# ADM – Homework4

*Giacomo Parmendola(1462237)*
*Vigèr Durand Azimedem Tsafack(1792126)*
*Mattia Podio(1554740)*

## ✓ **First Step**: Graph creation

**1.** Here we process the JSON file and create a graph with all the authors as nodes. Two nodes are connected if they share, at least, one publication and the weight of an edge is computed in this way: $w(a1, a2) = 1 - J(p1, p2)$ .
Find the python code in the file *graph_creation.py*.
See the explanation of the procedure used in the (.rm) file on the GitHub repository.

*Computation time:*

```
...data loaded...
...graph creation completed...
Elapsed time:  2.0472614765167236
```

## ✓ **Second Step**: Some statistics

**1.** Given a conference in input, return the subgraph induced by the set of authors who published at the input conference at least once: Find the python code in the file *graph_creation.py*

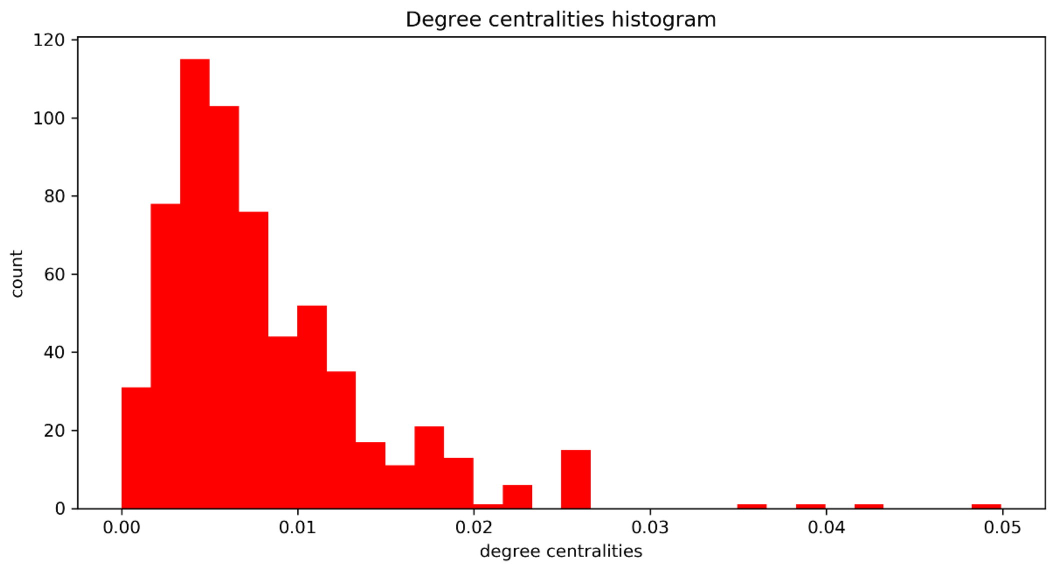
First, we ask the user to insert some conference ID:

```
***************************************************************************
 Given a conference in input, the program returns the subgraph induced
 by the set of authors who published at the input conference at least once
***************************************************************************


Insert the conference ID: 3345
```

*Computation time:*

```
...first subgraph creation completed...
Elapsed time:  0.0
```

