Algorithmic Methods of Data Mining -Homework 4

Caliskan-Gobbo-Galli, Group 11

In this project we carry out information from Computer Scientists network and apply some graph methodologies seen in class.
From the DBLP dataset were provided two:

- a reduced json file, which contains a part of the total network, beneficial for testing and debugging the code;
- a full one, wich is used to get the result needed.

FIRST PART:
In the first part the json file is preprocessed and from the data a graph is created.
The creation of the graph is based assuming the Authors ID as nodes and the links between them are weighted following the Jaccard Distance (obtained by subtracting the J.Similarity to 1):

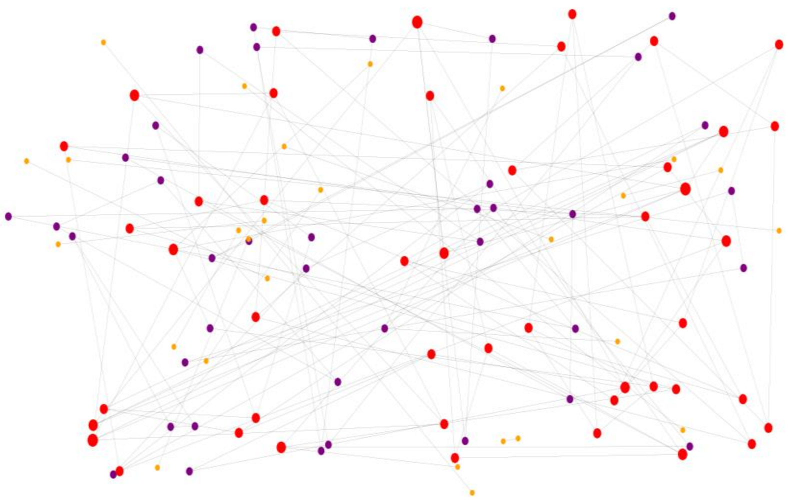
$$w(a1, a2) = 1 - J(p1, p2),$$

where:

- $a1, a2$ are the authors,
- $p1, p2$ are the set of the two author's publications,
- $J(p1, p2)$ is the Jaccard Simlarity (also known as intersection over union, that measures similarity (and diversity) beetween finite sample sets; $0 \leq J(p1, p2) \leq 1$, if the sets are both empty it is defined equal 1).

STATITSTICS:
2a
In this part various functions were computed to get and visualize subgraph and centrality measures.
The purpose of this subgraph is to show, given a Conference ID as input, how many Authors pubblished in it.

This is the subgraph related to the Conference ID 3052. The different color and size of nodes reflects the different degree related.

