Dealing with Networks

December 22, 2017

1 Homework 4

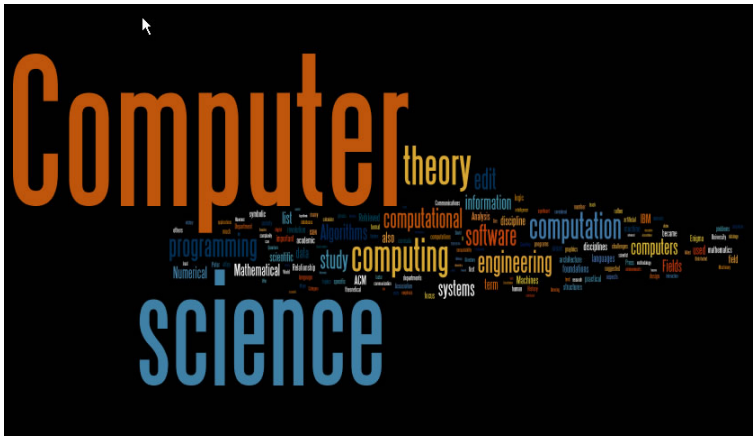
Data Science - Algorithmic Methods of Data Mining fall 2017

1.0.1 Group 4

Authors:

- *Alfonso D'Amelio* → 1658170
- *Eren Besli* → 1738128

Main Focus: Dealing with networks and carry out information from this.
Dataset: Computer Scientists publication.



1.0.2 Through this pdf file, we will explain and comment the results obtained in the project.

First of all we imported our code from the *modules.py* file and we created a data structure that would allow us a quick and simple implementation of our graph So then we built-up our network

```
In [1]: import modules
        import networkx as nx
        modules.buildDataStructure()
        G = nx.Graph()
        G = modules.buildGraph()
```

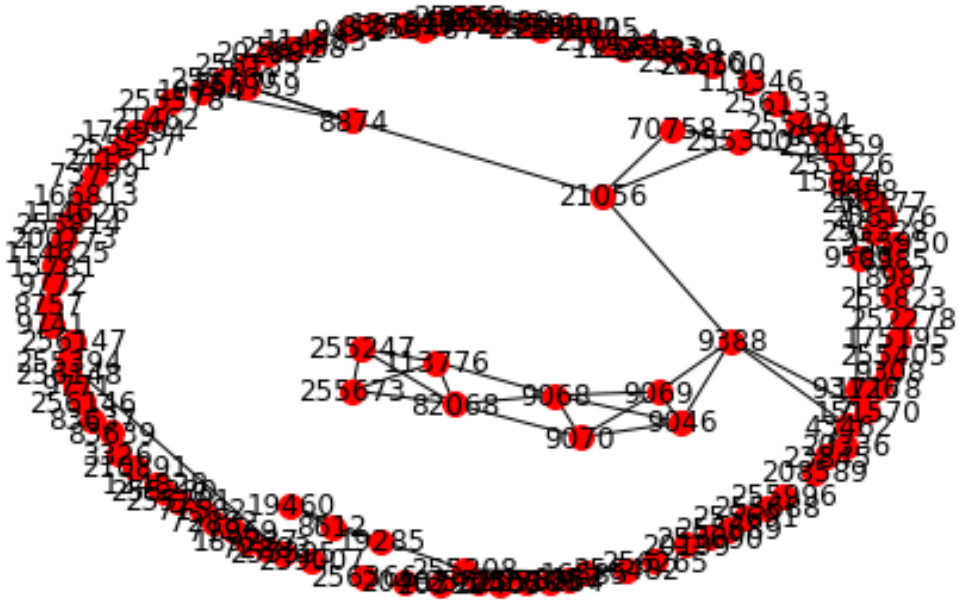
The results shows up that our network-graph is composed by:

- 904.664 nodes
- 3.679.276 edges

At this point we move to the point 2A of homework, in which it is required, given a conference in input, to return the subgraph induced by the set of authors We choose like an input the conference n° 3052, we extract all the authors that joined that conference and we create a subgraph of the entire network, visualizing it

```
In [2]: #given a conference ID return the subgraph induced by the set of authors who published
        conf=int(input())
        H = G.subgraph(modules.searchConfId(conf))
        modules.plot_subgraph(H)
```

3052



With the following attributes:

