INA - homework 2

Matic Isovski, 63180442

March 2022

Homework #2

This homework is complete and will not be changed. The homework does not require a lot of writing, but may require some thinking. It does not require a lot of processing power, but may require efficient programming. It accounts for 10% of the course grade. Any questions and comments regarding the homework should be directed to Piazza.

Submission details

This homework is due on **April 1st** at 11:59pm, while late days expire on **April 4th** at 11:00am. The homework must be submitted through (1) Gradescope (entry code **6PDZD3**) and (2) eUcilnica. (1) Submission to Gradescope should include answers to all questions, each on a separate page, which may also demand pseudocode, proofs, tables, plots, diagrams and other. (2) Submission to eUcilnica should include at least this cover sheet with signed honor code and all the programming code used to complete the exercises. The homework is considered submitted only when *both* parts have been submitted. Failing to include this honor code in the submission will result in **10% deduction**. Failing to submit all the developed code will result in **50% deduction**.

Honor code

Students are strongly encouraged to discuss the homework with other classmates and form study groups. Yet, each student must then solve the homework by itself without the help of others and should be able to redo the homework at a later time. In other words, students are encouraged to collaborate, but should not copy from one another. Referring to any solutions obtained from classmates, course books, previous years, found online or other material, is considered an honor code violation. Also, stating any part of the solutions in class or on Piazza is considered an honor code violation. Finally, failing to name the correct study group members, or filling out the wrong date or time of the submission, is also considered an honor code violation. Honor code violation will not be tolerated! Any student violating the honor code will be reported to **faculty disciplinary committee** and vice dean for education.

Name & SID: Matic Isovski tzXs80Wu2yUQE9mgRVJojB7O6GlnZh5edHMTDaxL

Study group: ____________________

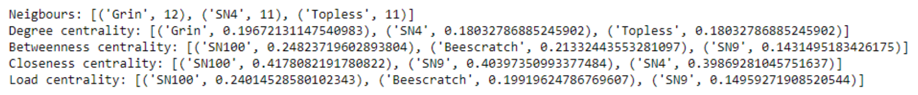
Date & time: 01.04.2022 17:37

I acknowledge and accept the honor code.

Signature: Matic

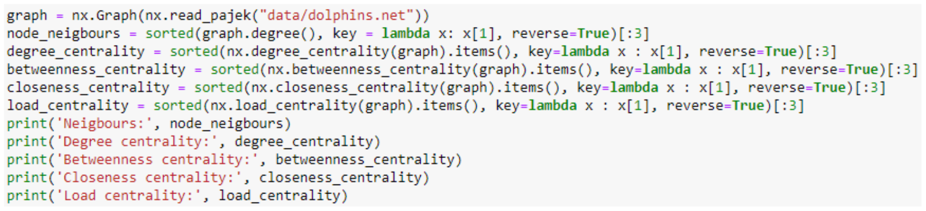
1 Where is SN100?

For the importance of node I calculated number of neigbours, degree centrality, betweenness centrality, cloceness centrality and load centrality for each node. 3 out of 5 found SN100 as the most important node (betweenness centrality, cloceness centrality and load centrality). We can see all 5 ranks in Figure 1.



```
Neigbours: [('Grin', 12), ('SN4', 11), ('Topless', 11)]
Degree centrality: [('Grin', 0.19672131147540983), ('SN4', 0.18032786885245902), ('Topless', 0.18032786885245902)]
Betweenness centrality: [('SN100', 0.24823719602893804), ('Beescratch', 0.21332443553281097), ('SN9', 0.143149518342617 5)]
Closeness centrality: [('SN100', 0.4178082191780822), ('SN9', 0.40397350993377484), ('SN4', 0.3986928104575163 7)]
Load centrality: [('SN100', 0.24014528580102343), ('Beescratch', 0.19919624786769607), ('SN9', 0.14959271908520544)]
```

Figure 1: Node importance ranks.



```
graph = nx.Graph(nx.read_pajek("data/dolphins.net"))
node_neigbours = sorted(graph.degree(), key = lambda x: x[1], reverse=True)[:3]
degree_centrality = sorted(nx.degree_centrality(graph).items(), key=lambda x : x[1], reverse=True)[:3]
betweenness_centrality = sorted(nx.betweenness_centrality(graph).items(), key=lambda x : x[1], reverse=True)[:3]
closeness_centrality = sorted(nx.closeness_centrality(graph).items(), key=lambda x : x[1], reverse=True)[:3]
load_centrality = sorted(nx.load_centrality(graph).items(), key=lambda x : x[1], reverse=True)[:3]
print('Neigbours:', node_neigbours)
print('Degree centrality:', degree_centrality)
print('Betweenness centrality:', betweenness_centrality)
print('Closeness centrality:', closeness_centrality)
print('Load centrality:', load_centrality)
```

Figure 2: Code printout.

