NGSA: Network Science Analytics
Assignment 1

Joël Seytre

February 25, 2018

1 Graph Theory and Graph Properties

- **Question 1**
  Let $\boldsymbol{A}$ *be the adjacency matrix of an undirected graph (unweighted, with no self-loops) and* $\mathbf{1}$ *be the column vector whose elements are all 1. In terms of these quantities and simple matrix operations like matrix transpose and matrix trace, write expressions for:*

  (a) *The vector* $\boldsymbol{k}$ *whose elements are the degrees* k_i *of the nodes.*

  $$\boldsymbol{k} = \boldsymbol{A} * \mathbf{1}$$

  (b) *The number* m *of edges in the graph.*

  $$m = \frac{sum(\boldsymbol{A} * \mathbf{1})}{2} = \frac{\mathbf{1}^T * \boldsymbol{A} * \mathbf{1}}{2}$$

  (c) *The matrix* $\boldsymbol{N}$ *whose element* N_{ij} *is equal to the number of common neighbors of nodes* i *and* j*.*

  $$\boldsymbol{N} = \boldsymbol{A}^2$$

- **Question 2**

  Consider a bipartite network, with its two types of nodes (type 1 and 2), and suppose that there are n_1 nodes of type 1 and n_2 nodes of type 2. Show that the mean degrees c_1 and c_2 of the two types are related by

$$c_2 = \frac{n_1}{n_2} * c_1$$

  If we consider a bipartite network, then each edge has one vertex in each type of nodes.
  Let us call $\boldsymbol{N}$ the total number of edges in the graph. There are $\boldsymbol{N}$ edges connected to both types of nodes.
  By definition of the mean degrees c_1 and c_2 we have:

$$c_1 = \frac{N}{n_1} \text{ and } c_2 = \frac{N}{n_2}$$

  From that we can derive:

$$n_1 * c_1 = \boldsymbol{N} = n_2 * c_2$$

  i.e.

$$c_2 = \frac{n_1}{n_2} * c_1$$

- **Question 3**
  Let $G = (V, E)$ be an unweighted, undirected graph with no self-loops.
  The (i, j)-th element of $\boldsymbol{A}^l$ (i.e., the adjacency matrix raised to the l-th power) counts the number of paths of length l that start from node i and end at node j.
  A triangle in a graph corresponds to a clique of three nodes.

  (a) *Using simple matrix operations, express the total number of triangles in the graph $\Delta(G)$, as a function of the adjacency matrix $\boldsymbol{A}$.*

  We simply take the sum over all the diagonal elements of $\boldsymbol{A}^3$, but we remember to divide by 6 in order to not count the same triangle multiple times.
  The number 6 comes from the 3 choices of starting points times the 2 possible direction of going along the triangle.

  $$\Delta(G) = \frac{\text{Tr}(\boldsymbol{A}^3)}{6}$$

  Note: if $\boldsymbol{S}$ is a symetric matrix composed of 0s and 1s then $\text{Tr}(\boldsymbol{S}^2) = \text{sum}(\boldsymbol{S})$

  (b) *Similarly, express the total number of triangles in the graph $\Delta(G)$ as a function of the eigenvalues λ_i, $\forall i \in V$ of $\boldsymbol{A}$.*

  We know that for a given matrix $\boldsymbol{A}$ and its eigenvalues $(\lambda_i)_i$, $\text{Tr}(\boldsymbol{A}) = \sum \lambda_i$.
  We also know that if λ is an eigenvalue of $\boldsymbol{A}$ then λ^n is an eigenvalue of $\boldsymbol{A}^n$.
  We can derive:

  $$\Delta(G) = \frac{\sum \lambda_i^3}{6}$$

  (c) *Let Δ_i, $\forall i \in V$ be the number of triangles that node i participates in. Express Δ_i as a function of the spectrum (i.e., eigenvalues and/or eigenvectors) of the adjacency matrix $\boldsymbol{A}$.*

