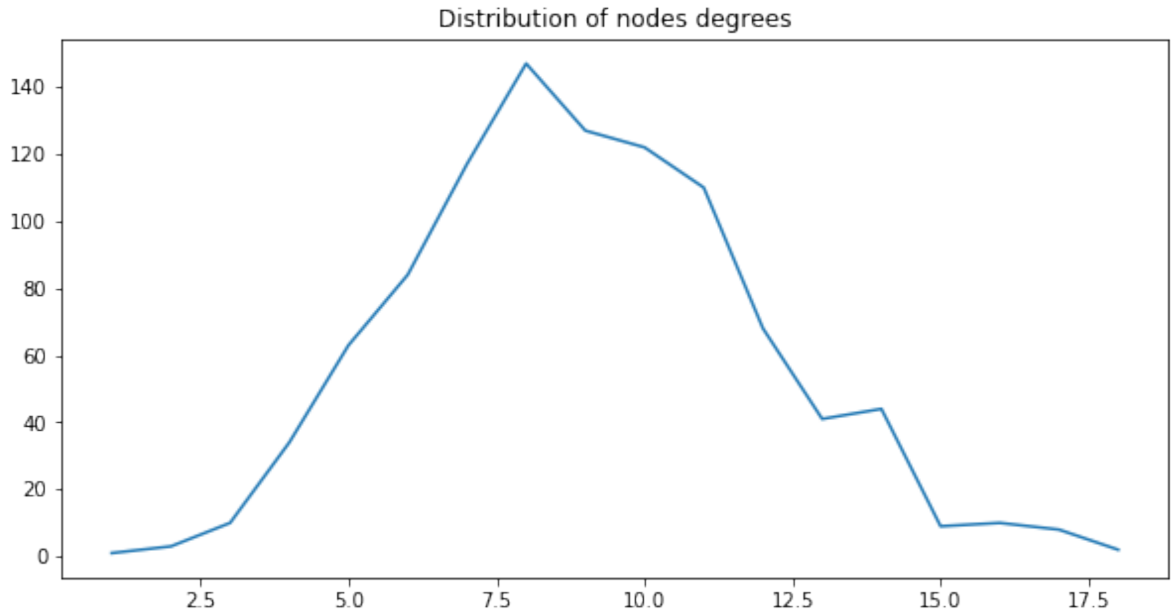
Question 8 (c)

In [47]:
```
dict_degrees_GERR=dict(GERR.degree())
list_degrees_GERR=dict_degrees_GERR.values()
print 'Mean degree of the nodes of the GERR:', np.mean(list_degrees
_GERR)
```

```
Mean degree of the nodes of the GERR: 8.992
```

In [49]:
```
dict_degrees_GERR=dict(GERR.degree())
list_degrees_GERR=sorted(dict_degrees_GERR.values())

dico_degrees_distribution_GERR={}
for i in list_degrees_GERR:
    if i not in dico_degrees_distribution_GERR:
        dico_degrees_distribution_GERR[i]=1
    else:
        dico_degrees_distribution_GERR[i]+=1
#dico_degrees_distribution_GERR
```

In [51]:
```
plt.figure(figsize=(10,5))
plt.plot(*zip(*sorted(dico_degrees_distribution_GERR.items())))
plt.title('Distribution of nodes degrees')
plt.show()
```



Question 9

(a) (i)

The produced Kronecker graph is connected, if:

$$\boxed{b + c > 1}$$

Here $b + c = 0.26 + 0.53 = 0.79$. So the produced Kronecker graph is NOT connected.

The graph has a giant connected component of size $\Theta(n)$ if:

$$\boxed{(a + b)(b + c) > 1}$$

Here $(a + b)(b + c) = (0.99 + 0.26)(0.26 + 0.53) = 0.9875$. So the produced Kronecker graph is NOT connected.

