Introduction to Network Analysis
Homework 1

Nace Gorenc

March 2021

1 Networkology

1.1 Node degrees

We could assume that the degree distribution of the left network is Gaussian due to the graph being consisted of nodes with degrees around 4 and not having hubs. Thus the degree sequence is probably increasing until the forth element and then it decreases rapidly.

On the other side, the right network consists majorly of nodes with the degree less than 4 and some hubs of a high degree. Thus the degree sequence has probably high values until the forth element, then it decreases to 0 and finally the elements which correspond to the number of degrees of the hubs have values 1.

1.2 Connected components

We want to show that $n - c \leq m$. It is trivial to show that this formula is correct for small number of m and n.

Firstly, let us show that if the inequality holds for m links, it also holds for $m+1$ links. We can add a link in two different ways. We ether add a link inside connected component or add a link that connects two components. If we add a link inside already connected component, the number of connected components does not change and we get $n - c \leq m + 1$. Moreover, if we add a link that connects two components, the number of connected components decreases by 1 and we get $n - c - 1 \leq m + 1 \Rightarrow n - c \leq m + 2$.

Now let us show that the inequality holds when we increase the number of nodes n by 1. Adding a node increases the number of components by 1, thus $n + 1 - (c + 1) \leq m \Rightarrow n - c \leq m$.

Lastly, let us increase number of connected components c. If we add another connected component with n_0 nodes, it contains at least $n_0 - 1$ links. Thus $n + n_0 - (c + 1) \leq m + n_0 - 1 \Rightarrow n - c \leq m$.

Now we want to show that $m \leq \binom{n-c+1}{2}$. We know that the number of links in the complete graph equals $\binom{n}{2}$. In the best case scenario we have a graph

with c connected components, in which one component is a complete graph and the other $c-1$ components consists of the single nodes. In this case the largest components contains $n-c+1$ nodes, thus $m = \binom{n-c+1}{2}$ which is the highest possible number of links, hence $m \leq \binom{n-c+1}{2}$.

The criteria for m that ensures a connected network is $n-1 \leq m \leq \binom{n}{2}$ which is not useful due to not knowing if there is indeed only one connected component.

1.3 Weak and strong connectivity

We already implemented an algorithm for finding connected components. The same algorithm in a directed network if it follows the links in any direction would find weakly connected components. However, the algorithm that follows the links ether in the proper or opposite direction without any other modification would not find anything. Nevertheless, if we combine those two algorithms, we get a Kosaraju's 1 algorithm which finds all strongly connected components. The printout of the implementation can be found below.

```
def add_to_stack(i, nodes, s, graph):
    nodes[i] = False
    for j in graph[i]:
        if nodes[j]:
            add_to_stack(j, nodes, s, graph)
    s.append(i)

def invert(graph):
    g = [[] for _ in range(len(graph))]
    for i in range(len(graph)):
        for j in graph[i]:
            g[j].append(i)
    return g

def search(g, i, nodes, comp):
    nodes[i] = False
    comp.add(i)
    for j in g[i]:
        if nodes[j]:
            comp = search(g, j, nodes, comp)
    return comp

def scc(graph):
    nodes = [True for _ in range(len(graph))]
    s = []
    for i in range(len(nodes)):
        if nodes[i]:
            add_to_stack(i, nodes, s, graph)

    gi = invert(graph)
```

```
    nodes = [True for _ in range(len(gi))]
    components = []
    while s:
        i = s.pop()
        if nodes[i]:
            components.append(search(gi, i, nodes, set()))
    return components
```

Algorithm 1 Kosaraju's algorithm

procedure STRONGLY CONNECTED COMPONENTS(graph G)
  $nodes \leftarrow$ *boolean list* of True
  $s \leftarrow$ empty *stack*
  for node $i \in G$ **do**
    if $nodes[i]$ **then**
      addOnStack(i, $nodes$, s, G)
  $G_{inverse} \leftarrow$ invert(G)
  $nodes \leftarrow$ *boolean list* of True
  $C \leftarrow$ empty *list*
  while s not empty **do**
    $i \leftarrow s$.pop()
    if $nodes[i]$ **then**
      C.append(search(i, $nodes$, $G_{inverse}$, empty *set*))
  return C
procedure ADDONSTACK(node i, boolean list $nodes$, stack s, graph G)
  $nodes[i] \leftarrow$ False
  for $j \in G[i]$ **do**
    if $nodes[j]$ **then**
      addToStack(j, $nodes$, s, G)
  s.append(i)
procedure SEARCH(node i, boolean list $nodes$, graph G, set c)
  $nodes[i] \leftarrow$ False
  c.add(i)
  for $j \in G[i]$ **do**
    if $nodes[j]$ **then**
      $c \leftarrow$ search(j, $nodes$, G, c)
  return c

