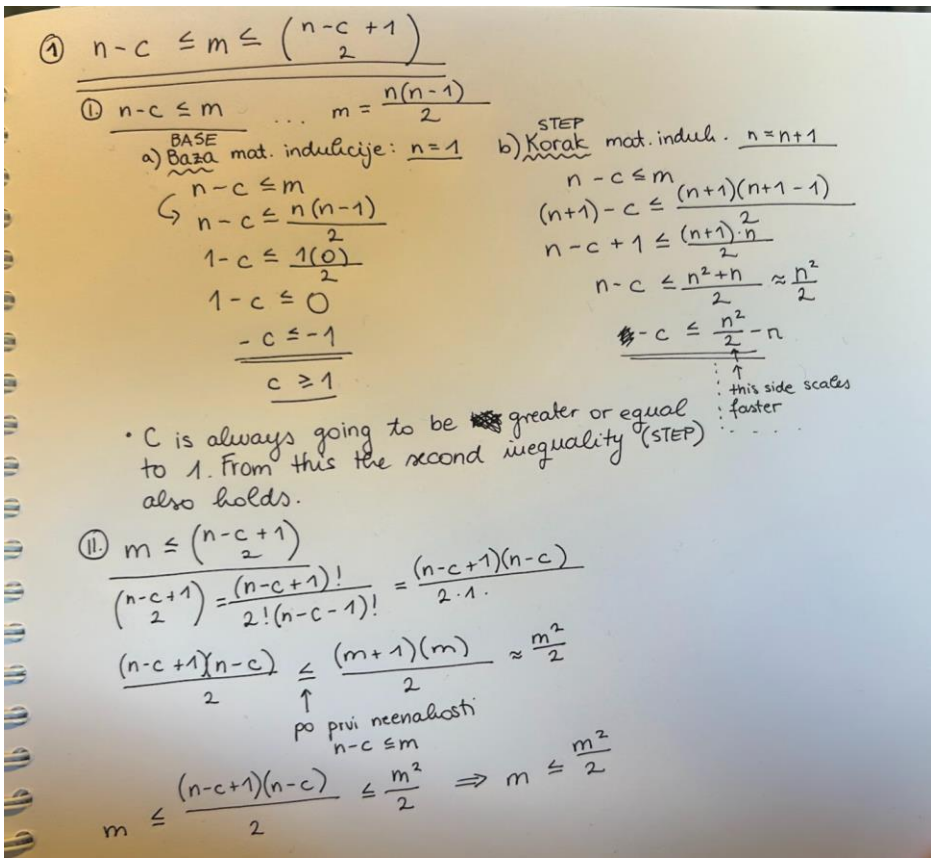
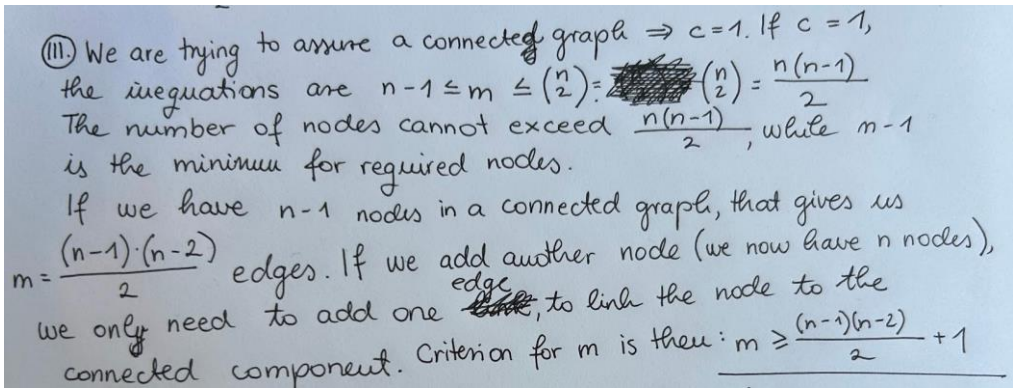
Introduction to Network Analysis – Homework 1 – Ana Polančec, 63190439