2 Is software scale-free?

In Figure 3 we can see that only in-degree distributions of both networks are power-law, hence scale-free, while in/out and out degree sitributions are more of polynomial. This is quite expected since libreries are following the decorator pattern. Base libraries represent hubs in a graph. Result for Lucene in-degree is $\gamma = 1.75$ and $\gamma = 1.78$ for Java in-degree, using $k_{min} = 3$. All gamma values can be seen in Figure 4.

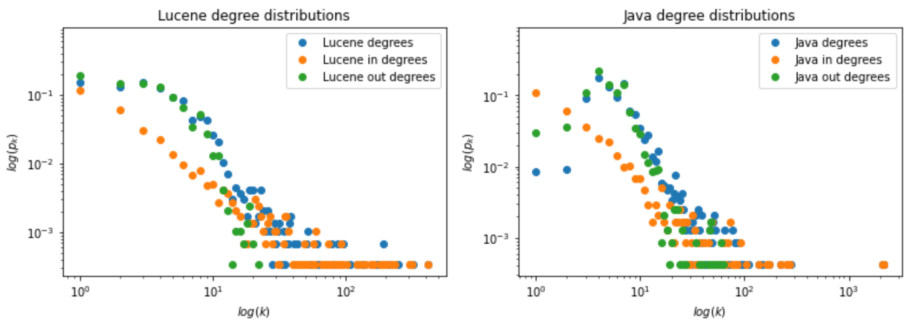
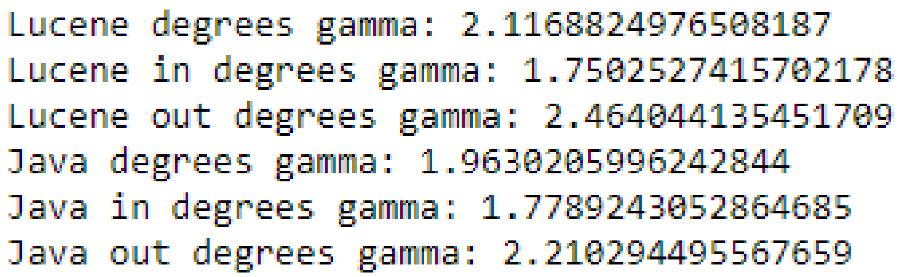
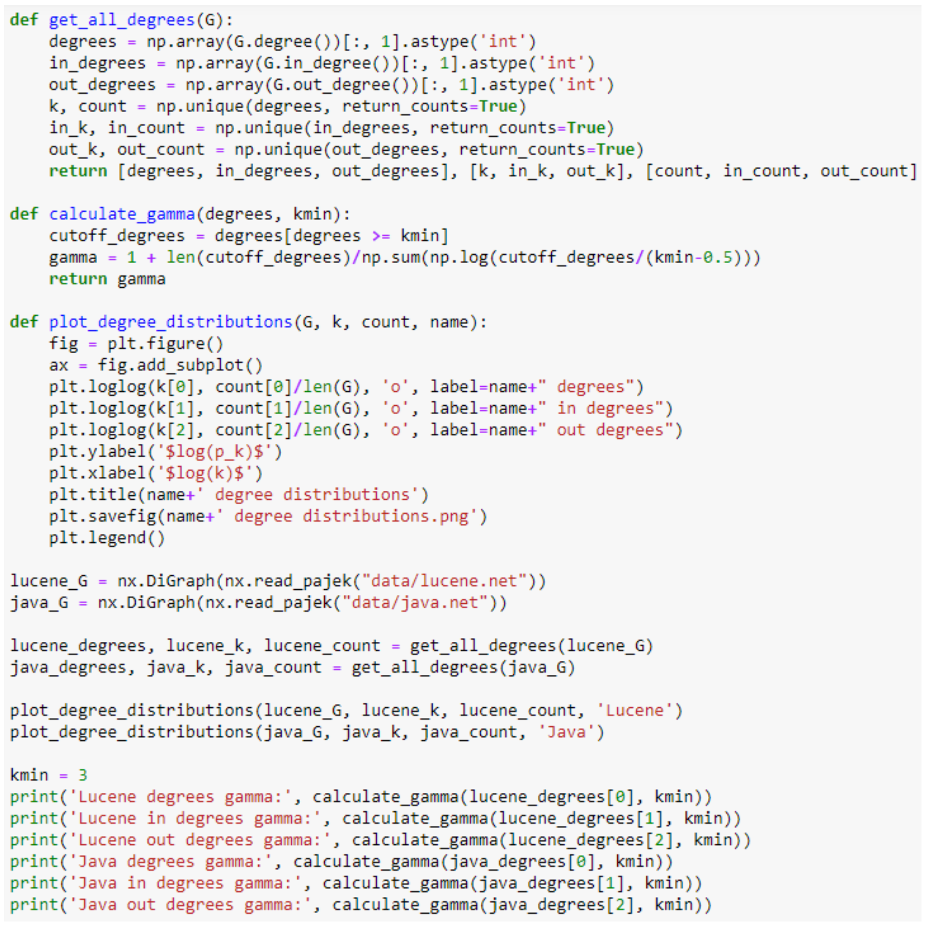