1.4 Node and network clustering

In the example network, we start with two connected nodes. Each additional node has a degree of 2 and is only connected to two staring nodes. Hence two starting nodes have clustering coefficient C_i equal to $\frac{2 \cdot t}{(t+1)t}$, where t represents number of added nodes, and every additional node has clustering coefficient

equal to 1 meaning that $\langle C \rangle = 1$ when $n \to \infty$. In addition, number of linked triples in network increases faster than number of triangles, thus $C = 0$ when $n \to \infty$. Formally:

$$\begin{aligned}
\lim_{n\to\infty} \langle C \rangle &= \lim_{n\to\infty} \frac{1}{n} \sum_i C_i = \lim_{n\to\infty} \frac{n - 1 + \frac{2(n-2)}{(n-1)(n-2)}}{n} = \\
&= \lim_{n\to\infty} \frac{n - 1 + \frac{2n-4}{n^2-3n+2}}{n} = \lim_{n\to\infty} \frac{1 - \frac{1}{n}}{1} = 1 \\
\lim_{n\to\infty} C &= \lim_{n\to\infty} \frac{3n}{3(n-2) + 2\binom{n-2}{2}} = \lim_{n\to\infty} \frac{3n}{3n - 6 + (n-2)(n-3)} = \\
&= \lim_{n\to\infty} \frac{3n}{3n - 6 + n^2 - 5n + 6} = \lim_{n\to\infty} \frac{3n}{n^2 - 2n} = \lim_{n\to\infty} \frac{\frac{3}{n}}{1 - \frac{2}{n}} = 0
\end{aligned}$$

Number of linked triples is calculated as sum of number of linked triangle formed by two starting nodes and one additional node and the number of possible linked triples with one starting node and two added nodes.

We could build the network with this property with only one starting node. Each iteration we add two additional nodes and connect them with the starting node so they form a triangle. Example of this network is shown in the figure 1. The starting node would in this case have a degree of $n-1$ and each additional node would have a degree of 2 meaning that clustering coefficient of the starting node equals $\frac{1}{n-2}$ (where n is odd number, not important in the limit) and the clustering coefficient of the other nodes equals 1. Thus $C = 1$ when $n \to \infty$. Comparable to the previous example, it is obvious that the number of linked triples in network increases faster than numbers of triangles, thus $C = 0$ when $n \to \infty$. Formally:

$$\begin{aligned}
\lim_{n\to\infty} \langle C \rangle &= \lim_{n\to\infty} \frac{1}{n} \sum_i C_i = \lim_{n\to\infty} \frac{n - 1 + \frac{\frac{n-1}{2}}{(n-1)(n-2)}}{n} = \\
&= \lim_{n\to\infty} \frac{n - 1 + \frac{n-1}{2n^2-6n+4}}{n} = \lim_{n\to\infty} \frac{1 - \frac{1}{n}}{1} = 1 \\
\lim_{n\to\infty} C &= \lim_{n\to\infty} \frac{3\frac{n-1}{2}}{n - 1 + \binom{n-1}{2}} = \lim_{n\to\infty} \frac{\frac{3n-3}{2}}{n - 1 + \frac{(n-1)(n-2)}{2}} = \\
&= \lim_{n\to\infty} \frac{\frac{3n-3}{2}}{n - 1 + \frac{n^2-3n+2}{2}} = \lim_{n\to\infty} \frac{\frac{3}{2n} - \frac{3}{2n^2}}{\frac{1}{n} - \frac{1}{n^2} + \frac{1}{2} - \frac{3}{2n} + \frac{1}{n^2}} = 0
\end{aligned}$$

Number of linked triples is calculated as sum of number of linked triangle formed by a starting nodes and two connected additional nodes and the number of possible linked triples with one starting node and two added nodes which are not connected with a direct link.

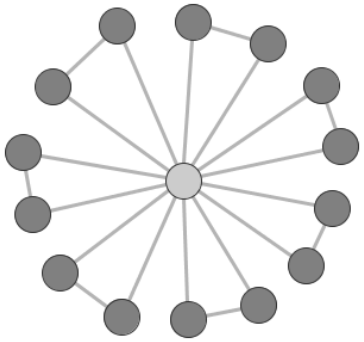

Figure 1: Example of the network.