  The number of triangles that i participates in is the i-th diagonal component of $\boldsymbol{A}^3$ divided by 2 (2 possible directions to form triangle from a 3-way path from i to i).
  If we define $\mathbf{1}_i$ the the vector containing only 0s and a 1 in the i-th position, we have:

  $$\Delta_i = \frac{\mathbf{1}_i^T * A^3 * \mathbf{1}_i}{2}$$

  $\boldsymbol{A}$ is a symmetric matrix so we can diagonalize it (Spectral Theorem) as follows, where P is an orthogonal matrix of eigenvectors:

  $$\boldsymbol{A} = P^{-1} * \boldsymbol{D} * P = P^T * \boldsymbol{D} * P$$

  thus

  $$\Delta_i = \frac{\mathbf{1}_i^T * (P^T * \boldsymbol{D} * P)^3 * \mathbf{1}_i}{2} = \frac{(P * \mathbf{1}_i)^T * \boldsymbol{D}^{\mathbf{3}} * (P * \mathbf{1}_i)}{2} = \frac{V_i^T * \boldsymbol{D}^3 * V_i}{2}$$

  where $\boldsymbol{D}$ is the diagonal matrix of eigenvalues and V_i is the i-th eigenvector.

2 Graph Models

- **Question 4**
  Consider the random graph $G_{n,p}$ with average degree c.

  (a) *Show that in the limit of large n, the expected number of triangles in the graph is $\frac{1}{6}c^3$. In other words, show that the number of triangles is constant, neither growing nor vanishing in the limit of large n.*

  We have seen in the class Lecture 2A that for a random graph and large n:
  - For a given node there are on average c^2 nodes at distance 2. *(Slide 34)*
  - The probability of existing for a given edge is $p = \frac{c}{n-1} \approx \frac{c}{n}$. *(Slide 27)*

  The expected number of triangles is then:

$$\frac{\text{nodes in the graph x nodes at distance 2 x } \boldsymbol{P}(\text{last edge})}{\text{3 vertices x 2 directions}} = \frac{n * c^2 * \frac{c}{n}}{6} = \frac{c^3}{6}$$

  (b) *A connected triplet is defined as a triplet of nodes uvw, with edges (u, v) and (v, w) (the edge (u, w) can be present or not). Show that the expected number of connected triplets in the graph is $\frac{1}{2}n * c^2$.*

  Similarly we obtain the expected number of triplets:

$$\frac{\text{nodes in the graph x nodes at distance 2}}{\text{number of possible end vertices}} = \frac{n * c^2}{2}$$

  (c) *The clustering coefficient of a graph can also be expressed as*

$$C = \frac{\textit{(number of triangles)} * 3}{\textit{(number of connected triplets)}}$$

  Calculate the clustering coefficient of the $G_{n,p}$ random graph using the above formula based on (a) and (b), and confirm that for large n it agrees with the value shown in class (Lecture 2A; slide number 32).

  From the previous questions we can derive:

$$C = \frac{\frac{c3}{6} * 3}{\frac{n*c^2}{2}} = \frac{c}{n}$$

  which is the expected result from the Lecture 2A.

3 Centrality Criteria

- **Question 5**
  Suppose that we define a new centrality criterion $x_i, \forall i \in V$ to be a sum of contributions as follows: 1 for node i itself, α for each node at (geodesic) distance 1 from i, α^2 for each node at distance 2, and so forth, where α <1 is a given constant.

  (a) *Write an expression for x_i in terms of α and the geodesic distances d_{ij} between node pairs.*

  $$x_i = \sum_{j \in V} \alpha^{d_{ij}}$$

  (b) *Describe briefly (max 3 lines) an algorithm for computing this centrality measure. What is the complexity of calculating x_i for all $i \in V$?*

  We could proceed as follows:

  - Compute the pairwise shortest path for all (i, j) and store it in a matrix $\boldsymbol{M}$. This could be achieved with the Dijkstra algorithm;
  - For each column in $\boldsymbol{M}$ compute x_i as defined above.

  As seen in Lecture 3B Slide 14, the complexity would be that of Dijkstra's algorithm i.e. $O(n^2 logn + nm)$.