- 120 nodes
- 129 edges

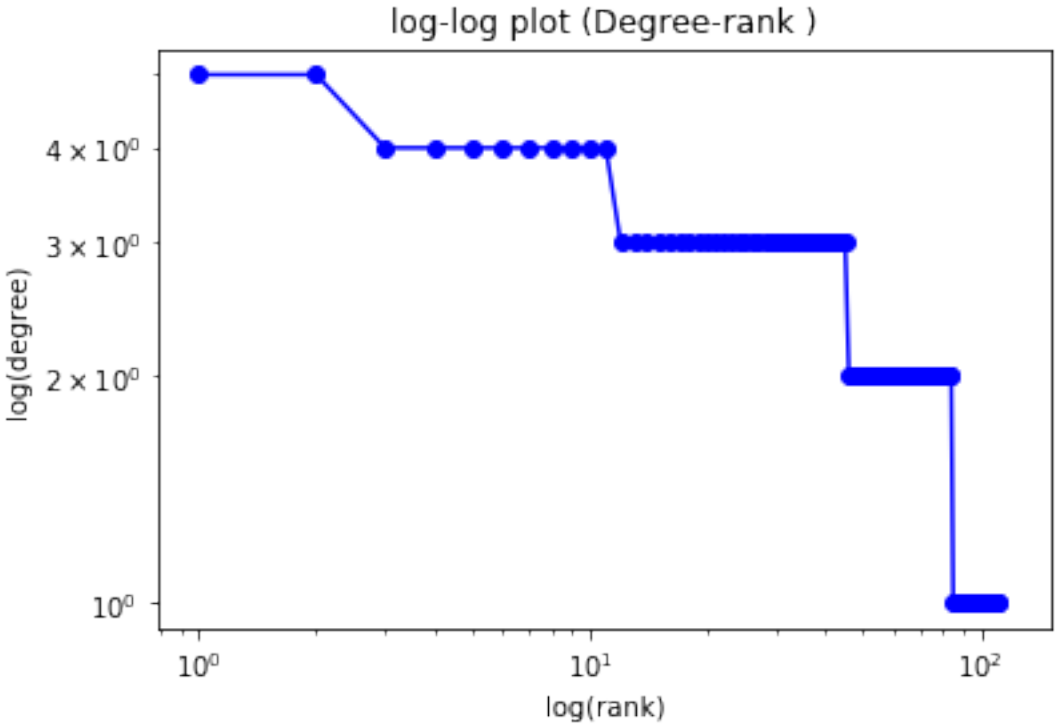
Then on this subgraph we computed some centrality measure:

- Degree
- Betweenness
- Closeness

1.0.3 Degree centrality

which is defined as the number of links incident upon a node

```
In [3]: modules.loglog_plot(H)
```



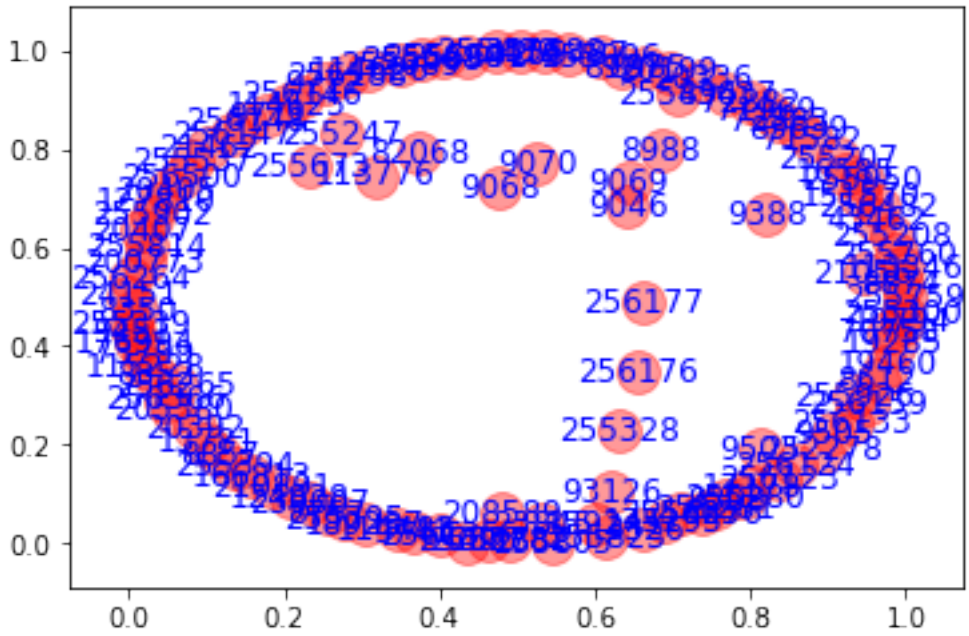
In the log-log plots we can see how the trend follows the power law. Since our network is not random, the Poisson distribution is not a good fit for the data.