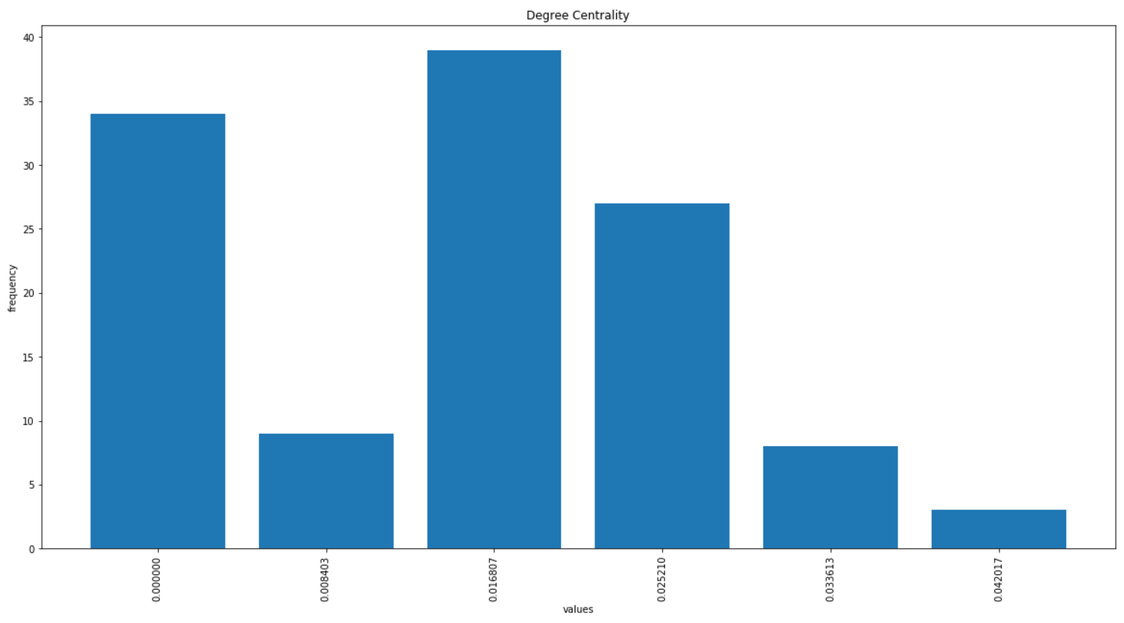


The centralities considered, and centrality is interpret as how central the node is, are:

- Degree Centrality
- Closeness Centrality
- Betweeness Centrality

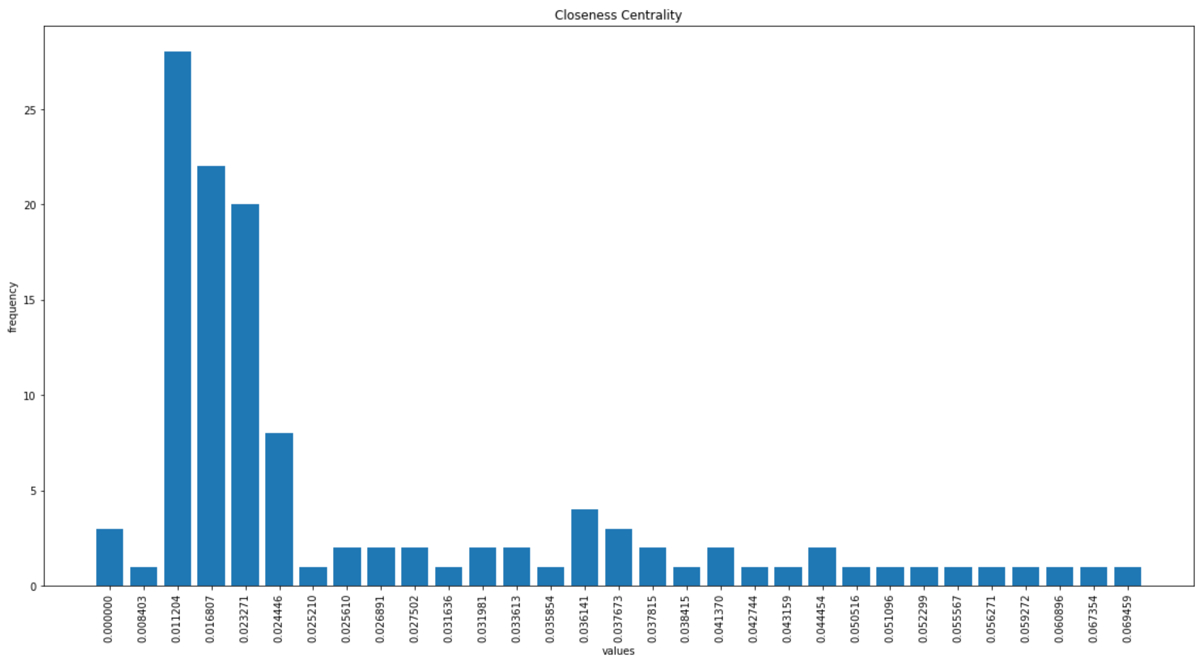
Degree Centrality

The degree centrality for a node is the fraction of nodes it is connected to. As assumption important nodes has more connenction; the `nx.degree_centrality()` function returns values for each node, in this way is possible to define nodes that are more important in the network. Analyzing the results is imaginable to find who holds more information or find the most connecting individuals. As we can expect, in this example, the important nodes (highest value) are few.