- **Question 6**
  Consider an undirected, unweighted graph of n nodes that is composed by exactly two sub-graphs of size n_A and n_B, which are connected by a single edge (A, B). Show that the closeness centralities C_A and C_B of nodes A and B respectively, are related by

$$\frac{1}{C_A} + \frac{n_A}{n} = \frac{1}{C_B} + \frac{n_B}{n}$$

Let G_A and G_B be the subgraphs of sizen_A and n_B. By definition, we have:

$$\frac{1}{C_A} = \frac{\sum_{j \in G} d_{Aj}}{n} = \frac{\sum_{j \in G_A} d_{Aj} + \sum_{j \in G_B} d_{Aj}}{n} = \frac{\sum_{j \in G_A} d_{Aj}}{n} + \frac{\sum_{j \in G_B} (1 + d_{Bj})}{n}$$

i.e.

$$\frac{1}{C_A} - \frac{n_B}{n} = \frac{\sum_{j \in G_A} d_{Aj}}{n} + \frac{\sum_{j \in G_B} (d_{Bj})}{n}$$

And we can similarly derive:

$$\frac{1}{C_B} - \frac{n_A}{n} = \frac{\sum_{j \in G_A} d_{Aj}}{n} + \frac{\sum_{j \in G_B} (d_{Bj})}{n}$$

Thus we have:

$$\frac{1}{C_A} - \frac{n_B}{n} = \frac{1}{C_B} - \frac{n_A}{n}$$

i.e.

$$\frac{1}{C_A} + \frac{n_A}{n} = \frac{1}{C_B} + \frac{n_B}{n}$$

4 Analyzing a Real Network

- **Question 7**

(a) Basic properties of the network:

1. Number of nodes: 5 242; number of edges: 14 496.
2. Number of CCs: 355. The distribution of the CCs' sizes can be seen in Figure 1.

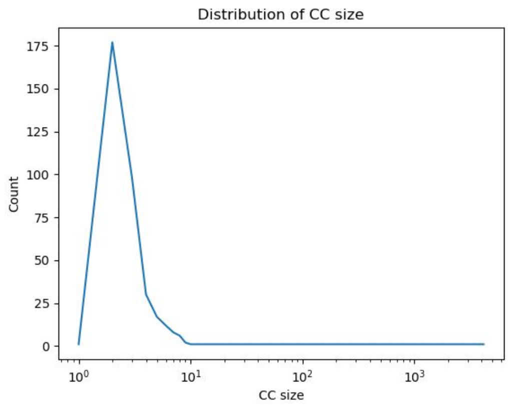


Figure 1: Size of connected components.

3. GCC: Number of nodes in the GCC: 4 158 (79.3% of total).
Number of edges: 13 428(92.6% of total).
We can see that almost all the edges are concentrated within the GCC, more so than the nodes (13.3% difference). We can see that there are a high number of 2-connected components from Figure 1: those count as 1 edge but 2 nodes, which explains the difference in percentages.

(b) Degree distribution:

- Minimum degree: 1;
- Maximum degree: 81;
- Median degree: 3;
- Mean degree: 5.5.

We can see that the median degree is quite smaller than the mean degree, which is understandable given the fact that the degree distribution is right-skewed.
We can see that there are nodes that are alone and that the most-connected node has only 81 connections out of 5 242. This is understandable since no scientist will have collaborated with a high percentage of the community.
Using the `powerlaw` Python library I obtained a power law with coefficient $\alpha = 2.09$. The degree distribution is plotted in Figure 2.

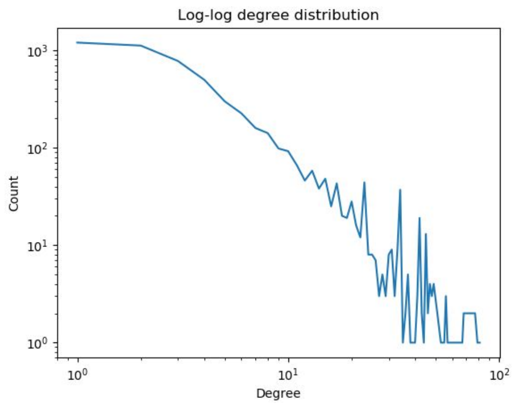


Figure 2: Degree distribution.

(c) *Triangles*
There are 47 779 triangles in the network.
The triangle participation is plotted in Figure 3.
It is interesting to see that there are 3 times more nodes that participate in 3 triangles than nodes that participate in 2 triangles (probably because if you participate in 2 separate triangles there is a high chance of participating in a 3rd composed of vertices of the first 2).

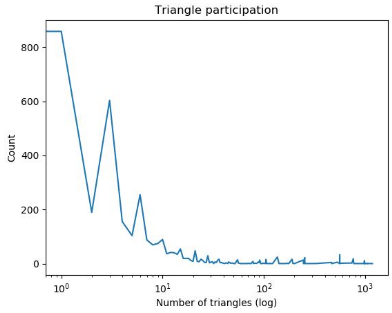


Figure 3: Triangle participation.