1. Connected components

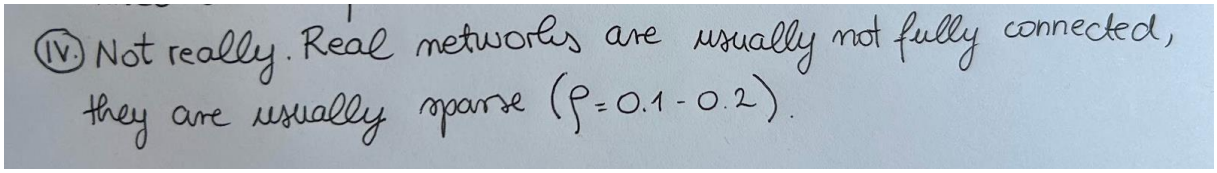
- Proven inequalities with mathematical induction and reasoning (5%)



- Criterion for m to assure a connected graph (2.5%)

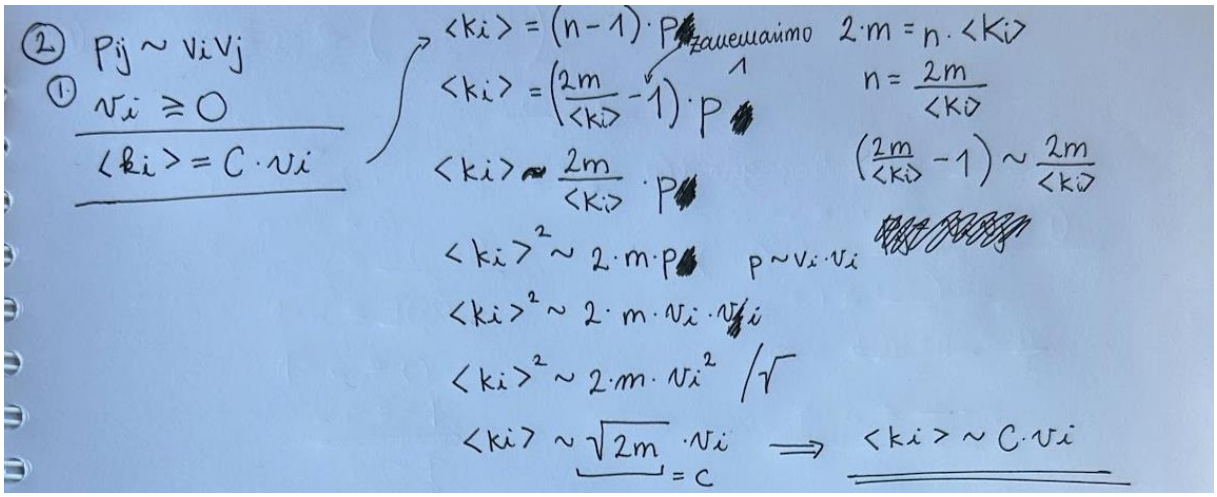


- Is the criterion practically useful for real networks? (2.5%)