Figure 3: Degree distributions for Lucene (left) and Java (right) graphs.



```
Lucene degrees gamma: 2.1168824976508187
Lucene in degrees gamma: 1.7502527415702178
Lucene out degrees gamma: 2.464044135451709
Java degrees gamma: 1.9630205996242844
Java in degrees gamma: 1.7789243052864685
Java out degrees gamma: 2.210294495567659
```

Figure 4: Gamma results.



```python
def get_all_degrees(G):
    degrees = np.array(G.degree())[:, 1].astype('int')
    in_degrees = np.array(G.in_degree())[:, 1].astype('int')
    out_degrees = np.array(G.out_degree())[:, 1].astype('int')
    k, count = np.unique(degrees, return_counts=True)
    in_k, in_count = np.unique(in_degrees, return_counts=True)
    out_k, out_count = np.unique(out_degrees, return_counts=True)
    return [degrees, in_degrees, out_degrees], [k, in_k, out_k], [count, in_count, out_count]

def calculate_gamma(degrees, kmin):
    cutoff_degrees = degrees[degrees >= kmin]
    gamma = 1 + len(cutoff_degrees)/np.sum(np.log(cutoff_degrees/(kmin-0.5)))
    return gamma

def plot_degree_distributions(G, k, count, name):
    fig = plt.figure()
    ax = fig.add_subplot()
    plt.loglog(k[0], count[0]/len(G), 'o', label=name+" degrees")
    plt.loglog(k[1], count[1]/len(G), 'o', label=name+" in degrees")
    plt.loglog(k[2], count[2]/len(G), 'o', label=name+" out degrees")
    plt.ylabel('$log(p_k)$')
    plt.xlabel('$log(k)$')
    plt.title(name+' degree distributions')
    plt.savefig(name+' degree distributions.png')
    plt.legend()

lucene_G = nx.DiGraph(nx.read_pajek("data/lucene.net"))
java_G = nx.DiGraph(nx.read_pajek("data/java.net"))

lucene_degrees, lucene_k, lucene_count = get_all_degrees(lucene_G)
java_degrees, java_k, java_count = get_all_degrees(java_G)

plot_degree_distributions(lucene_G, lucene_k, lucene_count, 'Lucene')
plot_degree_distributions(java_G, java_k, java_count, 'Java')

kmin = 3
print('Lucene degrees gamma:', calculate_gamma(lucene_degrees[0], kmin))
print('Lucene in degrees gamma:', calculate_gamma(lucene_degrees[1], kmin))
print('Lucene out degrees gamma:', calculate_gamma(lucene_degrees[2], kmin))
print('Java degrees gamma:', calculate_gamma(java_degrees[0], kmin))
print('Java in degrees gamma:', calculate_gamma(java_degrees[1], kmin))
print('Java out degrees gamma:', calculate_gamma(java_degrees[2], kmin))
```

Figure 5: Code printout.