(d) *Spectral counting of triangles*
Finding eigenvalues involves inverting matrices and here we are manipulating quite large matrices ($\approx$ 5 000 x 5 000).
By looking at the eigenvalue distribution in Figure 4, we can see that the major contributions to $\frac{\sum_i \lambda_i^3}{6}$ will be contained in the first eigenvalues. Additionally, the eigenvalues tend to compensate each other starting around the 300-th eigenvalue.
We plot the respective errors and the computation times associated to various values of k in Figure 5. $k = 500$ seems like a good value.

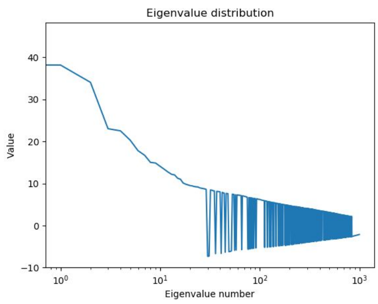


Figure 4: The eigenvalues of the adjacency matrix.

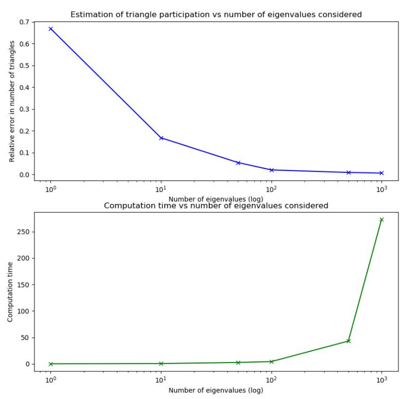


Figure 5: Triangle participation.

Code for question 7

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
import scipy
import powerlaw
from datetime import datetime

questions = [1,2,3]

G = nx.read_edgelist("CA-GrQc.txt")
connected_components = sorted(nx.connected_component_subgraphs(G),
                              key = len, reverse=True)
gcc = connected_components[0]
if 1 in questions:
    print("Number of nodes: %i" % G.number_of_nodes())
    print("Number of edges: %i" % G.number_of_edges())

    print("\nNumber of CC: %i" % nx.number_connected_components(G))

    cc_size = [g.number_of_nodes() for g in connected_components]
    values = sorted(np.unique(cc_size))
    counts = [cc_size.count(val) for val in values]

    please_plot = True
    if please_plot:
        plt.plot(values, counts)
        ax = plt.gca()
        ax.set_xscale('log')
        plt.title("Distribution of CC size")
        plt.xlabel("CC size")
        plt.ylabel("Count")
        plt.show()

    print("Number of nodes in GCC: %i i.e %.1f%% of total graph"
          % (gcc.number_of_nodes(),
             100*float(gcc.number_of_nodes())/float(G.number_of_nodes())))
    print("Number of edges in GCC: %i i.e %.1f%% of total graph"
          % (gcc.number_of_edges(),
             100*float(gcc.number_of_edges())/float(G.number_of_edges())))
elif 2 in questions:
    nodes = G.nodes
    degrees = [d for n, d in nx.degree(G)]
    print("Min degree: %i" % np.min(degrees))
    print("Max degree: %i" % np.max(degrees))
    print("Mean degree: %.1f" % np.mean(degrees))
    print("Median degree: %i" % np.median(degrees))

    values = sorted(np.unique(degrees))
    counts = [degrees.count(val) for val in values]
    fit = powerlaw.Fit(degrees, discrete=True)
    print('Power law! alpha= ', fit.power_law.alpha, ' - sigma= ', fit.power_law.sigma)
    please_plot = True
    if please_plot:
        plt.plot(values, counts)
        ax = plt.gca()
        ax.set_xscale('log')
        ax.set_yscale('log')
        plt.title("Log-log degree distribution")
        plt.xlabel("Degree")
        plt.ylabel("Count")
        plt.show()
elif 3 in questions:
    to_remove = []
    for edge in gcc.edges:
        if edge[0] == edge[1]:
            to_remove += [edge[0]]
    for n in to_remove:
        gcc.remove_edge(n, n)
    A = np.matrix(nx.to_numpy_matrix(gcc))
    B = A ** 3
    num_triangles = int(np.trace(B)/6)
    print("Total number of triangles: %i" % num_triangles)
```