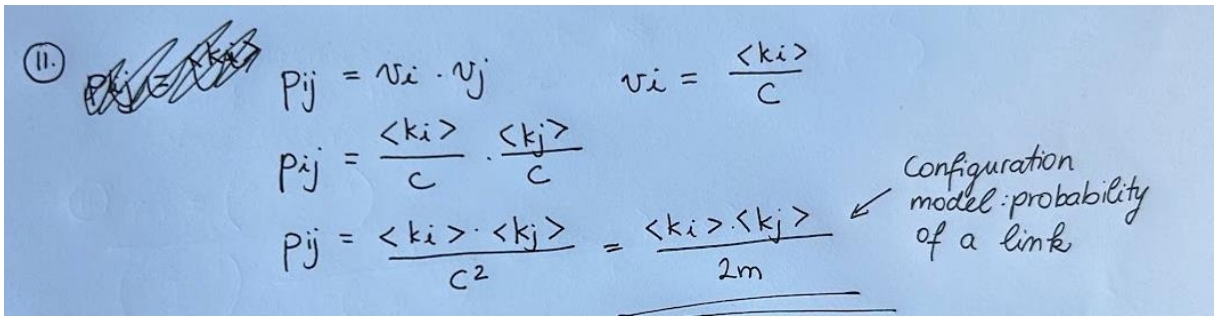


2. Node linking probability

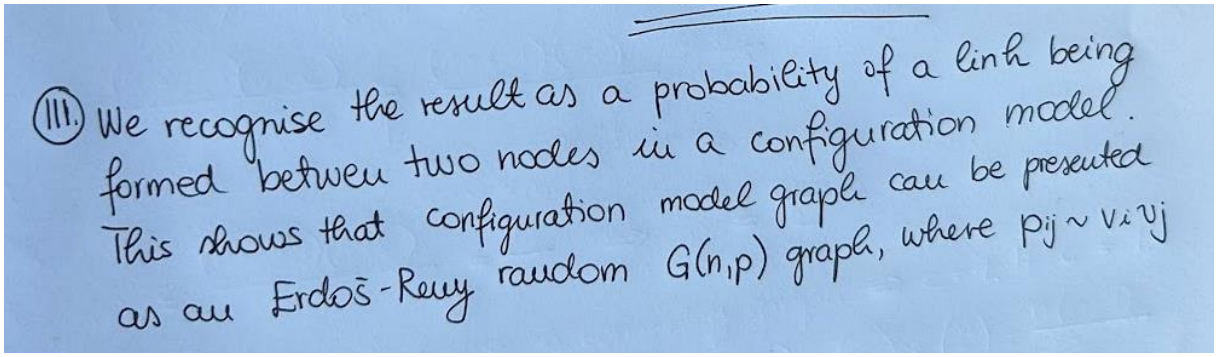
- $\langle k_i \rangle$ is proportional to v_i (5%):



- Expression of p_{ij} in terms of *{k}* (5%):



- Comment of the results (2.5%):



3. **Random node selection**

- Network representation (2.5%):

For a network representation we are choosing between an adjacency matrix, adjacency list and an edge list. For this example edge list is used as the network representation, as it provides O(1) time complexity for accessing a random link. With choosing a random link we provide each node with the probability of being chosen – each nodes probability is proportional to its degree. A random link has 1/m probability of being chosen, which leads to k_i/m probability od i-th node to be chosen. Between the nodes that form the at-random-chosen link, one is chosen at random, giving it 0.5 probability of being chosen. The probability of a node being chosen is as follows:

$$p_i = \frac{ki}{m} * \frac{1}{2} = \frac{ki}{2m}$$

- Describe the algorithm / provide pseudo code (5%):

```
def get_random_node(edge_list):
    selected_edge = select_random_edge(edge_list)
    random_index = randomly_select([0,1])
    selected_node = selected_edge[random_index]
    return selected_node
```

The algorithm takes the edge lit, randomly chooses an edge from the list, then randomly picks between 0 and 1, which determines which node, out of two defining the link, will be picked.

4. Weak & strong connectivity

- Question 1 (2%)

Depth-first search in a directed network (following links in any direction) would find weakly connected components, as it would ignore the direction of the links.

- Question 2 (2%)

Depth-first search in a directed network (following links in the right direction) will not necessarily provide weakly or strongly connected component. The algorithm needs to be modified, to ensure that it finds strongly connected components.

- Question 3 (2%)

Same as in the right direction, algorithm following the links in the opposite direction doesn't necessarily provide us with a weakly/strong connected component. It needs to be modified.

- Pseudocode / printout of the implementation (4%)

```
# algorithm for finding strongly connected components in directed networks
import networkx as nx

#read nodes and edges from the file
def get_nodes_and_edges(file_path):
    G = nx.DiGraph(name = 'enron')

    with open(file_path, 'r') as file:
        next(file)

        for line in file:
            if line.startswith('*arcs'): #found edges
                break
            node_id = line.split()
            G.add_node(int(node_id[0]) - 1, label = node_id[1])

        # Read arcs and edges
        for line in file:
            edge = line.split()
            first =  int(edge[0]) - 1
            second = int(edge[1]) - 1
            G.add_edge(first, second)
    return G


file_path = "enron.net"
G = get_nodes_and_edges(file_path)
print(G)
```

