Report

December 22, 2017

1 Report

Giulia Rinaldi, Michele Di Muccio, Mohammad Khodaygani

1.0.1 Part 1: Generation Graph

generator_graph.py The import for all modules are the following:

```
In [1]: import networkx as nx
        import json
        from matplotlib import pyplot as plt
        import matplotlib.patches as mpatches
        from matplotlib import pylab as pl
        import matplotlib.patches as mpatches
        from operator import itemgetter
```

In this module we have two function: - visualize_graph: Allowing to visualize a graph

```
In [2]: def visualize_graph(graph):

            pl.figure()
            nx.draw_networkx(graph)
            pl.show()
```

- generate_graph: Builting the graph on a a jason file

```
In [3]: def generate_graph(path):

            ## Parsed of the data
            url = path
            with open(url) as jsonfile:
                data = json.load(jsonfile)

            ## Definition of the dictionaries
            # dictionary : keys -> id_authors, values -> authors with common publications
            dic_auth = {}
            # dictionary: keys -> id_authors, values -> important attributes
            dic_conf = {}
```

```
# dictionary: keys -> id_authors, values -> publications
dic_pub = {}

## Loop on data to take the important info
for entry in data:
    authors = entry['authors']
    id_conf, title_conf, id_pub = entry['id_conference_int'], \
    entry['id_conference'], entry['id_publication_int']
    ident, names = zip(*[(auth['author_id'], auth['author']) \
                         for auth in authors])
## The keys of the dictionaries are the author_ids
    for i in range(len(ident)):
        id_key, id_name = ident[i], names[i]
        lst = set(ident) - {id_key}
        conf = {'author': id_name, 'id_conference': (id_conf, title_conf)}

        try:
            dic_auth[id_key] = dic_auth[id_key].union(lst)
            dic_conf[id_key]['id_conference'].append(conf['id_conference'])
            dic_pub[id_key].append(id_pub)
        except:
            dic_auth[id_key] = lst

            conf['id_conference'] = [conf['id_conference']]
            dic_conf[id_key] = conf
            dic_pub[id_key] = [id_pub]

## Definition of graph by dictionary of authors
G = nx.from_dict_of_lists(dic_auth)

## Setting of attributes per each node
for u, v, d in G.edges(data=True):

    G.node[u]['author'] = dic_conf[u]['author']
    id_conf, id_conf_int = zip(*dic_conf[u]['id_conference'])
    G.node[u]["id_conference_int"], G.node[u]["id_conference"] = \
    list(id_conf), list(id_conf_int)
    G.node[u]["id_publication_int"] = dic_pub[u]

    G.node[v]['author'] = dic_conf[v]['author']
    id_conf, id_conf_int = zip(*dic_conf[v]['id_conference'])
    G.node[v]["id_conference_int"], G.node[v]["id_conference"] = \
    list(id_conf), list(id_conf_int)
    G.node[v]["id_publication_int"] = dic_pub[v]

    a, b = set(dic_pub[u]), set(dic_pub[v])
    d['weight'] = 1 - len(a.intersection(b)) / float(len(a.union(b)))
```

```
            ## List with isolated nodes
            list_no_edge = nx.isolates(G)

            # Add attributes to isolated nodes
            for u in list_no_edge:
                G.node[u]['author'] = dic_conf[u]['author']
                id_conf, id_conf_int = zip(*dic_conf[u]['id_conference'])
                G.node[u]["id_conference_int"] , G.node[u]["id_conference"] = \
                list(id_conf), list(id_conf_int)
                G.node[u]["id_publication_int"] = dic_pub[u]

            return G
```

In *main.py* we call these function to generate and visualize the graph.

```
In [4]: G = generate_graph('/Users/MicheleDiMuccio/Desktop/AMD_HM4/reduced_dblp.json')

        print('The number of nodes is ', nx.number_of_nodes(G))
        print('The number of edges is ', nx.number_of_edges(G))
```

```
The number of nodes is  7771
The number of edges is  16488
```

1.0.2 Part 2: Statistics

statistics.py There are two functions:

