INA - homework 1

Matic Isovski, 63180442

March 2022

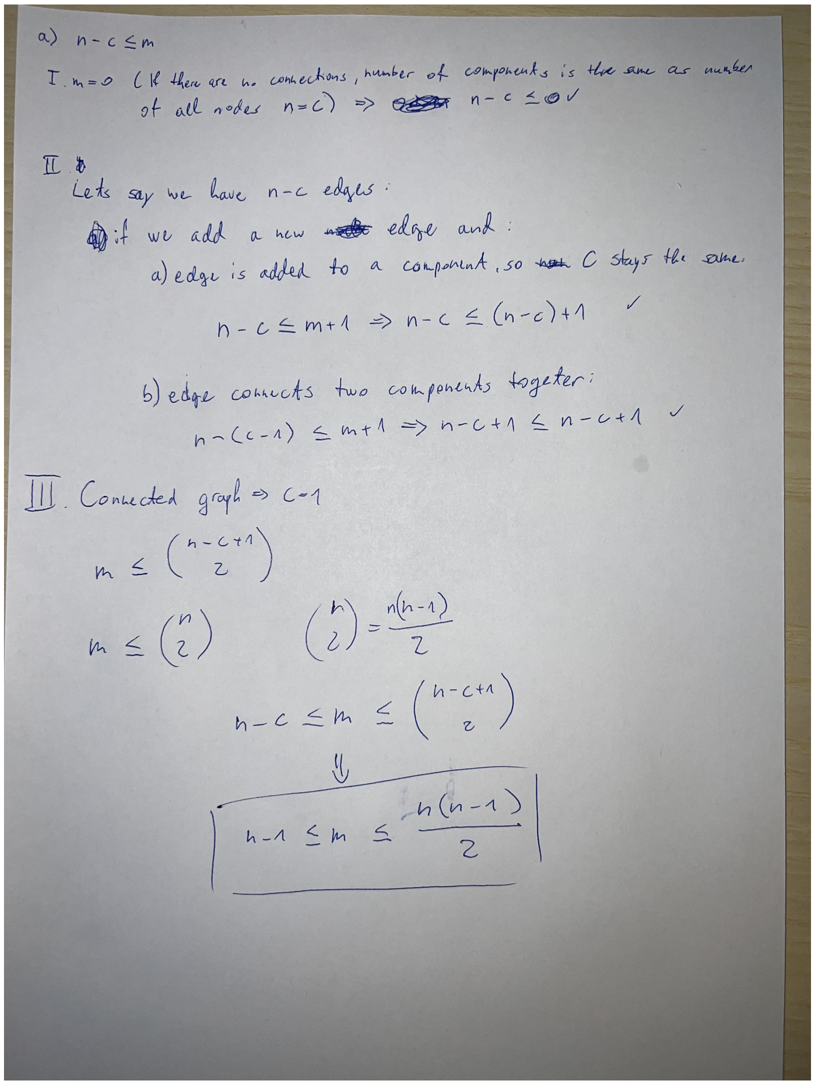
1 Networkology

1.1 Node degrees

At a glance, the first network has more hubs than the second one, so their degree sequences and degree distributions should be different. First network should have more linear-like distribution, while the second one has more of low-degree nodes and a few of hubs.

1.2 Connected components

So for a connected graph we need at least $n - 1$ edges and can have up to $\frac{n(n-1)}{2}$ edges. For real-world networks this criterion stands, but it is closer to minimum (for example in 'aps-2010-2011' directed network, there are around 19k nodes and 42k connections).



a) $n - c \leq m$

I. $m = 0$ (If there are no connections, number of components is the same as number of all nodes $n = c$) $\Rightarrow$ $n - c \leq 0$ ✓

II.

Lets say we have $n-c$ edges:

if we add a new edge and:

a) edge is added to a component, so C stays the same.

$$n - c \leq m + 1 \Rightarrow n - c \leq (n-c) + 1 \quad ✓$$

b) edge connects two components togeter:

$$n - (c-1) \leq m + 1 \Rightarrow n - c + 1 \leq n - c + 1 \quad ✓$$

III. Connected graph $\Rightarrow$ $c = 1$

$$m \leq \binom{n-c+1}{2}$$

$$m \leq \binom{n}{2} \qquad \binom{n}{2} = \frac{n(n-1)}{2}$$

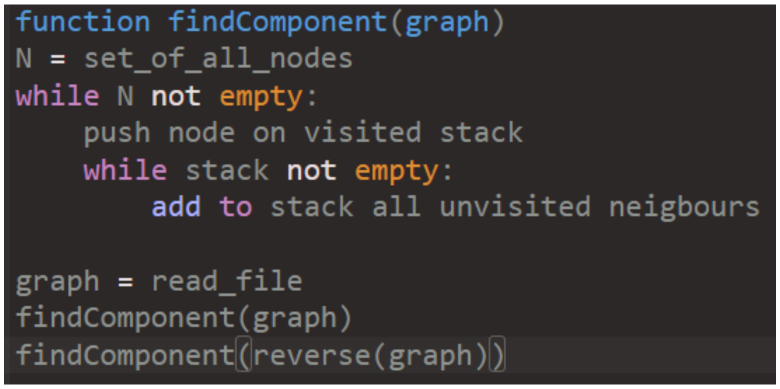
$$n - c \leq m \leq \binom{n-c+1}{2}$$

$$\Downarrow$$

$$\boxed{n - 1 \leq m \leq \frac{n(n-1)}{2}}$$

1.3 Weak & strong connectivity

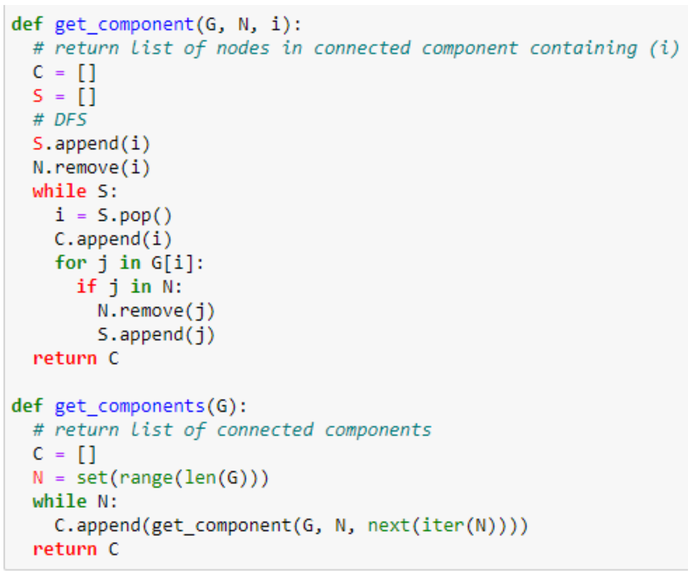
Searching components in just one direction would give us weak connections, so components would be wrong. We would have to search in both ways so we can find cycles and isolated nodes, which are components of a graph.



```
function findComponent(graph)
N = set_of_all_nodes
while N not empty:
    push node on visited stack
    while stack not empty:
        add to stack all unvisited neigbours

graph = read_file
findComponent(graph)
findComponent(reverse(graph))
```

Figure 1: Pseudocode



```
def get_component(G, N, i):
  # return list of nodes in connected component containing (i)
  C = []
  S = []
  # DFS
  S.append(i)
  N.remove(i)
  while S:
    i = S.pop()
    C.append(i)
    for j in G[i]:
      if j in N:
        N.remove(j)
        S.append(j)
  return C

def get_components(G):
  # return list of connected components
  C = []
  N = set(range(len(G)))
  while N:
    C.append(get_component(G, N, next(iter(N))))
  return C
```

Figure 2: Printout

1.4 Node & network clustering

We can use clustering coefficient based on triplets $C = \frac{3 \times triangles}{triplets}$. By adding nodes to the graph $n \to \infty$, we are increasing the number of triplets: $C \to 0$.

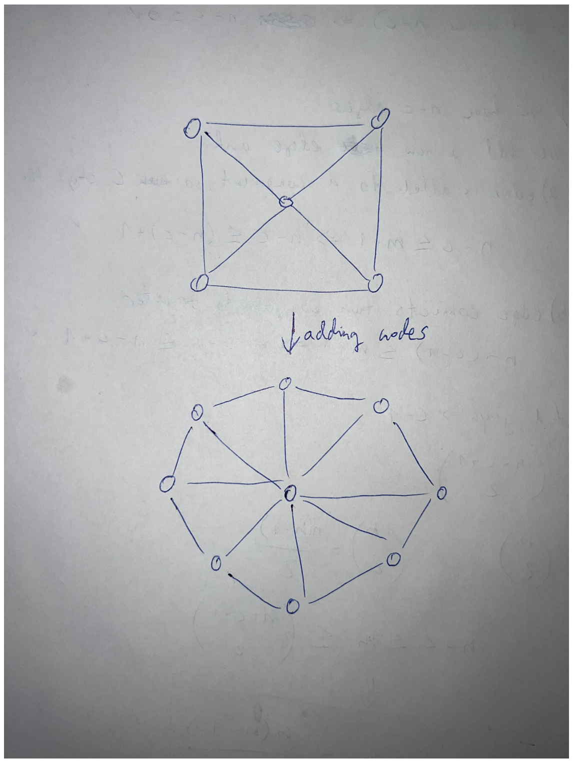
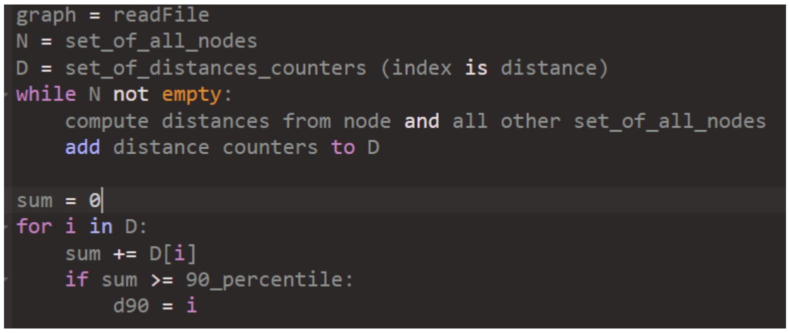