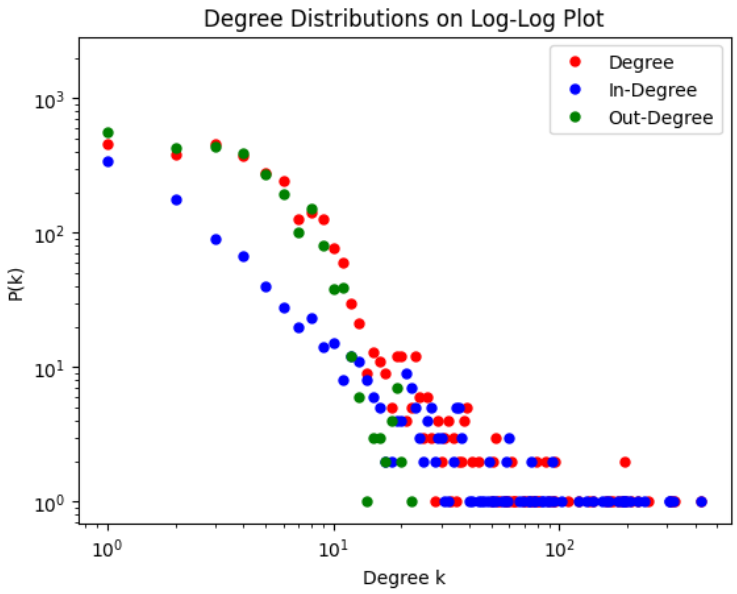
```python
def dfs(node, graph, discovery, low, stack, result, time):
    discovery[node] = time
    low[node] = time
    time += 1
    stack.append(node)

    for neighbor in graph[node]:
        if discovery[neighbor] == -1:
            dfs(neighbor, graph, discovery, low, stack, result, time)
            low[node] = min(low[node], low[neighbor])
        elif neighbor in stack:
            low[node] = min(low[node], discovery[neighbor])

    if low[node] == discovery[node]:
        component = []
        while True:
            top = stack.pop()
            component.append(top)
            if top == node:
                break
        result.append(component)

def tarjan_scc(graph):
    discovery = {node: -1 for node in graph}
    low = {node: -1 for node in graph}
    stack = []
    result = []
    time = 0

    for node in graph:
        if discovery[node] == -1:
            dfs(node, graph, discovery, low, stack, result, time)

    return result
```

- Number of strongly connected components (4%) – 78335 components
- Size of the largest connected component (4%) – 8870 nodes
- Comment of the results (2%)

The results show that there is a lot of connected components in the graph. The larges one contains 9164 out of 87273 nodes. That makes sense, given the nature of the graph. It is an email communication network, where nodes are expected to link with others and not themselves, however nodes are not meant to link with many others. This is how we get a large number of strongly connected components and not one huge one.

5. Is Java software scale-free?

- Draw a plot with three distributions (3 × 2%)



- Reason whether the distributions are seemingly scale-free and why (3 × 2%).

The only distribution that seems to follow the power law is the in-degree distribution. That is compliant with our assumption based on the nature of the network. There are some main classes, that other classes connect to. These classes are hubs as their in-degree is higher than that of their neighbours. The out-degree distribution does not follow the power law distribution, as not many classes connect to many others. Degree distribution follows a similar pattern to out-degree, it also doesn't follow the power law distribution.

- Compute γ· of scale-free distributions for k · min = 5 (4%).

Gamma for in-degree at k_min = 6 is 1.8196234.

- Provide a printout of all the code used to solve the exercise (4%)