3 Errors and attacks on the Internet

From plots in Figure 6 we can cleary see, that the Internet network is not robust to malicious attacks, while Erdos Renyi random graph is more robust. On the other hand, both graphs are robust to random attacks. The reason why is that the Internet network follows the power-law distribution of degree, so if we remove hubs, we disconect the graph. But Erdos Renyi random graph has no central hubs, so we are not removing a lot of connections when removing random nodes. When performing malicious attack, probability of selecting a node is proportional to its degree, while in random attack each node has the same probability to be choosen.

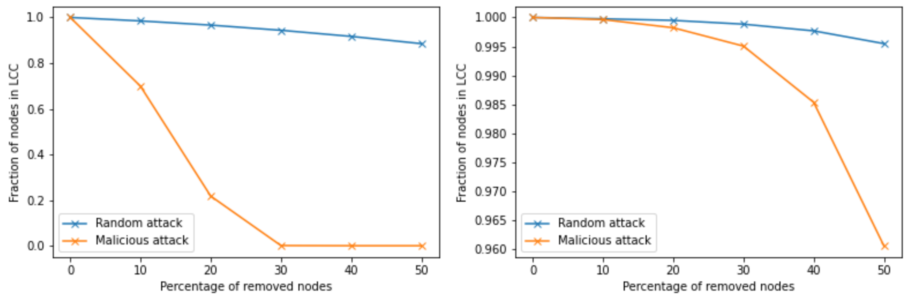
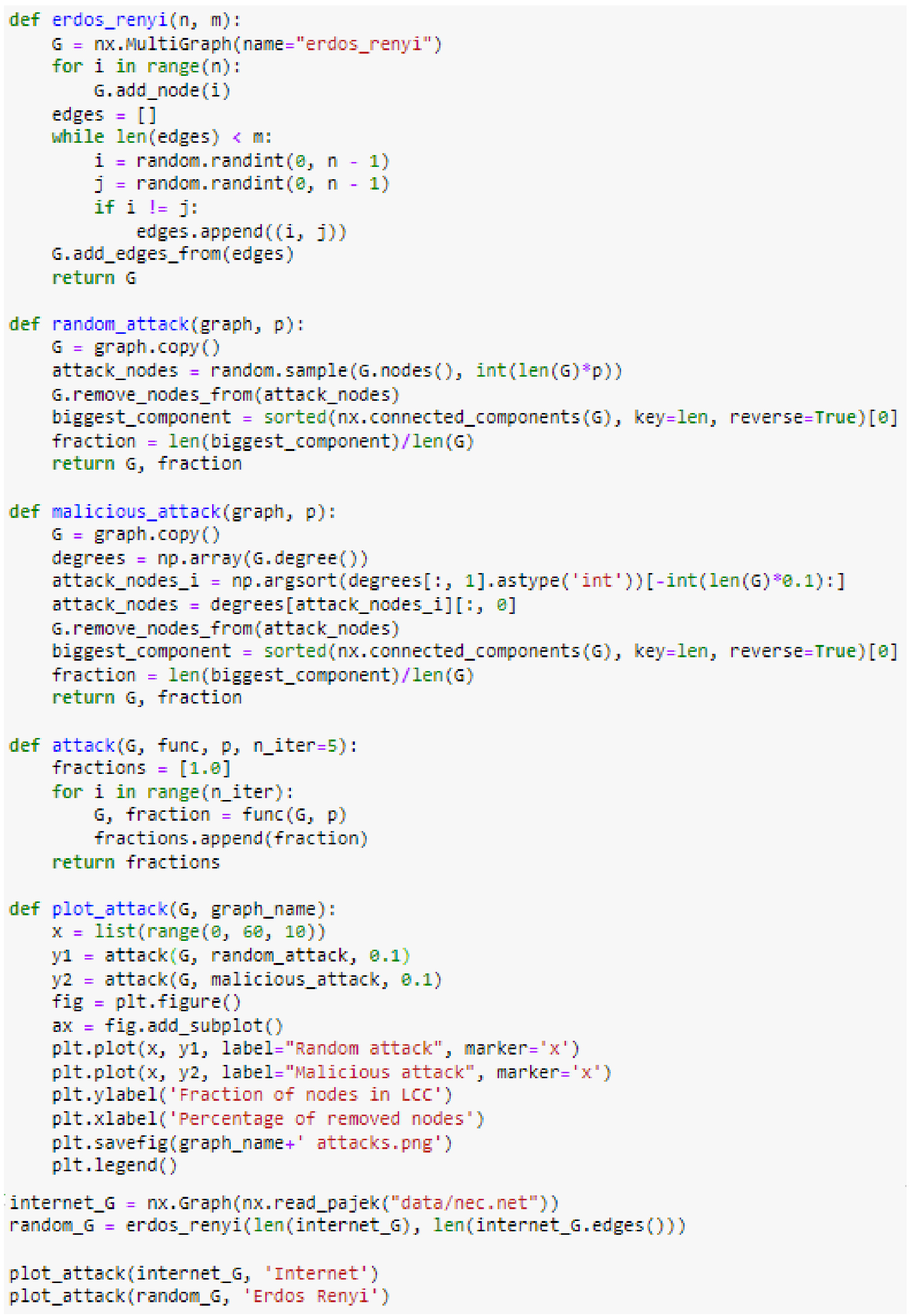