(b)

Similar structural properties :

- Global aspect of networks : nodes + edges
- Triangle participation distribution

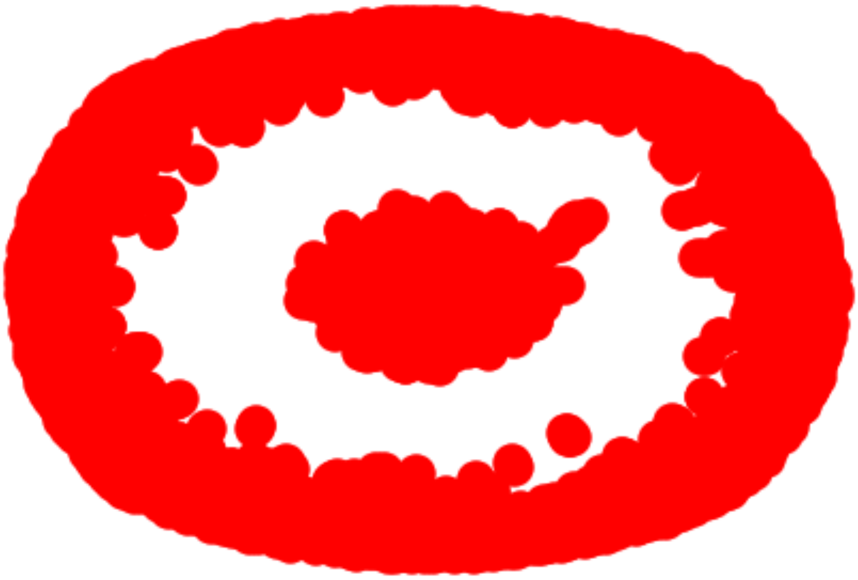
In [77]:
```
import numpy as np
A=np.asarray([[0.99, 0.26],[ 0.26, 0.53]])
B=np.kron(A,A)
for i in range(11):
    B=np.kron(B,A)
B
```

Out[77]:
```
array([[  8.77521023e-01,   2.30460067e-01,   2.30460067e-01, ...,
          9.44746671e-08,   9.44746671e-08,   2.48115287e-08],
       [  2.30460067e-01,   4.69783982e-01,   6.05248660e-02, ...,
          1.92582975e-07,   2.48115287e-08,   5.05773470e-08],
       [  2.30460067e-01,   6.05248660e-02,   4.69783982e-01, ...,
          2.48115287e-08,   1.92582975e-07,   5.05773470e-08],
       ...,
       [  9.44746671e-08,   1.92582975e-07,   2.48115287e-08, ...,
          4.86346315e-04,   6.26586829e-05,   1.27727315e-04],
       [  9.44746671e-08,   2.48115287e-08,   1.92582975e-07, ...,
          6.26586829e-05,   4.86346315e-04,   1.27727315e-04],
       [  2.48115287e-08,   5.05773470e-08,   5.05773470e-08, ...,
          1.27727315e-04,   1.27727315e-04,   2.60367219e-04]])
```

In [78]:
```
GCC_K = nx.Graph()
nodes = []
edges = []
kron_binary = B
for i in range(0,B.shape[0]):
    for j in range(0,B.shape[1]):
        rand_num = np.random.uniform(0,1)
        if B[i][j]>rand_num:
            kron_binary[i][j]=1
        else:
            kron_binary[i][j]=0
GCC_K = nx.from_numpy_matrix(kron_binary)
```

Global aspect of networks : nodes + edges

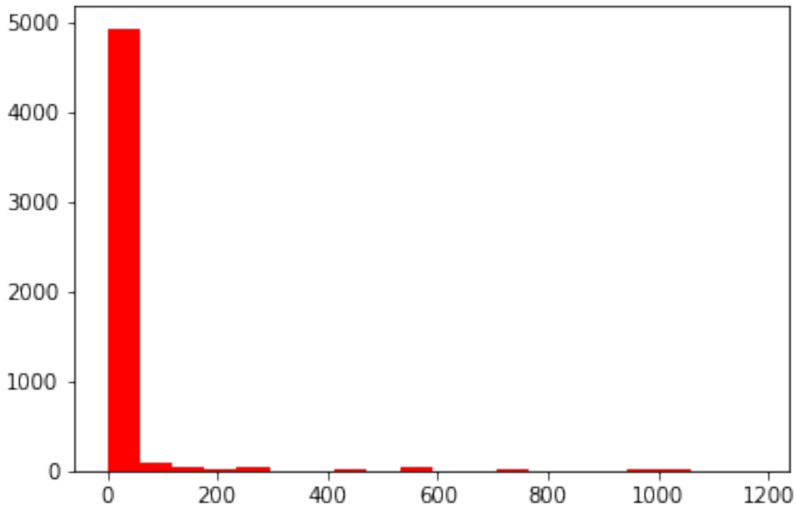
In [80]:
```
# plotting the graph and discovering similarity of shapes
## Kronecker graph
nx.draw(GCC_K)
```

