Colonna, Bonifazi, Yusifli

At first we imported all the modules we needed for the whole homework:

```
import json
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter
import networkx as nx
import numpy as np
import queue as Q
import sys
```

1. DATA

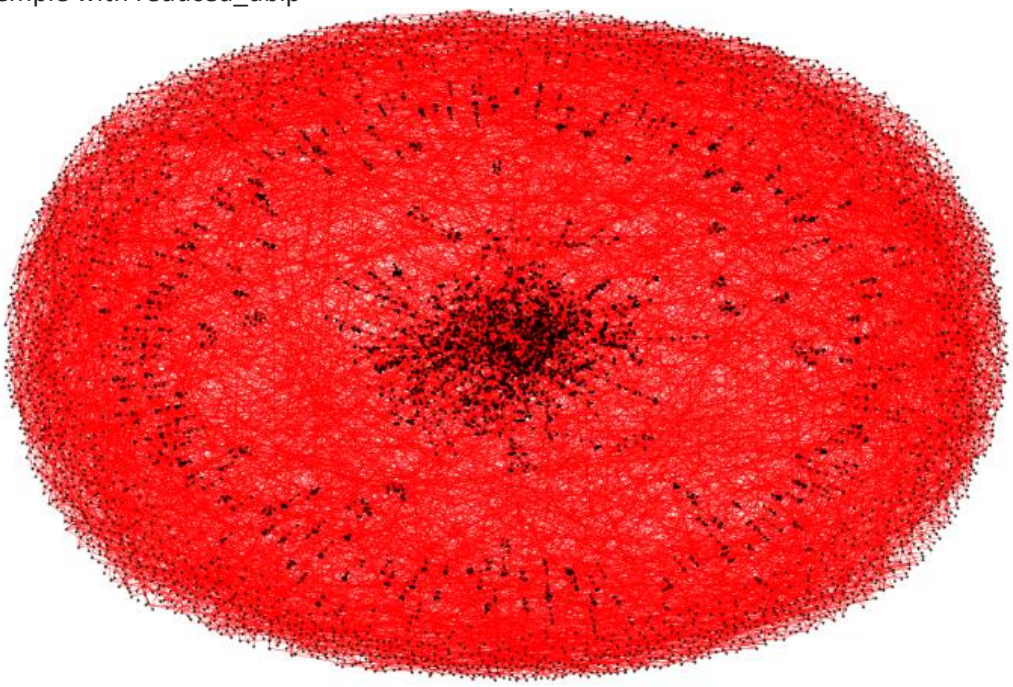
The goal of the first exercise was to create a graph G, by applying the graph methodologies.
After we brought up the full_dblp.json file, we decoded json data defining json.loads(data) as dataset. Having the dataset, we created a dictionary, using 2 for loop: one for dic in range(len(dataset)), one for j in range(len(dataset[dic]['authors'])). In this dictionary, we have as keys the authors ID, and for values a list of publications ID that each author wrote, as well as the conference ID in which each publication has been presented. The output has struct {authorID: [list of {confID:pubID}]}.

After that, we defined the Jaccard's Similarity function. We needed this function to weigh each edge of the graph. In fact, the graph's nodes are the authors, while the nodes are connected if they share, at least, one publication. For the jaccardSim we created 2 empty lists, in which we appended the keys. We calculated the length of the intersection between the two lists and as result we put:
similarityScore = (intersect / ((len(myList1) + len(myList2)) - intersect)).

We create the graph G. We used 3 for loops: one for data in dataset, the second for author in authors and the last one for author2 in authors. We needed 2 author's id, in order to put the condition aId != a2Id in the if, to not create edges on the same node. We defined the nodes before the if and then we have weighed each edge as following: edgeWeight = 1 - jaccardSim(myDict[author["author_id"]] , myDict[author2["author_id"]]).
After that, we also printed the graph's info: print(nx.info(G)).

At the end we print the graph:

