1 EXCERCISE - IMPORT DATA FROM JSON FILE

We used the standard procedure to open the json file in a "read" mode first (line 10), and then stored the file_data in the variable data. data, from now onwards, will be our row file containing all the unstructured data. Line 12 close the reading procedure.

The idea is to create 3 back-up struture (dictionaries), used as support while esecuting the script. In details these are the structures for each dictionary:

CONFERENCES : {" *id_conf_int* " : {"conf":" *id_conf* ", "title":" *title_conf* ", "pub_int":" [*id_pub_int_1*, *id_pub_int_2*, *id_pub_int_3*, ...]}}

AUTHORS : { " *aut_id*" : { "name":" *author_name* ", "pub":[*pub_int_1*, *pub_int_1*, *pub_int_1*,....]}}

PUBLICATIONS : { " *id_pub_int* " : { "pub":" *id_pub* ", "authors":[*aut_id_1*, *aut_id_1*, *aut_id_1*,...]}}

To do this we started looking in each row line (first for loop) and picking from the raw data the values under the ***id_conference_int*** and ***id_publication_int*** labels and stored as key of the dictionaries ***conferences*** and ***publications*** respectively. Those values were chosen for their unicity(??) and for a faster research in our back-up dictionaries. Stored the keys the dictionaries are now filled: Values under ***id_conference***, ***title***, and ***pub_int*** labels of the row data are stored as values of ***conf***, ***title*** and ***pub_int*** in ***conference*** dictionary; ***id_publication*** values are stored in ***pub*** key in ***publications*** dictionary together with an empty list titled ***authors***.

Scraping the row data we focus now to the ***authors*** section, adding each authors to the previous empty list. We want to fill the ***authors*** dictionary as well so we need first an IF condition to check if the author already exists as key or not and then choose if create a new instance for the author (if not already present) or just update it (the instance) with a new publication. If it's a new entry we create also a new node of the graph G with the ***aut_id***.

Once executed the first for loop we have, until now, the three dictionaries and only the nodes of the final graph. According to the request the edges in the graph have to link only authors that share at least one publication; for this reason we rely on the second for loop to get inside the ***publications*** dictionary created before to take a look at the authors that partecipate at the same publication.

If more than one author worked to the publication we start focusing on the list of the authors. Given that that authors are already sharing one publication (exactly the publication we are looking at) it'll be created an edge that link each writer to the other ones. However the edge between to authors must be weighted and according to the request, so..taken two authors we start looking at their publication using the ***authors*** dictionary and, once got into the list of publication we calculate the Jaccard similarity between the list. The result of the Jaccard similarity it'll be the weight of the edge (line 63)

2 EXCERCISE - STATISTICS AND VISUALIZATIONS

We ask the user to type the query, in particular, in the first case, we ask for an ***id*** of the conference in order to retrieve a subgraph induced of the authors that have published at least one work in the given conference, and in the second case, we ask for an ***id author*** and a ***d*** value in order to execute the algorithm and get the subgraph induced of all the authors linked to the input author that have at least hop distance equal to d.

First case.
Once acquired the query typed by the user we start looking inside the ***conferences*** dictionary accessing into the values of the given conference and we focus in ***"pub_int"*** to see which are the publications discussed in our conference. As we are interested in the authors that took part at each publication, and as a consequence at the conference as well, we dig a little bit down to the ***publications*** dictionary (in particular to the list of the ***authors*** with the second for loop) once acquired the id of the publication in the first for loop. All the authors that satisfy the request are stored in a list of authors; list then used to create the subgraph of G using the appropriate method ***G.subgraph*** . To plot the result we get, with a for loop, the degree of each node of the subgraph that we use then in the ***.draw*** method to plot the network putting in evidence the nodes linking their size to their degree. To get the degree, closeness and betweennes centrality we used the built-in function of networkx package (***.degree_centrality***, ***.closeness_centrality***, ***.betweenness_centrality*** respectively) and plotted the results using the plt.histogram.

Second case.
Once acquired the ***author id*** and the ***hop distance*** typed by the user we create the subgraph induced as explained in the introduction. To deal with this task we used an other built-in function, ***.egograph***, that take as variable the original graph, the hop distance and _radius??_ Again we rely on the degree of each node to draw properly the graph adapting the size of each node to the degree.

3 EXCERCISE - GENERALIZED VERSION OF ERDOS NUMBER

3a)
For point 3a, we implemented a function that replicates Dijkstra algorithm for shortest path. This function takes as input a source node and a target node (in this case is always id of Aris), and is divided into 3 steps:

Step 1) This is the instancing step:
G_neighbors: a matrix that stores, for every node, its neighbours and the weight of each edge reached: a boolean variable, set to False, that represents if the target node is reached.
temp_distances: a dict that contains, for each node, the temporary weight of the shortest path from the source (for source is 0).
final_distances: a dict that contains the weight of shortest path from the source. It starts with the initial node, with a value of 0.
t_distances: a heap with tuples, in the way (weight of path from source, node). It starts with all the neighbours of source node and their corresponding edge weight.
J: a dict, for every node it stores its predecessor in the shortest path. It starts with None values for every node in the graph, 0 for source and source for every neighbours of source node.

Step 2) Find the nearest node:
In this step, an element is popped out from ***t_distances*** (surely it is the one with the minimum distance). If the node related has been already been processed (is in ***final_distances*** keys), another element is taken from ***t_distances***. When a proper node is found, its ***final_distance*** is the first element of the tuple; if its the target, the algorithm stops.

Step 3) Update distances:
For every node neighbour of the node selected in step 2 that has not been processed, it is computed its distances from source going through step 2-node. If it's lower than the one in ***temp_distances***, this is updated, along with its predecessor in J, and weight of the path and node are pushed into the heap.

Step 2 is then repeated.

Final Step):
If target node was not reached, 'PATH NOT FOUND' is printed.
Otherwise, path from source to target and its weight are returned as a tuple. (Although path is not requested, this algorithm seems more complete).
Only the weight of the shortest path is printed.
There is also a check before the execution of the function, to verify that author id of the source node is in the database and this does not belong to Aris.

3b)

For this part, we created another function that is simply a modified version of the function used in part 3a. This new function, named ***multi_source_shortest_path_alg***, takes as argument a list of sources (these are the input nodes).

In Step 1, the difference is that in ***t_distances*** are pushed the neighbours of all the sources and their corresponding distance from them (if any node is neighbour of more than one source the nearest source is its predecessor).

Step 2 and 3 need no changes.

A dict is returned, the keys are all the nodes in the graphs and the values are tuples where the first element is the shortest path, and the second is the weight of said path. As before, although the path itself is not needed, this way the algorithm feels more complete.
Before applying the function, it is check if all the input nodes are in the graph.

It was also feasible to write a single function for both points, but we felt the code to be more readable this way.