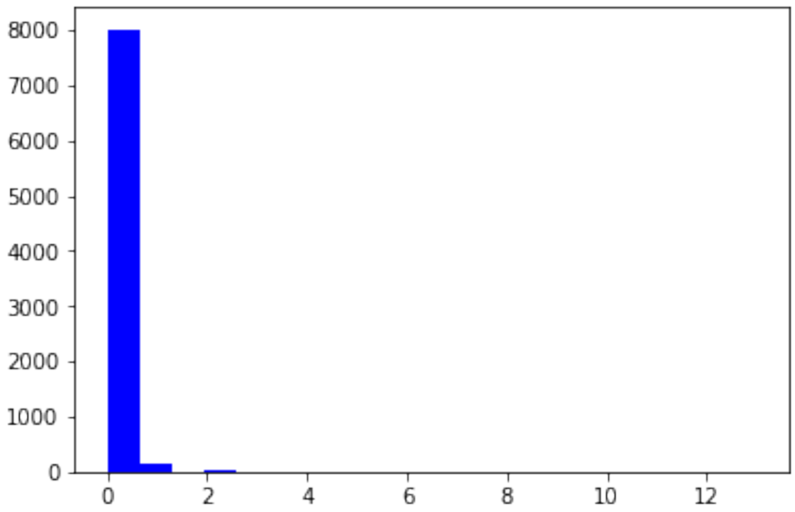


In [84]:
```
## Graph CA-GrQc
nx.draw(g)
```

Triangle participation distribution

```
In [88]: triangle_sequences = list(nx.triangles(g).values())
         triangle_sequences_K = list(nx.triangles(GCC_K).values())
         plt.hist(triangle_sequences,bins=20,color='red',label='Triangle par
         ticipation distribution of Kronecker Graph')
         plt.show()
         plt.hist(triangle_sequences_K,bins=20, color='blue', label='Triangl
         e participation distribution of Initial Graph')
         plt.show()
```





Question 10

(i) Random deletion: delete a randomly selected node.
(ii) Targeted deletion: delete a node chosen among the ones with the highest degree in the network.

1. Setting the relevant functions

In [54]:

```
#import math
#import random

# Get the numbers of nodes to remove from a graph based on the frac
tion of nodes we want to remove
def calculate_number_of_nodes_to_remove(graph,fraction_to_remove=0.
1):
    number_of_nodes = len(graph.nodes())
    result = int(math.floor(number_of_nodes*fraction_to_remove))
    return result

# Remove a specific number of nodes with RANDOM deletion
def remove_random_nodes(graph,number):
    #import random
    list_of_nodes = graph.nodes()
    # get a list of random nodes
    random_sample_to_remove = random.sample(list_of_nodes, number)
    # remove them from graph
    graph.remove_nodes_from(random_sample_to_remove)
    return graph

# Remove a specific number of nodes with TARGETED deletion
def remove_targeted_nodes(graph,number): #highest degree nodes
    my_dico_graph = dict(graph.degree())
    my_nodes_by_degree = sorted(my_dico_graph,key=lambda x: my_dico
_graph[x], reverse=True)
    # get the list of nodes sorted by degree descending
    targeted_sample_to_remove = my_nodes_by_degree[:number]
    # remove them from graph
    graph.remove_nodes_from(targeted_sample_to_remove)
    return graph

# Get the size of GCC and the size of the rest components :
#     - for a specific graph,
#     - after having removed specific number of nodes
#     - with specific deletion (random or targeted)

def calculate_new_sizes_GCC_and_Rest_Components(graph,removeMethod,
numberOfNodesToRemove):
        my_list_CC = []
        # remove nodes
        graph = removeMethod(graph,numberOfNodesToRemove)
        size_graph = len(graph.nodes())
        # list all CC of graphs
        my_list_CC = list(nx.connected_component_subgraphs(graph))
        # find GCC of the graph and its size
        graph_GCC = max(my_list_CC,key=len)
        size_GCC = len(graph_GCC.nodes())
        # remove GCC of the graph
        my_list_CC.remove(graph_GCC)
        # find size of the rest components :
        size_Rest_Components = size_graph - size_GCC
        return size_GCC,size_Rest_Components
```