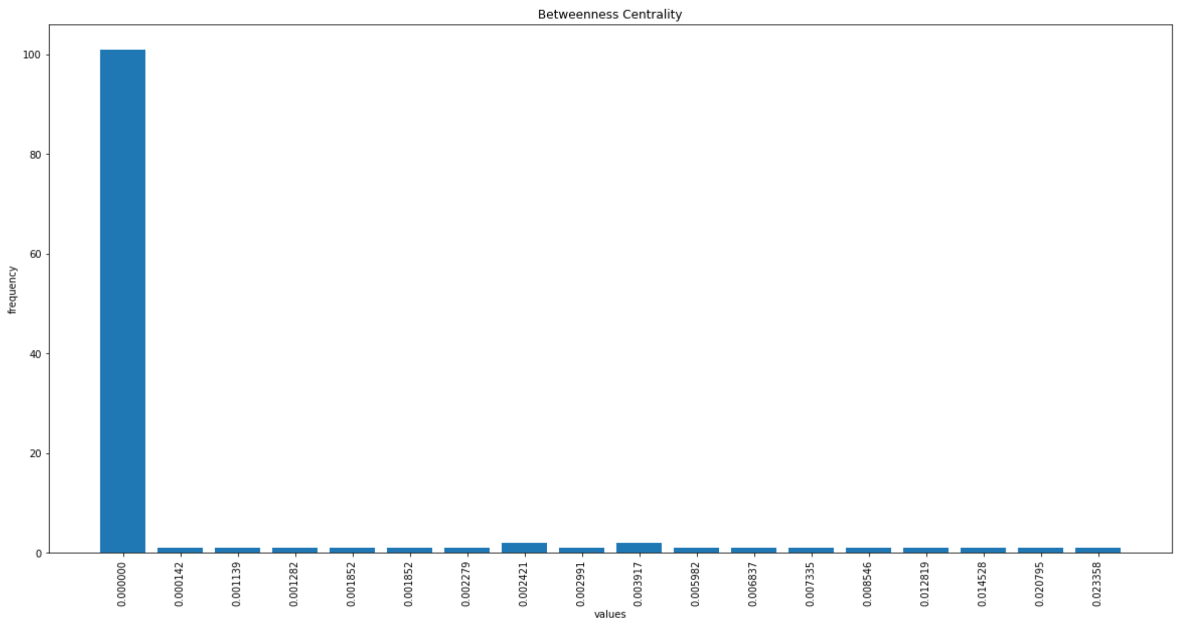
Closeness Centrality

As assumption important nodes are identify with the one more close to other nodes. The `nx.closeness_centrality()` function is calculated as the total number of nodes a node X can reach devided by the sum of the shortest of node X. Through this is possible to find individuals who are best placed in the network, and can influence it, in a faster way.



Betweenness Centrality

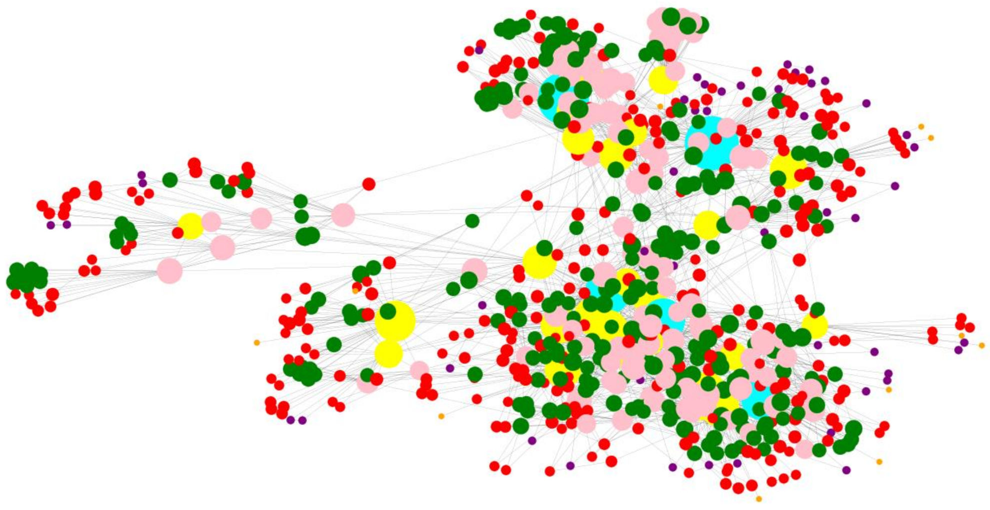
The importance of a node depends on how many shortest path, between all the possible paths, pass through a node X. As we can see, there are a lot nodes with a very low value (0.000), which means that shortest paths do not pass in alot of nodes, so the important nodes, taking in account this definition, are not so much.