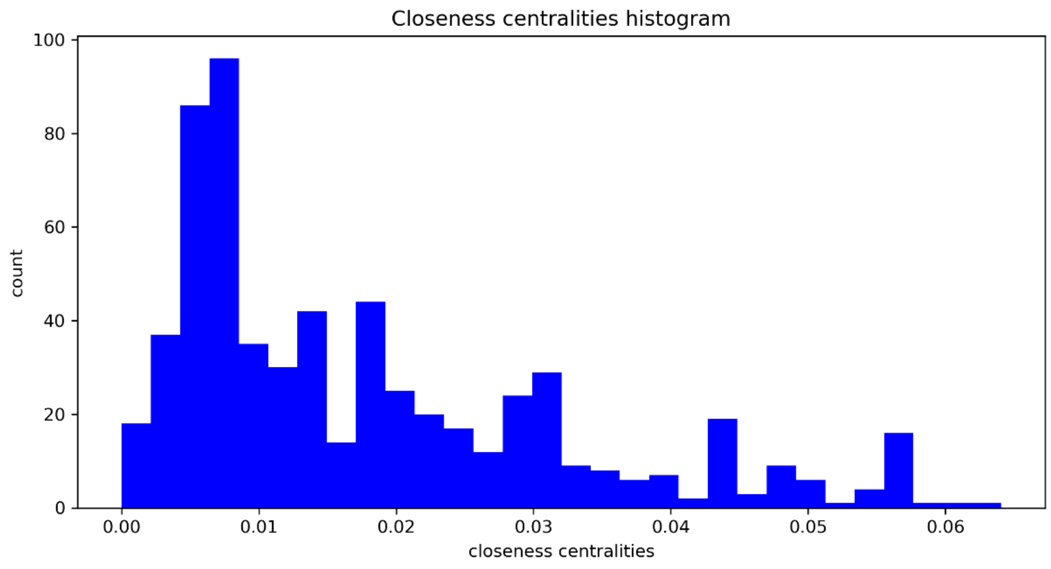
```
**************************
 Here are some statistics
**************************
```



Degree is a simple centrality measure that counts how many neighbors a node has. Thus, A node is important if it has many neighbors. From this histogram, we can state that the probability for a node to be important is quite low while the probability for a node to be less important is pretty height. (considering count as probability measure)

degree centrality = 0.005 $\Rightarrow$ count = 115 $\Rightarrow$ low importance, height probability

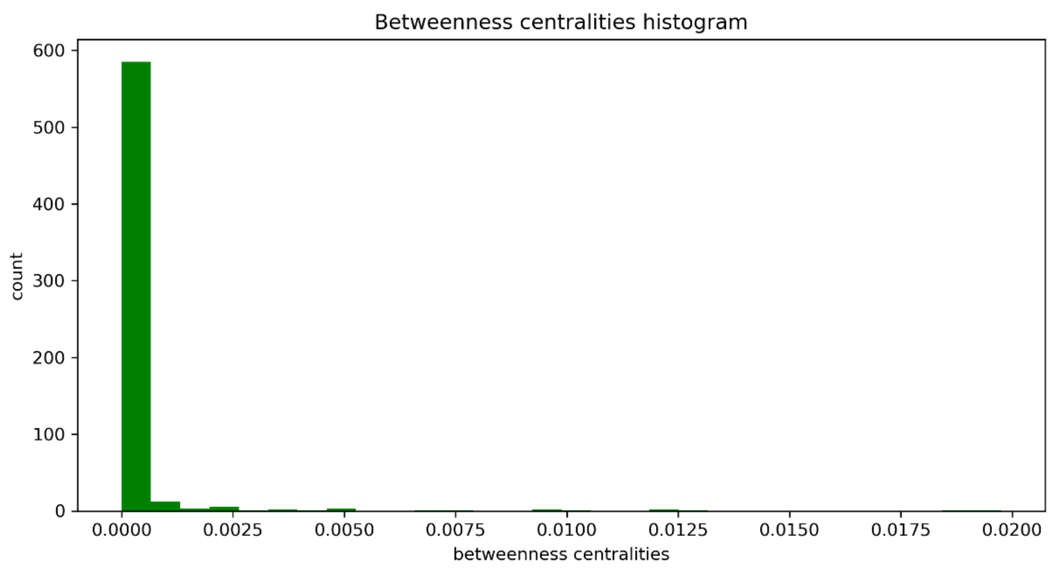
degree centrality = 0.05 $\Rightarrow$ count = 1 $\Rightarrow$ height importance, low probability