- stat_conference:

```
In [5]: def stat_conference(graph, conference):
            nodes = []
            try:
                str(conference)
                nodes = [p for (p, d) in graph.nodes(data=True) if conference in \
                         d['id_conference']]
            except:
                nodes = [p for (p, d) in graph.nodes(data=True) if conference in \
                         d['id_conference_int']]

            sub_graph = graph.subgraph(nodes)

            ## Plot graphs
            visualize_graph(sub_graph)

            ## Statistics

            degree_seq = sorted(nx.degree(sub_graph).items(), key=itemgetter(0) )
```

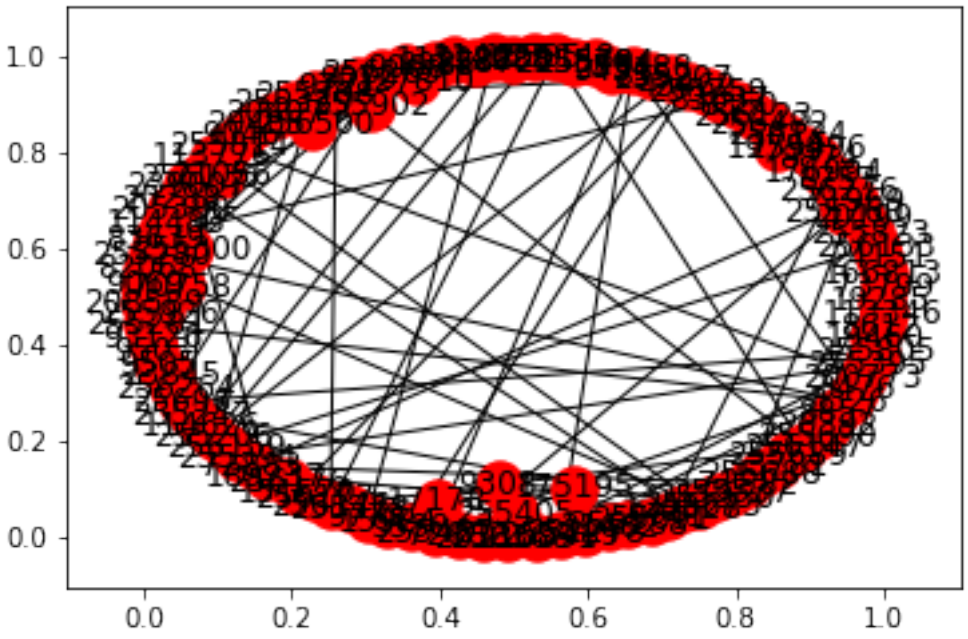
```
    nodes_lst, degree_lst = tuple(zip(*degree_seq))

    clos = sorted(nx.closeness_centrality(sub_graph).items(), \
                  key=itemgetter(0))
    closeness_lst = tuple(zip(*clos))[1]

    betw = sorted(nx.betweenness_centrality(sub_graph, \
                                            normalized=False).items(),\
                  key=itemgetter(0))
    betweeness_lst = tuple(zip(*betw))[1]

    ## Legend
    blue_patch = mpatches.Patch(color='blue', label="Degree")
    orange_patch = mpatches.Patch(color='orange', label="Closeness")
    green_patch = mpatches.Patch(color='green', label="Betweeness")

    ## Plot Statistics
    f, ax = plt.subplots(1)
    ax.plot(nodes_lst, degree_lst)
    ax.plot(nodes_lst, closeness_lst)
    ax.plot(nodes_lst, betweeness_lst)
    ax.xaxis.set_major_formatter(plt.NullFormatter())

    plt.title("Statistics' plot")
    plt.legend(handles=[blue_patch, orange_patch, green_patch])
    plt.show()
```