Figure 6: Robustness of the Internet (left) and Erdos Renyi graph (right).



```python
def erdos_renyi(n, m):
    G = nx.MultiGraph(name="erdos_renyi")
    for i in range(n):
        G.add_node(i)
    edges = []
    while len(edges) < m:
        i = random.randint(0, n - 1)
        j = random.randint(0, n - 1)
        if i != j:
            edges.append((i, j))
    G.add_edges_from(edges)
    return G

def random_attack(graph, p):
    G = graph.copy()
    attack_nodes = random.sample(G.nodes(), int(len(G)*p))
    G.remove_nodes_from(attack_nodes)
    biggest_component = sorted(nx.connected_components(G), key=len, reverse=True)[0]
    fraction = len(biggest_component)/len(G)
    return G, fraction

def malicious_attack(graph, p):
    G = graph.copy()
    degrees = np.array(G.degree())
    attack_nodes_i = np.argsort(degrees[:, 1].astype('int'))[-int(len(G)*0.1):]
    attack_nodes = degrees[attack_nodes_i][:, 0]
    G.remove_nodes_from(attack_nodes)
    biggest_component = sorted(nx.connected_components(G), key=len, reverse=True)[0]
    fraction = len(biggest_component)/len(G)
    return G, fraction

def attack(G, func, p, n_iter=5):
    fractions = [1.0]
    for i in range(n_iter):
        G, fraction = func(G, p)
        fractions.append(fraction)
    return fractions

def plot_attack(G, graph_name):
    x = list(range(0, 60, 10))
    y1 = attack(G, random_attack, 0.1)
    y2 = attack(G, malicious_attack, 0.1)
    fig = plt.figure()
    ax = fig.add_subplot()
    plt.plot(x, y1, label="Random attack", marker='x')
    plt.plot(x, y2, label="Malicious attack", marker='x')
    plt.ylabel('Fraction of nodes in LCC')
    plt.xlabel('Percentage of removed nodes')
    plt.savefig(graph_name+' attacks.png')
    plt.legend()

internet_G = nx.Graph(nx.read_pajek("data/nec.net"))
random_G = erdos_renyi(len(internet_G), len(internet_G.edges()))

plot_attack(internet_G, 'Internet')
plot_attack(random_G, 'Erdos Renyi')
```

Figure 7: Code printout.

4 HIV and network sampling

Both netwokrs are scale-free, because they have power-law degree distribution. For the network to be small-world $< d > \propto \frac{\log(n}{\log(<k>)}$ must stand and clustering coefficient must not be small, hence the original network is small-world, while sampled isn't. Computations result for both networks are shown in Figure 8, plots are shown in Figure 9.

```
        Graph | 'Social'
   Avg degree | 4.55 (205)
        Gamma | 2.06 (3)
Avg distance 1 | 7.5
Avg distance 2 | 6.1
   Clustering | 0.2659

        Graph | 'Sampled'
   Avg degree | 3.29 (25)
        Gamma | 2.52 (3)
Avg distance 1 | 8.8
Avg distance 2 | 5.9
   Clustering | 0.1356
```

Figure 8: Small-world info.