2b

In this part the hop distance is compute given an Author ID and an integer that identify how many 'hop' are desired. For instance if the integer inserted is equal to 3 our result will show, starting only from one node (the Author ID entered), to which nodes it is link, and then the code will do the same process again for these nodes, until the number of the hop needed (3) is reached.

In this example the input given was 256176 (Aris's ID) and we calculate an hop distance equal to 2.

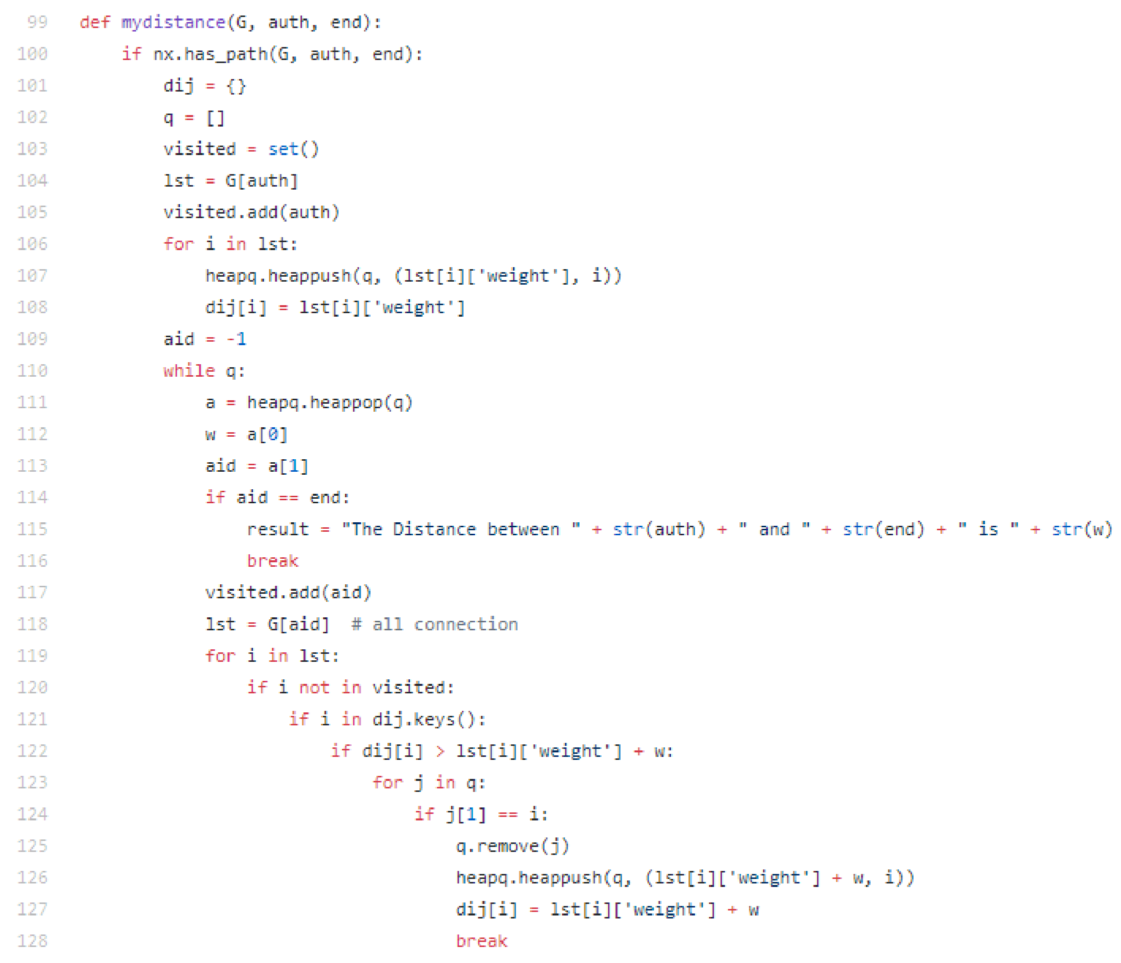


THIRD PART

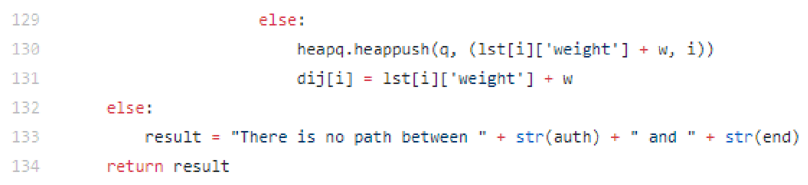
The part *a*, of the last part of the project, is about to find the shortest path between an Author ID, taken as input, and a specific end node (Aris's node). The measure of this shortest path is calculated and define by the weight edge defined in the first part of this project.
To achieve this goal we compute the Dijkstra's algorithm.
We define the function `mydistance(G, auth, end)` that takes as input a graph, an Author ID and the final node which has to be reached.

We decided as first thing to check if there is any path, with the networkx function `nx.has_path`; after that we put in a list all the node's neighbour and insert the input node in a set.
With the first *for* (*line 106*) we go through the list (*lst*) and we will push in a heap a tuple composed by the weight of the node *i* and the node itself (*w[i]*, *i*), and at the same time we create a dictionary with node *i* as key and weight as value.

The *while* loop (*line 110*) run untill *q* is empty; the idea is to take from the heap the smallest value, from that define the weight (*w*) and the Authors ID (*aid*). If this Authors ID is our end node, we will get the result and it breaks (this is because we don't have negative weights). Otherwise if it is not our final node, it will be insert in the *visited* set.
Again we will insert the neighbour of this node (*aid*'s neighbours) in a list and iterate through it. If the node taken into account is not in the set, we will see if this node is in the dictionary created before, and only if is inside and smaller than the previous one, this value will be changed in the dictionary and in the queue. Otherwise the weight will be calculated.



```
 99  def mydistance(G, auth, end):
100      if nx.has_path(G, auth, end):
101          dij = {}
102          q = []
103          visited = set()
104          lst = G[auth]