1.5 Effective diameter evolution

The algorithm 2 presents the pseudo code for the 90-percentile effective diameter d_{90}. The implementation of an algorithm can be found below. The results on citation networks of physics papers published are:

- 2010-2011: $d_{90} = 15$,
- 2010-2012: $d_{90} = 13$,
- 2010-2013: $d_{90} = 11$.

The results are not surprising since there is a great probability that the new papers will cite the old ones and that it will cite a great number of related papers, therefore new papers connect the older papers even more.

```
from _collections import deque
import numpy as np

def distances(graph, i):
    d = [None for _ in range(len(graph))]
    q = deque()
    q.append(i)
    d[i] = 0
    while q:
        i = q.popleft()
        for j in graph[i]:
            if d[j] is None:
                d[j] = d[i] + 1
                q.append(j)
    d = np.array([np.inf if x is None else x for x in d])
    return d

def network_distances(graph):
    d = list()
```

```
    for i in range(len(graph)):
        d.append(distances(graph, i))
    return d

if __name__ == '__main__':
    g = graph_init()
    d = np.percentile(np.array(network_distances(g)), 90)
    print('d_90 = ' + str(d))
```

Algorithm 2 90-percentile effective diameter d_{90}

procedure MAIN
  print percentile(networkDistences(G), 90)
procedure NETWORKDISTANCES(graph G)
  $D \leftarrow$ empty *list*
  for nodes $i \in N$ **do**
    D.append(distances(G, i))
  return D
procedure DISTANCES(graph G, node i)
  $D \leftarrow$ empty *array*
  $Q \leftarrow$ empty *queue*
  Q.enqueue(i)
  $D[i] \leftarrow 0$
  while not Q.isEmpty() **do**
    $i \leftarrow Q$.dequeue()
    for neighbors $j \in \Gamma_i$ **do**
      if $D[j]$ undefined **then**
        $D[j] \leftarrow D[i]+1$
        Q.enqueue(j)
  return D

2 Graph models

2.1 Random node selection

The network in this task could be represented by the dictionary where keys would be node indices end values list of neighboring nodes. The dictionary could also store the number of links. The probability of selecting node i equals to $\frac{size\ of\ list\ of\ neighbors}{2 \times number\ of\ links}$ which is calculated in constant time. The pseudo code can be found in the algorithm 3.

Algorithm 3 Select random node

1: **procedure** RANDOM SELECTION(graph G)
2: $P \leftarrow$ empty list
3: **for** node $i \in G$ **do**
4: P.append(size($G[i]$) / $(2 \cdot G['m'])$)
5: **return** random(P)

2.2 Node linking probability

The relationship of probability p_{ij} being proportional to $v_i \cdot v_j$ can bi written as $p_{ij} = D_{ij} \cdot v_i v_j$ for some constant D_{ij}. The expected value of node degree then equals to $\langle k_i \rangle = \sum_j p_{ij} = \sum_j D_{ij} v_i v_j = v_i \cdot \sum_j D_{ij} v_j = v_i \cdot C_i$.

We get

$$p_{ij} = D_{ij} v_i v_j = D_{ij} \frac{\langle k_i \rangle}{C_i} \frac{\langle k_j \rangle}{C_j}.$$

The result looks similar to the probability of link between nodes in configuration graph model.

2.3 Node degree distribution

Figure 2 represents node degree distributions of 3 networks. As we can observe, the Facebook social network and the random graph looks somehow similarly with an exception that Facebook social network has some nodes of zero or very low degree compared to the random graph which contains nodes with the degree 10 or more. Furthermore, both networks have some nodes of high degree. On the other hand, the Erdős-Rényi random graph has a degree distribution similar to the Poisson distribution, with the majority of the nodes with a degree around 11. The implementation of the preferential attachment model and the code used to compute p_k is shown below.

```
import networkx as nx
from math import ceil

def random_selection(g, t):
    m = g.number_of_edges()
    n = g.number_of_nodes()
    p = [0 for _ in range(n)]
    for i, k in g.degree:
        p[i] = k / (2*m)
    return np.random.choice(n, t, p=p, replace=False)

def complete_graph(n):
    g = nx.Graph()
    for i in range(n):
        g.add_node(i)
```

```
        for j in range(0, i):
            g.add_edge(j, i)
    return g

def random_graph(n, k):
    g = complete_graph(ceil(k) + 1)
    for i in range(ceil(k) + 1, n):
        nodes = random_selection(g, ceil(k/2))
        for j in nodes:
            g.add_edge(i, j)
    return g

def p_k(g, n):
    tmp = []
    for _, d in g.degree:
        tmp.append(d)
    unique = set(tmp)
    p = [0 for _ in range(max(unique) + 1)]
    for i in unique:
        p[i] = tmp.count(i) / n
    return p
```