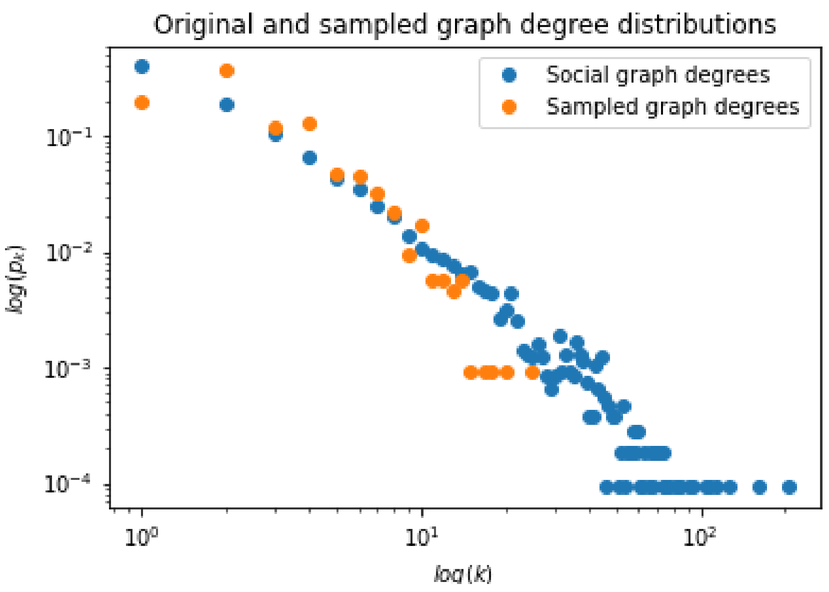
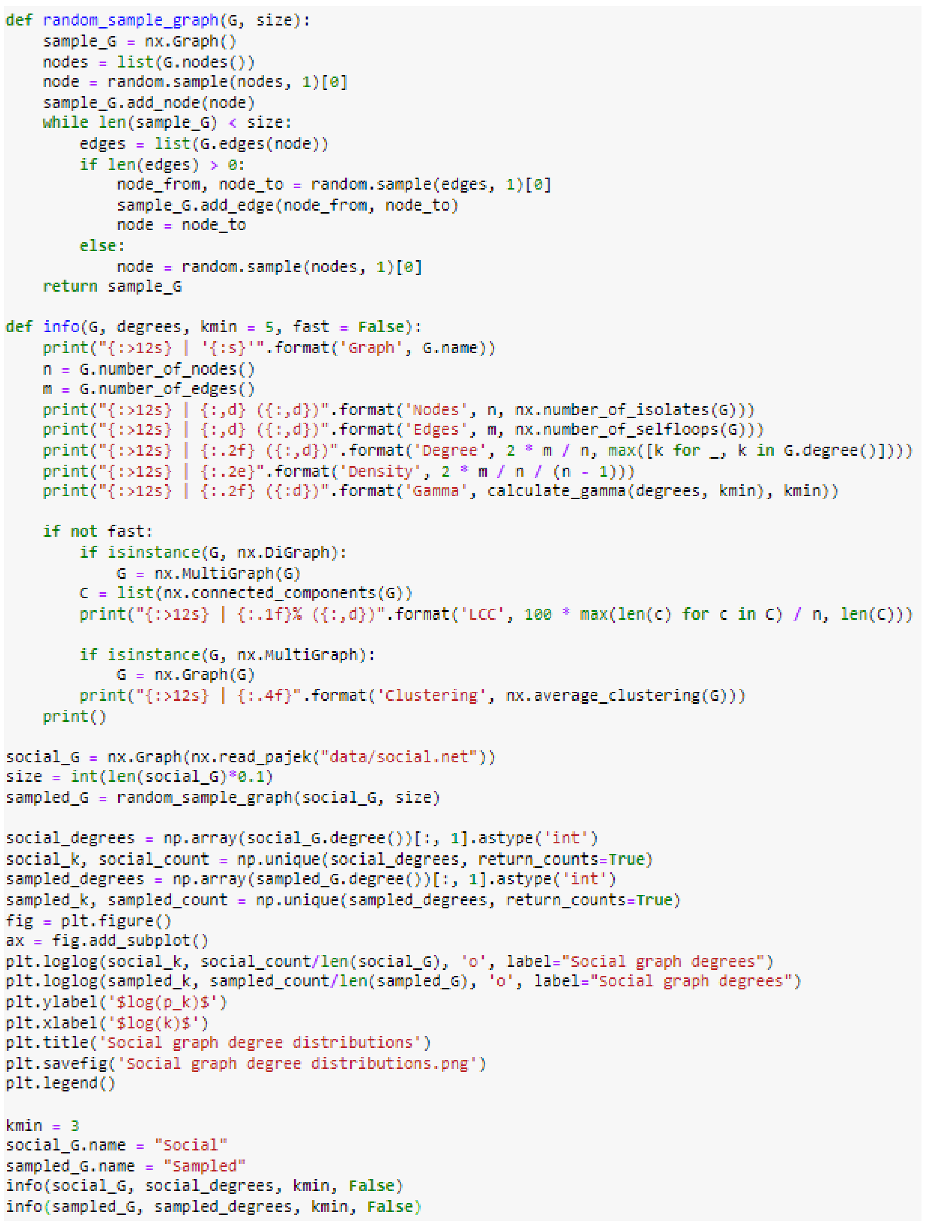