105          visited.add(auth)
106          for i in lst:
107              heapq.heappush(q, (lst[i]['weight'], i))
108              dij[i] = lst[i]['weight']
109          aid = -1
110          while q:
111              a = heapq.heappop(q)
112              w = a[0]
113              aid = a[1]
114              if aid == end:
115                  result = "The Distance between " + str(auth) + " and " + str(end) + " is " + str(w)
116                  break
117              visited.add(aid)
118              lst = G[aid]  # all connection
119              for i in lst:
120                  if i not in visited:
121                      if i in dij.keys():
122                          if dij[i] > lst[i]['weight'] + w:
123                              for j in q:
124                                  if j[1] == i:
125                                      q.remove(j)
126                                      heapq.heappush(q, (lst[i]['weight'] + w, i))
127                                      dij[i] = lst[i]['weight'] + w
128                                      break
```



```
129                      else:
130                          heapq.heappush(q, (lst[i]['weight'] + w, i))
131                          dij[i] = lst[i]['weight'] + w
132      else:
133          result = "There is no path between " + str(auth) + " and " + str(end)
134      return result
```

For the second part (*3b*) we changed the previous algorithm in the way we can define a GroupNumber for any nodes, that returns the minimum shortest path for each graph's node to a subset of node (maximum 21).
We define `calcGrNr(mlst, T)` function, that takes in input a list of Authors ID (*mlst*) and the graph.
With two *for* we will iterate at first through the graph's nodes and then, only if these nodes are not in the list, taken as input, through the element in the list; with the same function as before is checked if there is any path for any graph's nodes and the one in the list.
As the previous one now the distances are calculated.
In *line 164 →166* is considered, also, the peviouus calculated distance to avoid repetition in calculations and again for every nodes we put the distances to the subsets in a heap.
In *line 182* we put in a heap all the paths that goes from a generic node *i* to all end nodes and later from this heap we will get the minimum (in *line 185*).
The result will be shown as a dictionary with node as key and the GroupNumber as value.