```
triangle_participation = [B[i, i]/2 for i in range(len(B))]
values = sorted(np.unique(triangle_participation))
counts = [triangle_participation.count(val) for val in values]

please_plot = False
if please_plot:
    plt.plot(values, counts)
    ax = plt.gca()
    ax.set_xscale('log')
    plt.title("Triangle participation")
    plt.xlabel("Number of triangles (log)")
    plt.ylabel("Count")
    plt.show()

eigenvalues, eigenvectors = np.array(scipy.sparse.linalg.eigs(A, 1000))
eigenvalues = eigenvalues.real
plt.plot(eigenvalues)
ax = plt.gca()
ax.set_xscale('log')
plt.title("Eigenvalue distribution")
plt.xlabel("Eigenvalue number")
plt.ylabel("Value")
plt.show()

ready_for_long_computation = False
if ready_for_long_computation:
    num_eig = [1, 10, 50, 100, 500, 1000]
    errors = []
    delays = []
    for k in num_eig:
        start = datetime.now()
        eigenvalues, eigenvectors = np.array(scipy.sparse.linalg.eigs(A, k))
        eigenvalues = eigenvalues.real
        errors += [abs((np.sum(np.power(eigenvalues, 3)) / 6) - num_triangles)
                   / num_triangles]
        time_computation = datetime.now() - start
        delays += [time_computation.minutes
                   + time_computation.seconds
                   + time_computation.microseconds / 1000000]
    plt.subplot(211)
    plt.plot(num_eig, errors, '-bx')
    ax = plt.gca()
    ax.set_xscale('log')
    plt.title("Estimation of triangle participation vs number of eigenvalues")
    plt.xlabel("Number of eigenvalues (log)")
    plt.ylabel("Relative error in number of triangles")