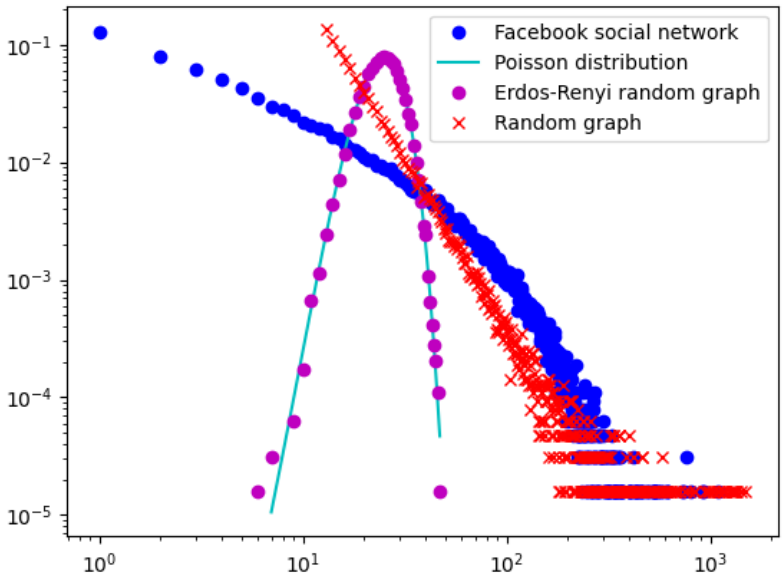

Figure 2: Node degree probability vs. node degree.

3 Node position

In this exercise we use the Spearman correlation coefficient on each of the three following measures: node degree, clustering coefficient and closeness centrality measure. The results can be found in table 1. As we can observe, the closeness centrality measure give us the best results which is expected due to the highest scoring nodes being geographically in the center of the network. Node degree gave us significantly lower results which is also expected considering that it favours nodes with higher degree and not the nodes with potentially high traffic load. Lastly, under performance of the clustering coefficient is not surprising at all since the network does not have any triangles.

Measure	Spearman correlation
node degree	0.273
clustering coefficient	NaN
closeness centrality measure	0.607

Table 1: Spearman correlation coefficients of measures.

Therefore, table 2 shows top 10 locations according to closeness centrality measure. As we can see, all of the location are ether on the Ljubljana ring road or near it which makes sense since a lot of people commute daily to Ljubljana.

Location	Traffic load	Closeness centrality measure
Zadobrova	33779.16001475427	0.06392931392931393
Šmartinska	33779.16001475427	0.06317411402157165
Sneberje	20686.53151593403	0.06307692307692307
Tomačevo	20686.53151593403	0.062436548223350256
Zaloška	20686.53151593403	0.0624048706240487
Šentjakob	26423.031567407153	0.0621840242669363
Litijska	31067.820408607127	0.061809045226130656
Dunajska	20686.53151593403	0.06171600602107376
Malence	55297.864701883555	0.06168505516549649
Domžale	26423.031567407153	0.061254980079681276

Table 2: Top 10 location according to closeness centrality measure.

The implementation for this exercise is shown below.

```
from scipy.stats import spearmanr
import networkx as nx

def node_position():
    g = [0] * 124
    d = [0] * 124
    c = [0] * 124
```

```
    l = [0] * 124
    with open('data/highways') as f:
        for line in f:
            if line.startswith('#'):
                a = line.split()
                i, k = int(a[1])-1, float(a[-1])
                g[i] = k
    graph = nx.read_adjlist('data/highways.net')
    for i, deg in graph.degree:
        ind = int(i)
        d[ind-1] = deg
        c[ind-1] = nx.clustering(graph, i)
        l[ind-1] = nx.closeness_centrality(graph, i)
    print(spearmanr(g, d))
    print(spearmanr(g, l))

    r = list(zip([x for x in range(1,125)], g, l))
    r.sort(key=lambda x: x[2], reverse=True)
    for i in range(10):
        print(r[i])
```