Figure 3: Graph example.

1.5 Effective diameter evolution

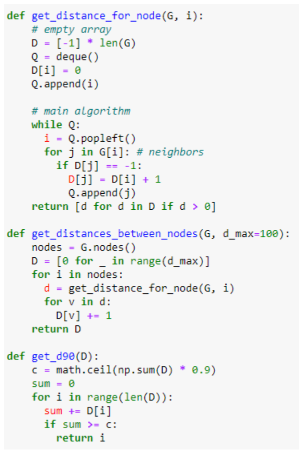
For aps_2010_2011 we got $d90 = 15$, for aps_2010_2012 we got $d90 = 13$, for last one something went wrong, but I assume that it gets even lower and that would indicate, that each year the graph is more connected.



```
graph = readFile
N = set_of_all_nodes
D = set_of_distances_counters (index is distance)
while N not empty:
    compute distances from node and all other set_of_all_nodes
    add distance counters to D

sum = 0
for i in D:
    sum += D[i]
    if sum >= 90_percentile:
        d90 = i
```

Figure 4: Pseudocode



```
def get_distance_for_node(G, i):
    # empty array
    D = [-1] * len(G)
    Q = deque()
    D[i] = 0
    Q.append(i)

    # main algorithm
    while Q:
      i = Q.popleft()
      for j in G[i]: # neighbors
        if D[j] == -1:
          D[j] = D[i] + 1
          Q.append(j)
    return [d for d in D if d > 0]

def get_distances_between_nodes(G, d_max=100):
    nodes = G.nodes()
    D = [0 for _ in range(d_max)]
    for i in nodes:
      d = get_distance_for_node(G, i)
      for v in d:
        D[v] += 1
    return D

def get_d90(D):
    c = math.ceil(np.sum(D) * 0.9)
    sum = 0
    for i in range(len(D)):
      sum += D[i]
      if sum >= c:
        return i
```