    plt.subplot(212)
    plt.plot(num_eig, delays, '-gx')
    ax = plt.gca()
    ax.set_xscale('log')
    plt.title("Computation time vs number of eigenvalues considered")
    plt.xlabel("Number of eigenvalues (log)")
    plt.ylabel("Computation time")
    plt.show()
```

- **Question 8**

(a) In Lecture 2A (Slide 27) we saw that $p = \frac{c}{n-1}$ so the mean degree should be ≈ 8.9.

(b) We saw in Lecture 2A (Slide 45) that the graph is connected if $c > ln(n)$.
Here $ln(1000) = 6.9$ so the graph is connected.

(c) The computed mean degree of the graph is 8.93. The degree distribution is plotted in Figure 6 and does look like a Poisson distribution with parameter c (see Figure 7 which was computed from https://homepage.divms.uiowa.edu/~mbognar/applets/pois.html).

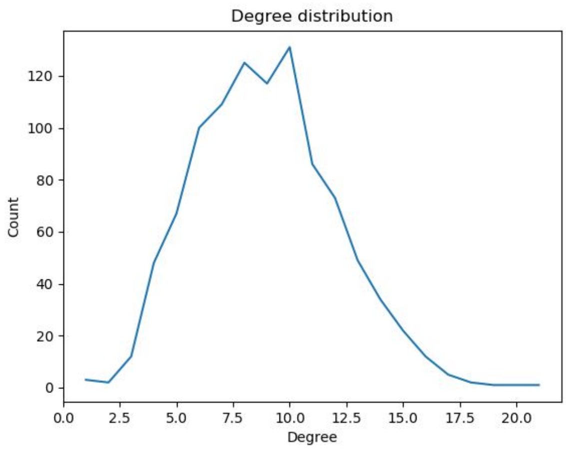


Figure 6: Degree distribution.

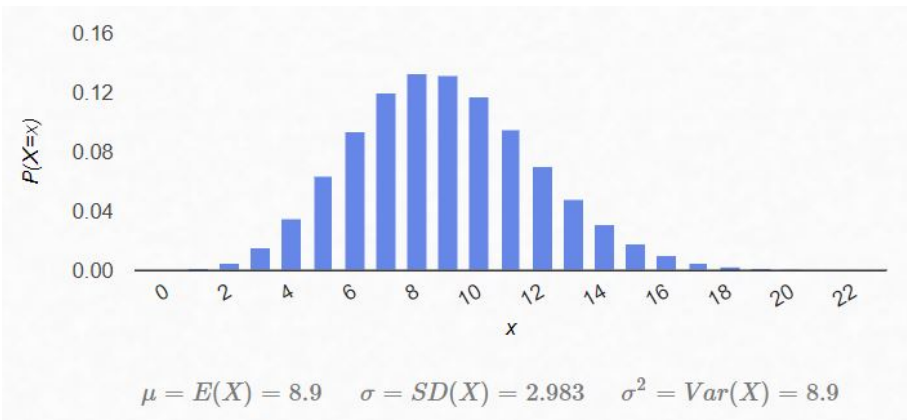


Figure 7: Poisson distribution of parameter $c = 8.9$.

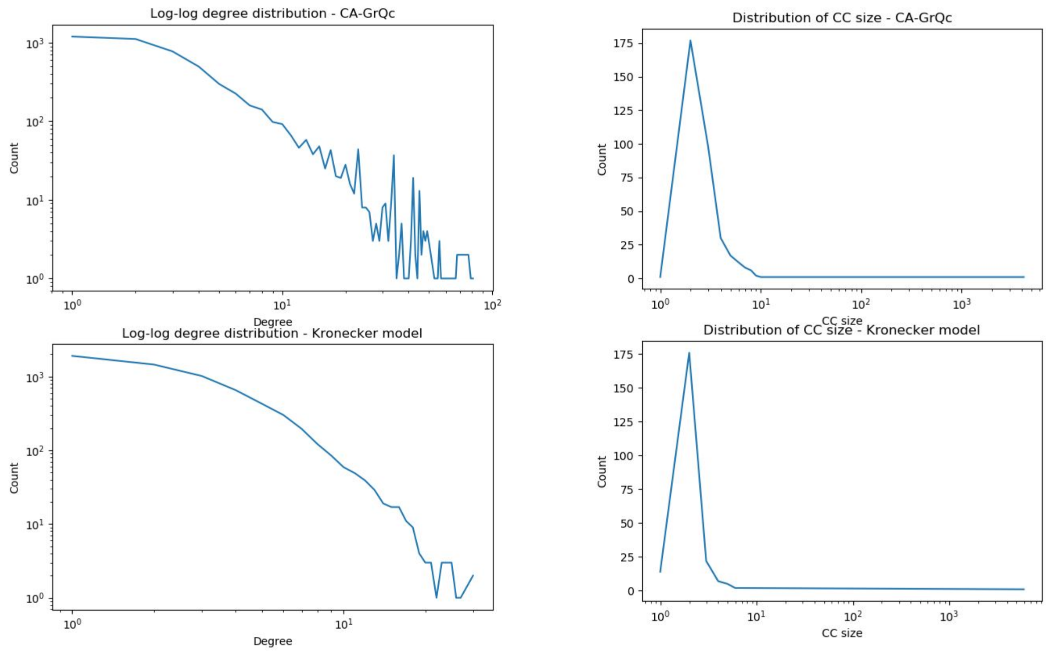
Code for question 8

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

n = 1000
p = 0.009
G = nx.fast_gnp_random_graph(n, p)
degrees = [d for n, d in nx.degree(G)]

print("Mean degree: %.2f" % np.mean(degrees))

values = sorted(np.unique(degrees))
counts = [degrees.count(val) for val in values]

please_plot = True
if please_plot:
    plt.plot(values, counts)
    ax = plt.gca()
    plt.title("Degree distribution")
    plt.xlabel("Degree")
    plt.ylabel("Count")
    plt.show()
```

- **Question 9**
While building the Kronecker graph, I chose to take k = 13 and end up with up to 8 192 nodes potentially. Naturally, some of these nodes will end up not existing (if all their edges are picked as 0): I ended up with 6 440 nodes whereas the real network has 5 242 nodes[1].

  (a) The produced Kronecker graph is not connected because $0.26 + 0.53 < 1$ and the GCC is not of size $\Theta(n)$ because $(0.99 + 0.26)(0.26 + 0.53) = 0.9875 < 1$.
  Those rules are extracted from Lecture 3A Slide 44 but we can for example see that if $b + c < 1$ then the degree probability of the last node (last row of the matrix) is $(b + c)^k \to 0$.
  Note: see Mahdian and Xu '07 for more complete proof, section 2.2 and 2.3.