1.0.4 Betweenness centrality

Betweenness centrality quantifies the number of times a node acts as a bridge along the shortest path between two other nodes.

Given that the output *nx.betweenness_centrality(G)* is a dictionary with keys equals to the nodes of the subgraph and values equals to the corresponding betweenness centrality, we create a function called *most_important* which returns the nodes with highest betweenness centrality measure. So higher the betweenness of a node more control this node has over the network

```
In [4]: Gt= modules.most_important(H)
        modules.plot_betwenness(Gt)
```

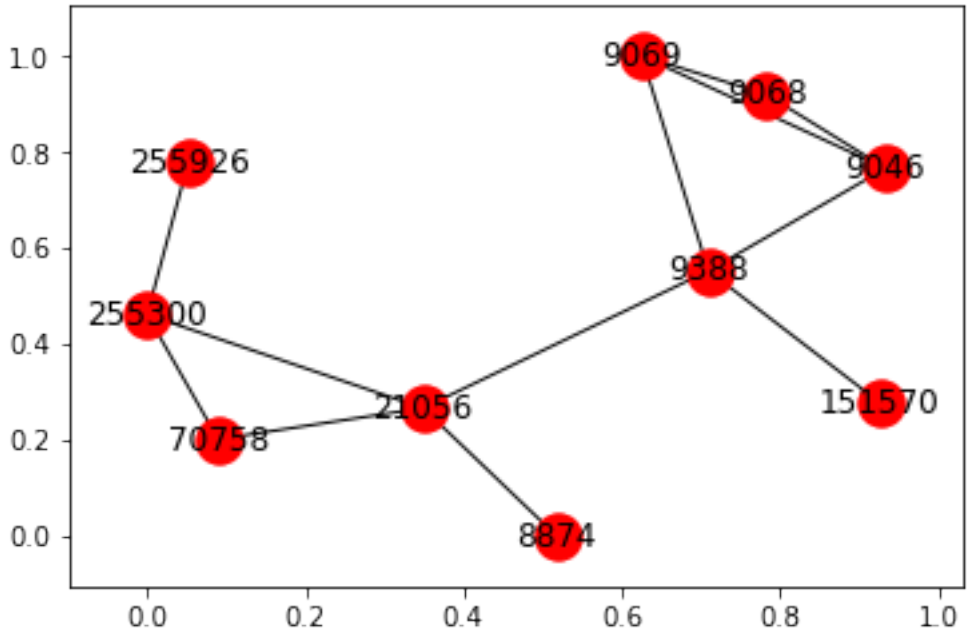


1.0.5 Closeness centrality

In a connected graph, the normalized closeness centrality (or closeness) of a node is the average length of the shortest path between the node and all other nodes in the graph. Thus the more central a node is, the closer it is to all other nodes.

Even for the closeness centrality we create a function which returns the top ten nodes with highest measure, and we visualize the subgraph induced by this node from the starting subgraph

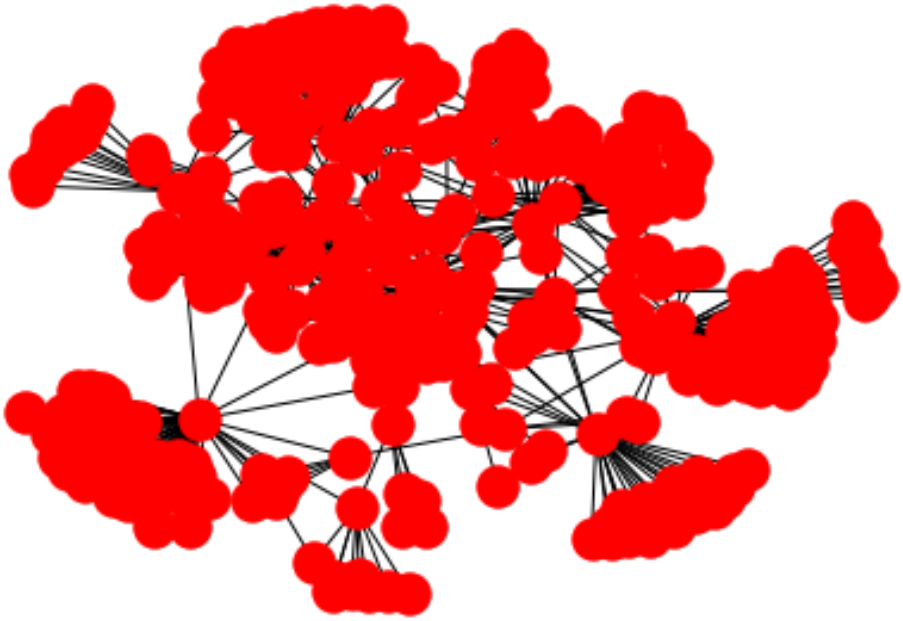
```
In [5]: modules.top_clos_node(H)
```