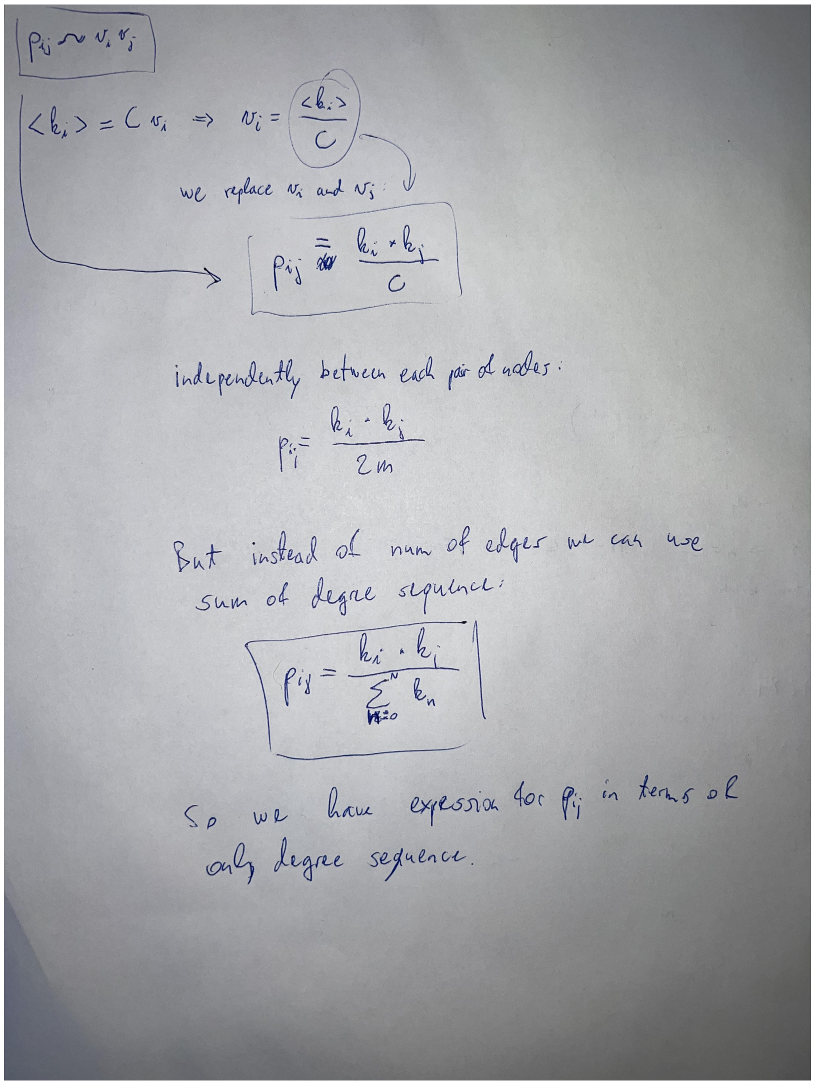
Figure 5: Printout

2 Graph models

2.1 Random node selection

In this case, we could use edge list graph representation. If the network is directed, then we would make it undirected by adding edges in both ways. For building, we would randomly choose an edge and take the end node. So there are $2m$ of edges and the chances of selecting a node is proportionate to the degree of the node k_i, so we get $\frac{k_i}{2m}$.

2.2 Node linking probability



$\boxed{p_{ij} \sim v_i v_j}$

$\langle k_i \rangle = C v_i \Rightarrow v_i = \frac{\langle k_i \rangle}{C}$

we replace v_i and v_j:

$$\boxed{p_{ij} = \frac{k_i * k_j}{C}}$$

independently between each pair of nodes:

$$p_{ij} = \frac{k_i \cdot k_j}{2m}$$

But instead of num of edges we can use sum of degree sequence:

$$\boxed{p_{ij} = \frac{k_i \cdot k_j}{\sum_{n=0}^{N} k_n}}$$

So we have expression for p_{ij} in terms of only degree sequence.

2.3 Node degree distribution

From the plots on image (7), we can see that the Facebook social networks has a lot more of high degree nodes (hubs) than ER graph. We can also see that the distribution of ER graph is binomial, and it fits the theoretical degree almost perfectly.

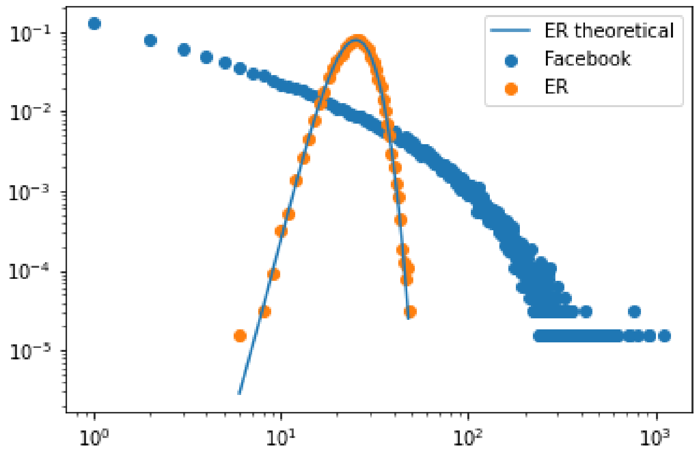
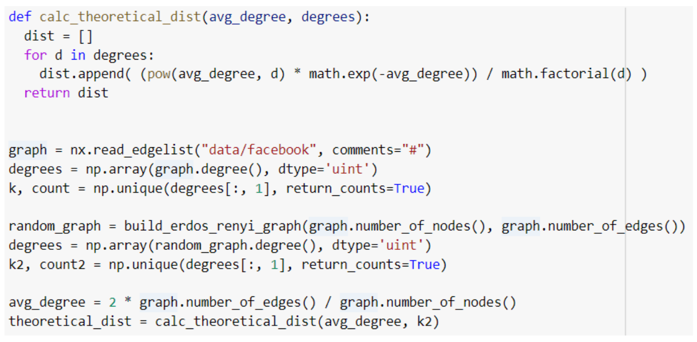


Figure 6: Node degrees distributions.



```
def calc_theoretical_dist(avg_degree, degrees):
  dist = []
  for d in degrees:
    dist.append( (pow(avg_degree, d) * math.exp(-avg_degree)) / math.factorial(d) )
  return dist


graph = nx.read_edgelist("data/facebook", comments="#")
degrees = np.array(graph.degree(), dtype='uint')
k, count = np.unique(degrees[:, 1], return_counts=True)

random_graph = build_erdos_renyi_graph(graph.number_of_nodes(), graph.number_of_edges())
degrees = np.array(random_graph.degree(), dtype='uint')
k2, count2 = np.unique(degrees[:, 1], return_counts=True)

avg_degree = 2 * graph.number_of_edges() / graph.number_of_nodes()
theoretical_dist = calc_theoretical_dist(avg_degree, k2)
```

Figure 7: Printout of p_k computing.