  (b) We plot below in Figure 8 the degree distribution as well as the size of the connected components.
  We can see that the degree distribution isn't as granular but has a very similar shape. The distribution of the connected components is very close as well.
  Lastly, the size of the connected components of size 2 is the same (value of the peak of Figure 8b), while the share of the GCC is similar (97.3% of the edges in the Kronecker model vs 92.6% in the real network).



(a) Degree distribution

(b) Size of connected components.

Figure 8: Comparing the Kronecker model with the real `CA-GrQc` graph.

[1] I could have added the nodes with 0 edges as nodes of degree 0 but decided against it, given that the original network doesn't have any such node. Indeed, the original graph is constructed solely on edges.

Code for question 9

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from datetime import datetime
from tqdm import tqdm


def generate_graph(M):
    g = nx.Graph()
    n = len(M)
    for l in tqdm(range(n)):
        for j in range(n - i):
            if np.random.choice([False, True], p=[1 - M[l, j], M[l, j]]):
                g.add_edge(l, j)
    return g.to_undirected()


A_1 = np.array([[0.99, 0.26], [0.26, 0.53]])

# k = 11 => 4096 nodes
k = 12

start = datetime.now()
A = A_1
for i in range(k):
    A = np.kron(A, A_1)
time_computation = datetime.now() - start
print("Computation time: %s" % time_computation)

start = datetime.now()
k_G = generate_graph(A)
time_computation = datetime.now() - start
print("Reading time: %s" % time_computation)

connected_components_k_G = sorted(nx.connected_component_subgraphs(k_G),
                                  key = len, reverse=True)
gcc_k_G = connected_components_k_G[0]

print("Number of edges in GCC (Kro. model): %i i.e %.1f%% of total graph (%i edges)"
      % (gcc_k_G.number_of_edges(), 100 * float(gcc_k_G.number_of_edges())
         / float(k_G.number_of_edges()), k_G.number_of_edges()))

print("Number of nodes in GCC (Kro. model): %i i.e %.1f%% of total graph (%i nodes)"
      % (gcc_k_G.number_of_nodes(), 100 * float(gcc_k_G.number_of_nodes())
         / float(k_G.number_of_nodes()), k_G.number_of_nodes()))

degrees_k_G = [d for n, d in nx.degree(k_G)]
values_k_G = sorted(np.unique(degrees_k_G))
counts_k_G = [degrees_k_G.count(val) for val in values_k_G]

cc_size_k_G = [g.number_of_nodes() for g in connected_components_k_G]
values_cc_k_G = sorted(np.unique(cc_size_k_G))
counts_cc_k_G = [cc_size_k_G.count(val) for val in values_cc_k_G]

G = nx.read_edgelist("CA-GrQc.txt")
connected_components_G = sorted(nx.connected_component_subgraphs(G),
                                key = len, reverse=True)
gcc_G = connected_components_G[0]

degrees_G = [d for n, d in nx.degree(G)]
values_G = sorted(np.unique(degrees_G))
counts_G = [degrees_G.count(val) for val in values_G]
```

```
cc_size_G = [g.number_of_nodes() for g in connected_components_G]
values_cc_G = sorted(np.unique(cc_size_G))
counts_cc_G = [cc_size_G.count(val) for val in values_cc_G]

please_plot = True
if please_plot:
    plt.subplot(211)
    plt.plot(values_G, counts_G)
    ax = plt.gca()
    ax.set_xscale('log')
    ax.set_yscale('log')
    plt.title("Log-log degree distribution - CA-GrQc")
    plt.xlabel("Degree")
    plt.ylabel("Count")

    plt.subplot(212)
    plt.plot(values_k_G, counts_k_G)
    ax = plt.gca()
    ax.set_xscale('log')
    ax.set_yscale('log')
    plt.title("Log-log degree distribution - Kronecker model")
    plt.xlabel("Degree")
    plt.ylabel("Count")
    plt.show()

    plt.subplot(211)
    plt.plot(values_cc_G, counts_cc_G)
    ax = plt.gca()
    ax.set_xscale('log')
    plt.title("Distribution of CC size - CA-GrQc")
    plt.xlabel("CC size")
    plt.ylabel("Count")