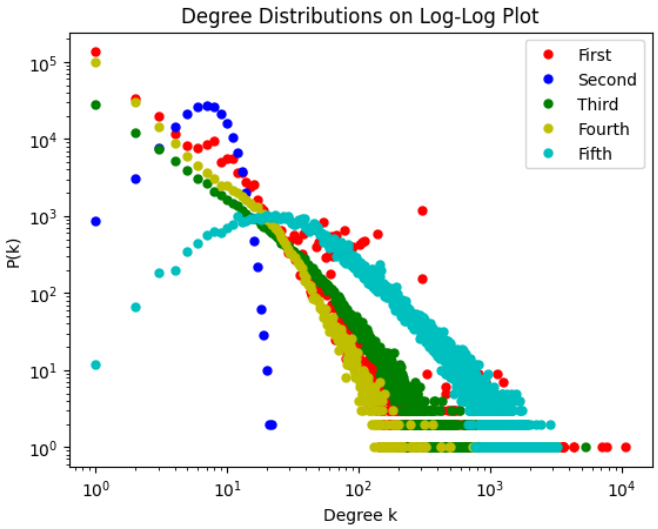
```
#compute degree distribution,in-degree distribution pk in and the out-degree distribution pkout,
file_path = "lucene.net"
G =get_nodes_and_edges(file_path)
degree_distribution = get_degree_distribution(G)
in_degree_distribution = get_in_degree_distribution(G)
out_degree_distribution = get_out_degree_distribution(G)
gamma_degree = calculate_power_law_exponent(in_degree_distribution, len(G.nodes), 6,0,0 )
print(gamma_degree)

#plot the degree distribution (in log), superimpose them on the same plot using dots of different colors
plt.loglog(degree_distribution.keys(), degree_distribution.values(), 'ro', label='Degree', linestyle='None', markersize=5)
plt.loglog(in_degree_distribution.keys(), in_degree_distribution.values(), 'bo', label='In-Degree', linestyle='None', markersize=5)
plt.loglog(out_degree_distribution.keys(), out_degree_distribution.values(), 'go', label='Out-Degree', linestyle='None', markersize=5)

plt.title('Degree Distributions on Log-Log Plot')
plt.xlabel('Degree k')
plt.ylabel('P(k)')
plt.legend()

# Show the plot
plt.show()
```

```python
def get_nodes_and_edges(file_path):
    G = nx.DiGraph(name = 'enron')

    with open(file_path, 'r') as file:
        next(file)

        for line in file:
            if line.startswith('*arcs'): #found edges
                break
            node_id = line.split()
            G.add_node(int(node_id[0]) - 1, label = node_id[1])

        # Read arcs and edges
        for line in file:
            edge = line.split()
            first =  int(edge[0]) - 1
            second = int(edge[1]) - 1
            G.add_edge(first, second)
    return G

def get_degree_distribution(G):
    degree_dist = {}
    for degree in G.degree():
        node, deg = degree
        if deg in degree_dist:
            degree_dist[deg] += 1
        else:
            degree_dist[deg] = 1

    sorted_degrees = {key: degree_dist[key] for key in sorted(degree_dist)}
    print(sorted_degrees)
    return sorted_degrees
```

```python
def get_in_degree_distribution(G):
    degree_dist = {}
    for degree in G.in_degree():
        node, deg = degree
        if deg in degree_dist:
            degree_dist[deg] += 1
        else:
            degree_dist[deg] = 1

    sorted_degrees = {key: degree_dist[key] for key in sorted(degree_dist)}
    print(sorted_degrees)
    return sorted_degrees

def get_out_degree_distribution(G):
    degree_dist = {}
    for degree in G.out_degree():
        node, deg = degree
        if deg in degree_dist:
            degree_dist[deg] += 1
        else:
            degree_dist[deg] = 1

    sorted_degrees = {key: degree_dist[key] for key in sorted(degree_dist)}
    print(sorted_degrees)
    return sorted_degrees

def calculate_power_law_exponent(degree_distribution,n, k_min, sum_ln, nodes_accounted):
    for degree in degree_distribution.items():
        k, p = degree
        if k >= k_min:
            sum_ln += np.log(k/(k_min - 0.5)) * p
            nodes_accounted += p
    gamma = 1 + nodes_accounted /(sum_ln)

    return gamma
```

6. Five networks problem

To decide which network is which, we first determine the initial differences that might help: all the networks apart form Erdos Reny random graph are real networks, which will show sparsity and other parameters, that differ from the Erdoš -Reny graph. Furthermore, two of the networks are directed. Based on the difference between number of edges when transforming each graph to an undirected graph, we determine that network_1 and network_4 are directed. Based on the nature of the networks we can assume that the citation network is directed, as is the university Web graph. Higer average degree is expected from the citation network, then from the university graph. Next, we identify the Erdoš-Reny graph, which should differ from the real networks, based on the degree distribution. Erdoš-Reny graph follows a Poisson distribution, making it the second network.



Between the last two options we determine that the fifth network is the IMDB actors' collaboration network, because it has small distances and a large average degree, which is expected of movie network. The third network is hence Flickr, a network with a small average degree than other undirected graphs.

Final verdict:

- Flickr social affiliation network: network_3.adj
- IMDb actors collaboration network: network_5.adj
- Small part of a university Web graph: network_4.adj
- Sample of computer science citation network: network_1.adj
- Realization of the Erdoš-Renyi random graph: network_2.adj

7. **Who to vaccinate?**

- Better immunization is achieved in the second scheme (first select the same individuals, but then rather vaccinate a random acquaintance of theirs).
- With vaccinating a random acquaintance of a selected person we have a higher probability of vaccinating a »hub« (a node with a high degree), which would prevent the spread of the disease in a more efficient way. This also follows the principle »Your friends always have more friends than you«.