Closeness centrality measures the mean distance from a vertex to other vertices. it gives low values to more central nodes and high values to less central ones, which is the opposite of other centrality measures. Here the histogram shows that the probability for a node to be important according to the closeness centrality is height while the probability to be less important is low.

closeness centrality = 0.008 $\Rightarrow$ count = 95 $\Rightarrow$ height importance, height probability

closeness centrality = 0.06 $\Rightarrow$ count = 1 $\Rightarrow$ low importance, low probability



Betweenness centrality measures the extent to which a vertex lies on paths between other vertices. Vertices with high betweenness may have considerable influence within a network by virtue of their control over information passing between others. They are also the ones whose removal from the network will most disrupt communications between other vertices. Here looking at the histogram we can say that we are again in a case were the probability for a node to be important is quite low while the probability to be less important is pretty height.

betweenness centrality = 0.001 $\Rightarrow$ count = 580 $\Rightarrow$ low importance, height probability
betweenness centrality = 0.0050 $\Rightarrow$ count = 1 $\Rightarrow$ height importance, low probability

We can generally state that two over three centrality measures (Degree and betweenness) follow the power law (Pareto principle also known as the 80/20 rule or the law of the vital few) which state that for many events, roughly 80% of the effects come from 20% of the causes. For example, 80% of Italy's land is owned by 20% of the population.

*Computation time:*

```
...centrality measures computation done...
...histograms creation done...
Elapsed time:  1.28629469871521
```

**2.** given in input an author and an integer *d*, get the subgraph induced by the nodes that have hop distance at most equal to *d* with the input author:
find the python code in the file *graph_creation.py*.
find the *bfsr()* and *bfsr()* functions used here in the file *Libhw4.py*

First, we ask the user to insert some author ID:

```
****************************************************************************
 Given in input an author and an integer d the program returns the subgraph
 induced by the nodes that have hop distance at most equal to d
****************************************************************************


Insert an author ID: 256176
```

We also ask the user for an integer *d* which is going to be the max for the hop-distance:

```
Insert an integer d: 2
```

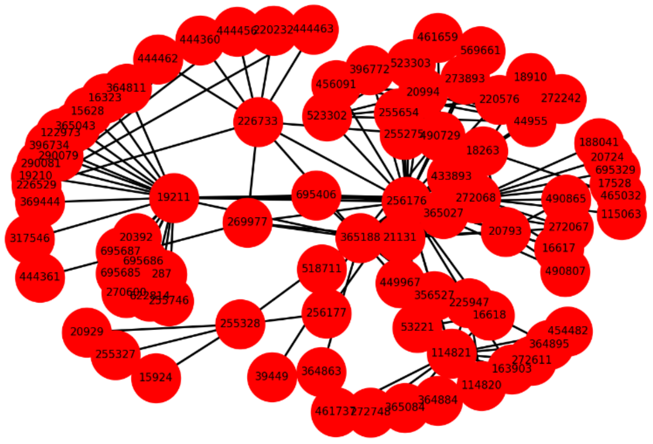
Then, we ask again the user to tell whether he wishes to use the recursive, the iterative breath first search algorithm (see the explanation of both algorithms in the (.rm) file on the GitHub repository) or even the *networkx.ego_graph()* to compute the hop-distances:

```
Choose what function you wish to use in order to compare the computation times
1 for iterative breath first search algorithm
2 for recursive breath first search algorithm
3 for networkx.ego_graph()
  essentially based on single source Dijkstra


1

...subgraph creation completed...
Elapsed time:  0.0

2

...subgraph creation completed...
Elapsed time:  0.0

3

...subgraph creation completed...
Elapsed time:  0.0
```

The outputs show that all the three algorithms have similar behavior in computation time. Thus, the reason why we decided to implement an iterative version of the bfs was indeed to avoid raising the RecursionError while going deeper into the graph.