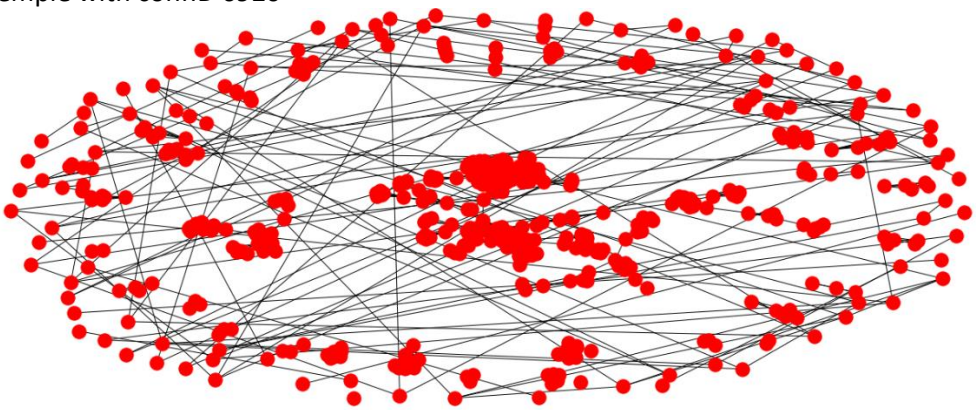
Exemple with reduced_dblp

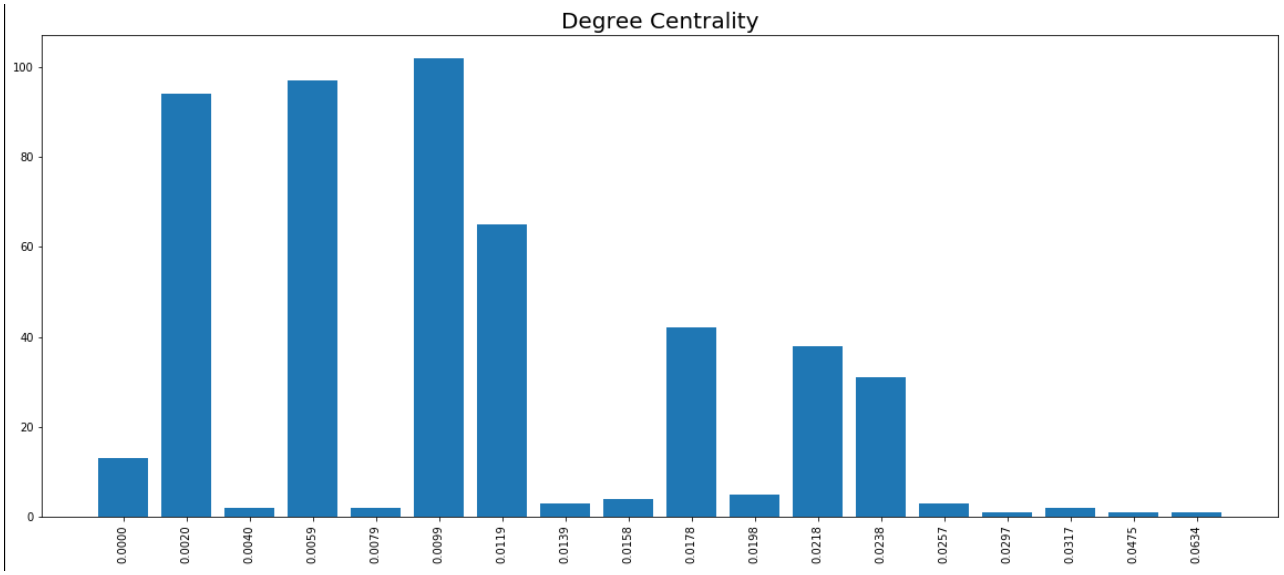


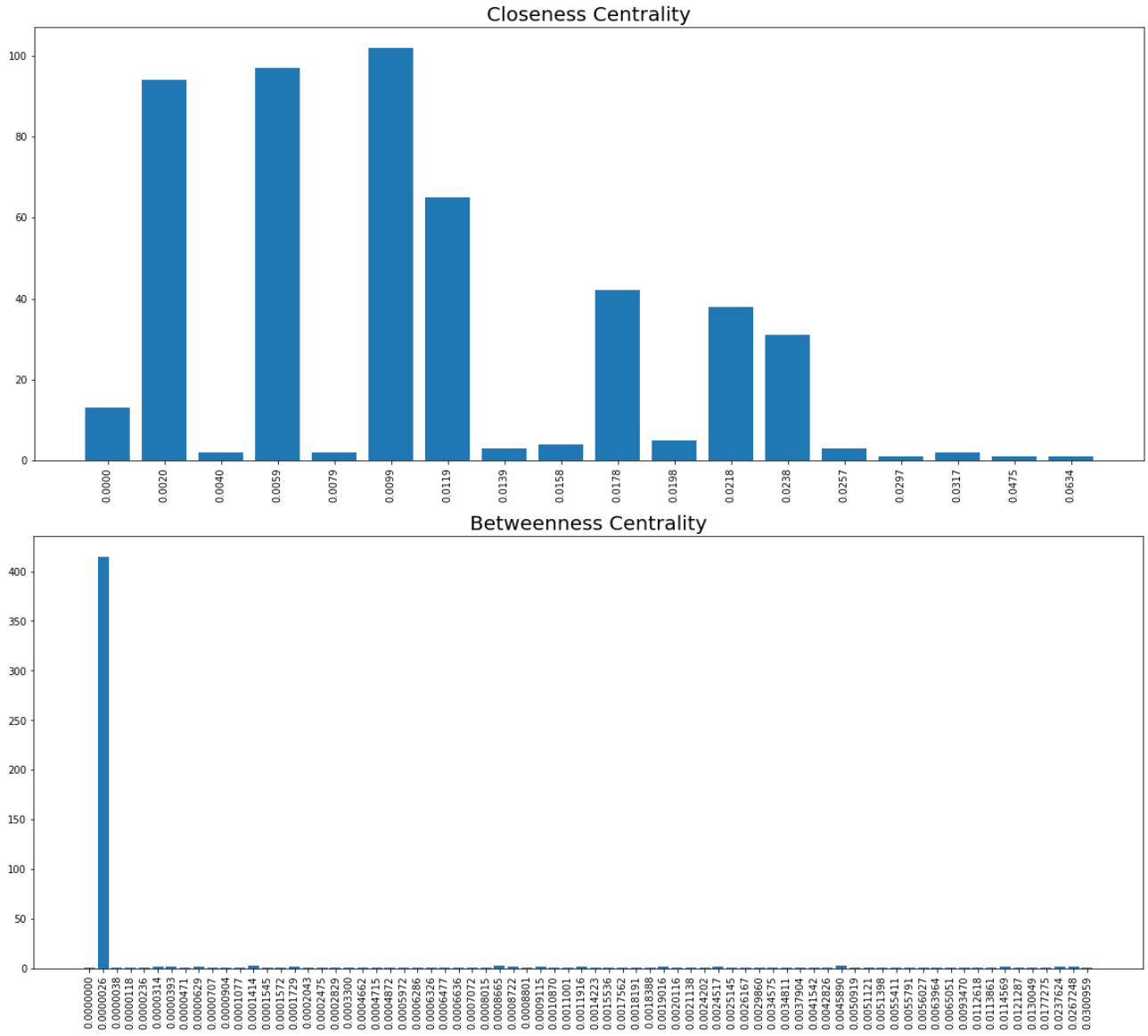
2. STATISTICS & VISUALIZATIONS

a) At first, we create a dictionary of struct {conference id: [list of authors who participated in that conference]}, to have a clear visualization of each conference's ID and the authors' ID of who have participated in each conference.
Choosing the conference in input searchconfID = input("Search confID: "), we then create a list of nodes, which are the authors in the input conference. Defining H the graph, we returned the subgraph induced by the set of authors who published at the input conference.
For what concern the centralities measures, we used the Python package networkx. It allowed us to calculate degree centrality, closeness centrality and betweenness centrality. For each one of these measures, we created a dictionary, and from each dictionary we plotted them. Besides, for the measures' plots, we used ticker's module from matplotlib, which contains classes to support completely configurable tick locating and formatting. We then imported the FuncFormatter function, which sets the labels.

Exemple with confID 6920

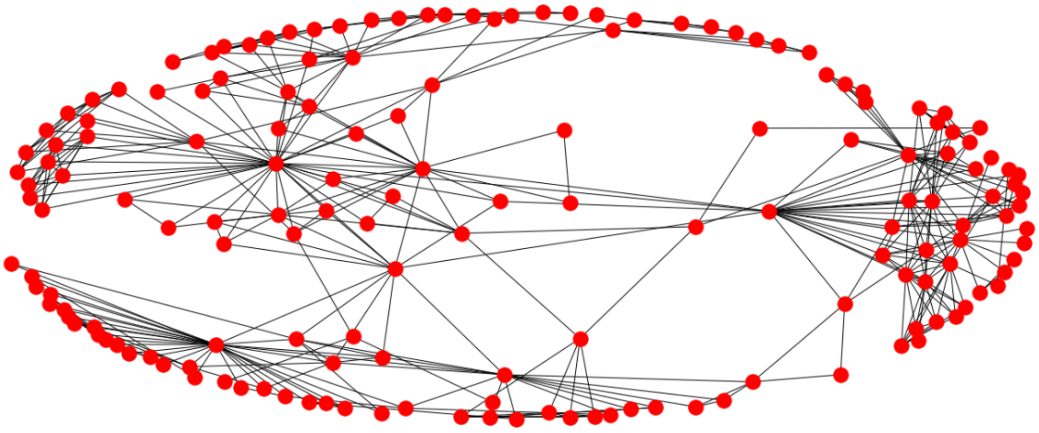






b) Choosing an author id and a integer, we returned an induced subgraph of neighbors centered at node (node is the author id), within a given radius (radius is the integer), using ego graph. Radius includes all neighbors of distance <= radius from the input node. In fact, radius is the hop distance we want. Then we plotted and visualized the graph.

Exemple with authorID: 256176



3. ARIS NUMBER

a) The goal of this exercise was to write a Python software that takes in input an author (id) and returns the weight of the shortest path that connects the input author with Aris.
We defined a function Shortest_path(G, start, end) where start is the authorID of Aris, while end is the authorID in input.
Starting from the node which is the ArisID, the function takes all the one step neighbor's nodes and it put them in an empty list neighb[]. It will be a list of tuples, and each tuple contains the shortest weighted distance between the starting node and another node, whose ID is the other tuple's element.
With the heapq.heappop function we took the smallest distance in the list and we checked the distances between the starting node and all the neighbors of the related node (in the tuple). If one of these distances was already in the list and the new one was shortest, we updated it in the list. Otherwise, if it wasn't in the list, we added it to our list of tuples.
Since we took the smallest distance, when it is related to the ending node, this was going to be our result.
If the starting node and the ending node weren't connected, we fixed the result as "inf".

b) We used the basic approach for this point. We calculate for each node in the graph the shortest path distance between this and all the nodes in the set in input. After that, we took the minimum between them with the heappop function.