    plt.subplot(212)
    plt.plot(values_cc_k_G, counts_cc_k_G)
    ax = plt.gca()
    ax.set_xscale('log')
    plt.title("Distribution of CC size - Kronecker model")
    plt.xlabel("CC size")
    plt.ylabel("Count")
    plt.show()
```

- **Question 10** My results are displayed on Figure 9.
  We can see that an attack has a bigger impact than a failure, in terms of splitting up the GCC into isolated components.

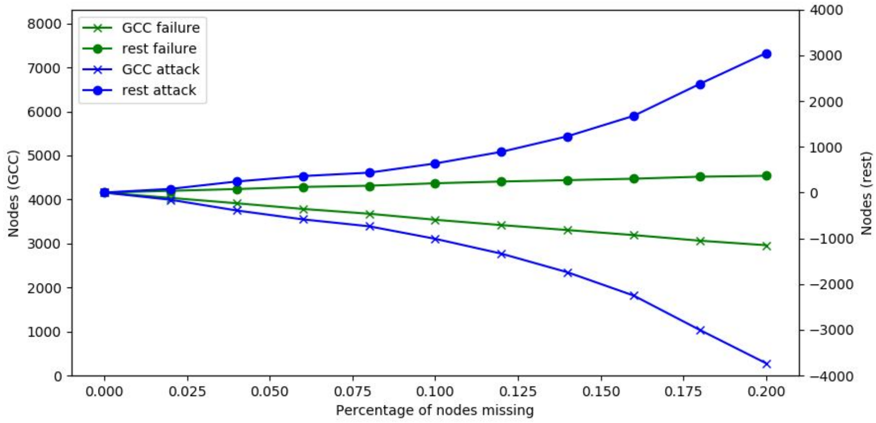


Figure 9: Comparing the impact of loss of nodes on the GCC and the rest of the nodes, in case of an attack and a failure scenario.

Code for question 10

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import random

G = nx.read_edgelist("CA-GrQc.txt")
G = max(nx.connected_component_subgraphs(G), key=len)

nodes = [n for n, d in nx.degree(G)]
degrees = [d for n, d in nx.degree(G)]

# sorts the nodes based on degrees
nodes = [n for _, n in sorted(zip(degrees, nodes), reverse=True)]
n = len(nodes)

# keep 20%
large_nodes = nodes[:int(0.2*n)]

G1 = G.copy()
G2 = G.copy()

starting_length = len(nodes)
current_length = starting_length
step = int(0.02 * starting_length)
GCCs_failure = []
rests_failure = []
totals = []
while current_length >= 0.8 * starting_length:
    gcc = max(nx.connected_component_subgraphs(G1), key=len)
    GCCs_failure += [len(gcc.nodes)]
    rests_failure += [len(nodes) - len(gcc.nodes)]
    totals += [len(nodes)]

    to_remove = random.sample(nodes, k=step)
    for node in to_remove:
        G1.remove_node(node)
    nodes = [n for n, d in nx.degree(G1)]
    current_length = len(nodes)

GCCs_attack = []
rests_attack = []
steps = []
current_steps = 0
current_length = starting_length
nodes = [n for n, d in nx.degree(G2)]
while len(large_nodes) > 0:
    gcc = max(nx.connected_component_subgraphs(G2), key=len)
    GCCs_attack += [len(gcc.nodes)]
    rests_attack += [len(nodes) - len(gcc.nodes)]
    totals += [len(nodes)]
    steps += [current_steps]
    current_steps += 0.02

    to_remove = random.sample(large_nodes, k=min(step, len(large_nodes)))
    for node in to_remove:
        G2.remove_node(node)
        large_nodes.remove(node)
    nodes = [n for n, d in nx.degree(G2)]
    current_length = len(nodes)

ax1 = plt.gca()
plt.ylabel("Nodes (GCC)")
plt.xlabel("Percentage of nodes missing")
ax2 = ax1.twinx()
plt.ylabel("Nodes (rest)")
ax1.set_ylim([0, 2*GCCs_attack[0]])
ax2.set_ylim([-4000, 4000])
g_f, = ax1.plot(steps, GCCs_failure, '-xg', label='GCC failure')
r_f, = ax2.plot(steps, rests_failure, '-og', label='rest failure')
g_a, = ax1.plot(steps, GCCs_attack, '-xb', label='GCC attack')
r_a, = ax2.plot(steps, rests_attack, '-ob', label='rest attack')
plt.legend(handles=[g_f, r_f, g_a, r_a])
plt.show()
```