*Output graph:*

```
*********************************
 Now let's visualise the subgraph
*********************************
```



## ✓ **Third Step**: Erdős number

**1.** Here we compute the weight of the shortest path that connects some input author with Aris. We use as measure of distance the weight of the edges.
Find the python code in the file *graph_creation.py*
Find the *shortest_path()* and *our_dijkstra()* functions used here in the file *Libhw4.py*

First, we ask the user to insert some author ID:

```
Insert the author ID: 16404
```

We then as again the author to choose which function he wishes to use between *shortest_path()* (adaptation of the bfs algorithm transforming it into a sort of dijkstra algoritm) and *nx.dijkstra_path_length():*

```
Choose what function you wish to use in order to compare the computation times
1 for Libhw4.shortest_path()
2 for nx.dijkstra_path_length()

1

...shortest path weight calculation completed...
        Elapsed time:  0.015517950057983398


2

...shortest path weight calculation using the networkx function completed...
        Elapsed time:  0.015596866607666016
```

the selected function returns in this case

```
In [67]: print(Shortest_Paths)
6.536515151515151
```

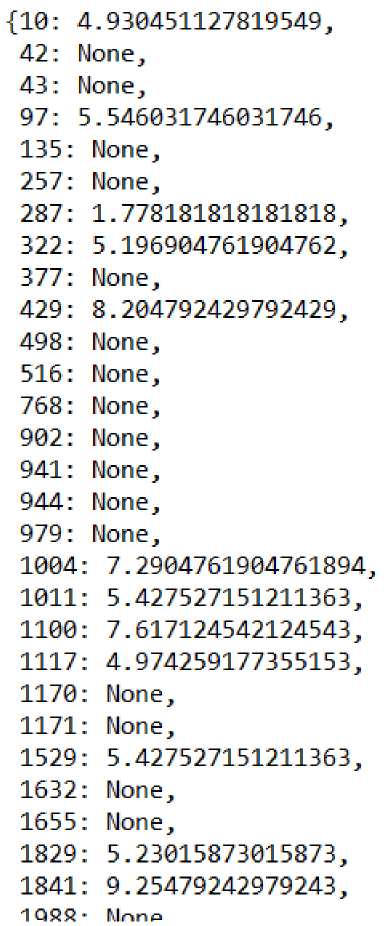
Here we notice that the two computation times are pretty close. If we neglect the availability of the processor while running these two functions, we could say that our algorithm is almost as fast as the networkx implementation of the Dijkstra algorithm.

**2.** Here we compute for each node of the graph, its GroupNumber, defined as follow:
*GroupNumber(v) = min$_{u \in I}${ShortestPath(v,u)}.*
Find the python code in the file *graph_creation.py*
Find the function *GroupNumbers()* in the file *Libhw4.py*

First, we ask the user to insert a list of author ID's:

```
Insert a list of author ID's separated by spaces
max 21 items: 256176 524380 143752
```

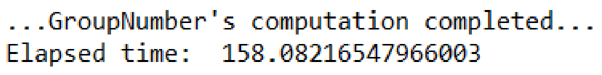
We then call the *GroupNumbers()* function which returns:



```
{10: 4.930451127819549,
 42: None,
 43: None,
 97: 5.546031746031746,
 135: None,
 257: None,
 287: 1.778181818181818,
 322: 5.196904761904762,
 377: None,
 429: 8.204792429792429,
 498: None,
 516: None,
 768: None,
 902: None,
 941: None,
 944: None,
 979: None,
 1004: 7.2904761904761894,
 1011: 5.427527151211363,
 1100: 7.617124542124543,
 1117: 4.974259177355153,
 1170: None,
 1171: None,
 1529: 5.427527151211363,
 1632: None,
 1655: None,
 1829: 5.23015873015873,
 1841: 9.25479242979243,
 1988: None
```

*None* represent the case when none of the input nodes is reachable from the corresponding node.

*Computation time:*



```
...GroupNumber's computation completed...
Elapsed time:  158.08216547966003
```