Figure 9: Original and sampled graph degree distributions.



```python
def random_sample_graph(G, size):
    sample_G = nx.Graph()
    nodes = list(G.nodes())
    node = random.sample(nodes, 1)[0]
    sample_G.add_node(node)
    while len(sample_G) < size:
        edges = list(G.edges(node))
        if len(edges) > 0:
            node_from, node_to = random.sample(edges, 1)[0]
            sample_G.add_edge(node_from, node_to)
            node = node_to
        else:
            node = random.sample(nodes, 1)[0]
    return sample_G

def info(G, degrees, kmin = 5, fast = False):
    print("{:>12s} | '{:s}'".format('Graph', G.name))
    n = G.number_of_nodes()
    m = G.number_of_edges()
    print("{:>12s} | {:,d} ({:,d})".format('Nodes', n, nx.number_of_isolates(G)))
    print("{:>12s} | {:,d} ({:,d})".format('Edges', m, nx.number_of_selfloops(G)))
    print("{:>12s} | {:.2f} ({:,d})".format('Degree', 2 * m / n, max([k for _, k in G.degree()])))
    print("{:>12s} | {:.2e}".format('Density', 2 * m / n / (n - 1)))
    print("{:>12s} | {:.2f} ({:d})".format('Gamma', calculate_gamma(degrees, kmin), kmin))

    if not fast:
        if isinstance(G, nx.DiGraph):
            G = nx.MultiGraph(G)
        C = list(nx.connected_components(G))
        print("{:>12s} | {:.1f}% ({:,d})".format('LCC', 100 * max(len(c) for c in C) / n, len(C)))

        if isinstance(G, nx.MultiGraph):
            G = nx.Graph(G)
        print("{:>12s} | {:.4f}".format('Clustering', nx.average_clustering(G)))
    print()

social_G = nx.Graph(nx.read_pajek("data/social.net"))
size = int(len(social_G)*0.1)
sampled_G = random_sample_graph(social_G, size)

social_degrees = np.array(social_G.degree())[:, 1].astype('int')
social_k, social_count = np.unique(social_degrees, return_counts=True)
sampled_degrees = np.array(sampled_G.degree())[:, 1].astype('int')
sampled_k, sampled_count = np.unique(sampled_degrees, return_counts=True)
fig = plt.figure()
ax = fig.add_subplot()
plt.loglog(social_k, social_count/len(social_G), 'o', label="Social graph degrees")
plt.loglog(sampled_k, sampled_count/len(sampled_G), 'o', label="Social graph degrees")
plt.ylabel('$log(p_k)$')
plt.xlabel('$log(k)$')
plt.title('Social graph degree distributions')
plt.savefig('Social graph degree distributions.png')
plt.legend()

kmin = 3
social_G.name = "Social"
sampled_G.name = "Sampled"
info(social_G, social_degrees, kmin, False)
info(sampled_G, sampled_degrees, kmin, False)
```

Figure 10: Code printout.

5 Who to vaccinate?

I would expect the second scheme to provide a better immunization. For the vaccination to work (prevent spreading), we must vaccinate as many people as possible. But we must also take into a count that if just select random people, the chances of them being in mutual companionship are small. Having just one vaccinated member in a group doesn't prevent spreading, while having a group full of vaccinated members does.