1.0.6 Hop distance

After these centrality measures and statistics visualization, we compute the Hop Distance given in input an author and an integer d, we get the subgraph induced by the nodes that have hop distance (i.e., number of edges) at most equal to d with the input author

```
In [22]: modules.hopDistance_nx(G,93126,2)
```



1.0.7 Shortest Path

Once the hop distance has been implemented, we move on to point 3, in which we explicitly ask to implement the shortest path from a node of the graph to Aris. in the code we implemented we created two Shortest Path functions:

- ShortestPath
- ShortestPath_heap

Dijkstra's algorithm uses an array of labels L[n]. During the computation the set *V* is partitioned into two subsets: *S*, which contains the nodes with permanent label, i.e the nodes i for which the label L[i] expresses the cost of a minimum path from s to i; *V§*, which contains the nodes j for which L [j] constitutes a temporary label, i.e it expresses an upper limit on the cost of a minimum path from s to j. Initially, only the origin node* s* **is permanently labeled and its permanent label is 0. All other nodes have temporary labels that are worth +inf and are stored in a Q list. At each iteration the algorithm removes from N the node i corresponding to the minimum temporary label which becomes permanent (thanks to the validity of the theorem) and then explores the outgoing star FS (i) of i, in order to eventually improve the temporary label of the nodes adjacent to i.

why we also implemented the one with the heap? because we realized that with the first function we implemented the execution time was very high, because for each node calculates the minimum path to get to all those to which it is connected. By documenting on the web and comparing our ideas with those of the other groups, we assumed that implementing Dijkstra with another technique was a faster solution.

For example, one of these implements Q as a binary heap. In this case, the Q heap must be built during initialization. At each iteration the extraction of the node i = Q [1] takes place in constant time, but at the extraction it must follow a re-heapification operation of Q and this operation costs $O(\ logn)$

the outgoing star FS (i) is inspected and for each node j adjacent to i for which the temporary label L improves, [j] the position of j in the heap Q must be updated. Therefore, total complexity becomes:

$O(m \cdot logn)$

```
In [6]: Aris_id=256176
        modules.shortestPath_heap(G,Aris_id, 16617)
```

```
Out[6]: 0.9696969696969697
```

1.0.8 The last step is to write a Python software that takes in input a subset of nodes (cardinality smaller than 21) and returns, for each node of the graph, its GroupNumber

```
In [7]: #group=[93126,256176]
        group=list(map(int,input().split()))
```

```
In [ ]: groupedList = modules.findGroupNodes(group,G)
```

This is the output in the reduce data set :

- groupnode = 93126

```
Out[13]: {93126: [dict_keys([20405]),
                  dict_keys([20407]),
                  dict_keys([255405]),
                  dict_keys([9308]),
                  dict_keys([175195]),
                  dict_keys([17178]),
                  dict_keys([255537]),
                  dict_keys([21462]),
                  dict_keys([255688]),
                  dict_keys([255689]),
                  dict_keys([255690]),
                  dict_keys([255691]),
                  dict_keys([15924]),
                  dict_keys([93126]),
                  dict_keys([93126]),
                  dict_keys([23845]),
```

- groupnode = 256176

```
256176: [dict_keys([20405]),
 dict_keys([20405]),
 dict_keys([20407]),
 dict_keys([20407]),
 dict_keys([255405]),
 dict_keys([255405]),
 dict_keys([9308]),
 dict_keys([9308]),
 dict_keys([175195]),
 dict_keys([175195]),
 dict_keys([17178]),
 dict_keys([17178]),
 dict_keys([255537]),
 dict_keys([255537]),
 dict_keys([21462]),
 dict_keys([21462]),
 dict_keys([255688]),
```

to reach Aris group run this:

- modules.printArisGroup(G)