In [55]:
```
#Example
H = GCC.copy()
total_Nodes_ex = len(H.nodes())
i = 20
size_GCC_ex,size_RC_ex = calculate_new_sizes_GCC_and_Rest_Component
s(H,remove_targeted_nodes,i)

print 'The graph has initially',total_Nodes_ex , 'nodes.'
print 'For', i, 'nodes to remove:'
print '- the new GCC has a size of', size_GCC_ex
print '- and the rest components have a total size of', size_RC_ex

print 'We can verify that we have:',i,'+',size_GCC_ex,'+',size_RC_e
x,'=',i+size_GCC_ex+size_RC_ex
```

```
The graph has initially 4158 nodes.
For 20 nodes to remove:
- the new GCC has a size of 4108
- and the rest components have a total size of 30
We can verify that we have: 20 + 4108 + 30 = 4158
```

In [59]:
```
# collect the size of GCC and the size of the rest components while
removing nodes in a range
def build_Dico_GCC_and_Dico_Rest_Components(H,remove_method,number_
of_nodes_to_remove):
    my_Dico_GCC={}
    my_Dico_Rest_Components={}
    for k in range(0,number_of_nodes_to_remove+1):
        H=GCC.copy()
        print k
        my_Dico_GCC[k],my_Dico_Rest_Components[k]=calculate_new_siz
es_GCC_and_Rest_Components(H,remove_method,k)
    return my_Dico_GCC, my_Dico_Rest_Components
```

2. Setting the parameters

In [60]:
```
H=GCC.copy()
fraction_to_remove = 0.20
number_of_nodes_to_remove = calculate_number_of_nodes_to_remove(H,f
raction_to_remove)
print 'We set the number of nodes to remove to:',number_of_nodes_to
_remove
```

```
We set the number of nodes to remove to: 831
```

3. Executing/Calculation

In [61]:
```
# dictionnaries for Random deletion
dico_GCC_targeted, dico_RC_targeted = build_Dico_GCC_and_Dico_Rest_
Components(H,remove_targeted_nodes,number_of_nodes_to_remove)
```

```
0
```

```
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
```

4. Plotting

In [63]:
```
X_GGC_Targeted = np.asarray(dico_GCC_targeted.keys())/4158.0
X_GGC_Random = np.asarray(dico_GCC_random.keys())/4158.0
X_RC_Targeted = np.asarray(dico_RC_targeted.keys())/4158.0
X_RC_Random = np.asarray(dico_RC_random.keys())/4158.0

Y_GGC_Targeted = np.asarray(dico_GCC_targeted.values())
Y_GGC_Random = np.asarray(dico_GCC_random.values())
Y_RC_Targeted = np.asarray(dico_RC_targeted.values())
Y_RC_Random = np.asarray(dico_RC_random.values())
```

In [64]:
```
plt.figure(figsize=(10,5))
plt.plot(X_GGC_Targeted, Y_GGC_Targeted,label='GCC, Targeted Deleti
on')
plt.plot(X_GGC_Random, Y_GGC_Random,label='GCC, Random Deletion')
plt.plot(X_RC_Targeted, Y_RC_Targeted,label='Comp, Targeted Deletio
n')
plt.plot(X_RC_Random, Y_RC_Random,label='Comp, Random Deletion')
plt.legend()
plt.title('GCC and rest components sizes vs. the fraction of delete
d nodes, for random or targeted strategies')
```

Out[64]:
```
Text(0.5,1,u'GCC and rest components sizes vs. the fraction of del
eted nodes, for random or targeted strategies')
```