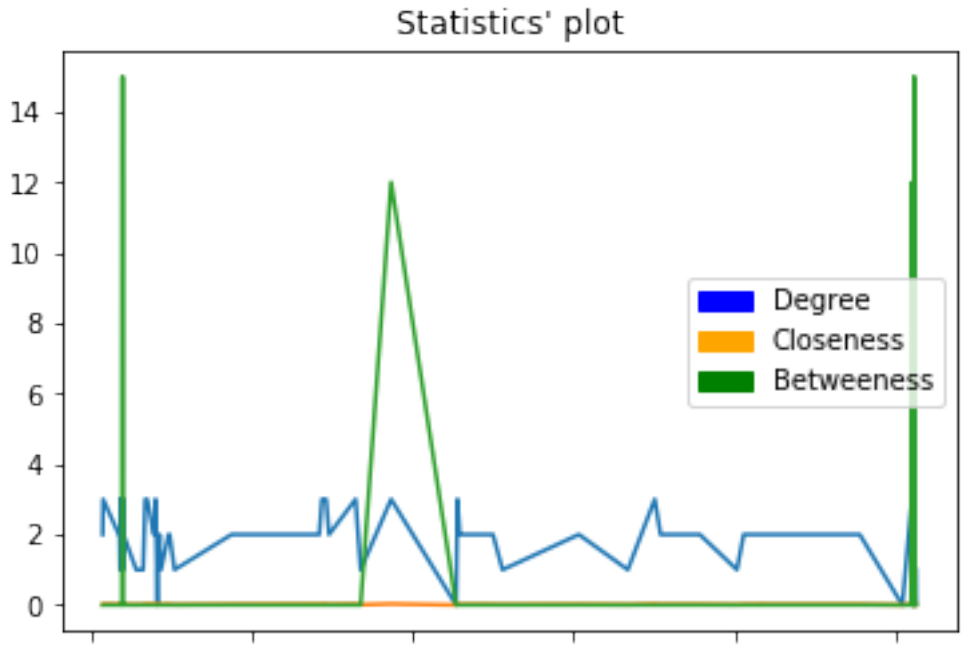
- stat_authors:

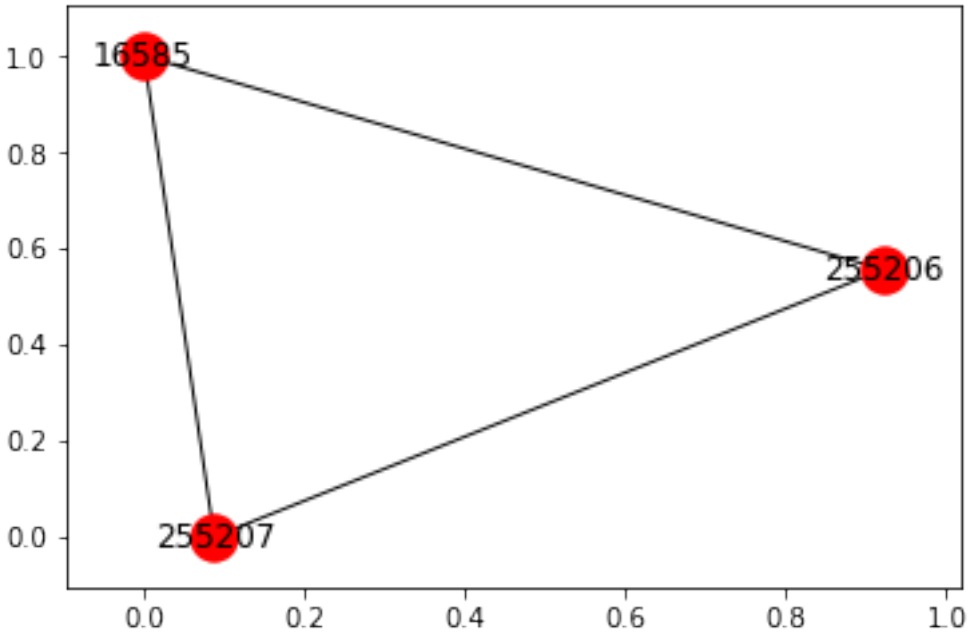
```
In [6]: def stat_authors(author, number, G):

    try:
        str(author)
        source = [p for (p, d) in G.nodes(data=True) if d['author'] \
                  == author][0]
    except:
        source = author

    ## Selection of the paths that respect the constrition of d
    p = nx.shortest_path(G, source=source)
    p_path = {k: v for k, v in p.items() if len(v) <= number + 1 \
              and len(v) > 1}

    ## Construction of the sub_graph
    nodes_p_path = list(p_path.keys())
    sub_graph = G.subgraph(nodes_p_path)

    ## Visualization of the graph
    visualize_graph(sub_graph)
```

In *main.py* we call these function to generate and visualize the graph.

```
In [7]: stat_conference(G, 'conf/pkdd/2011-1')
        stat_authors('mohammed j. zaki', 22, G)
```







In the first image we visualize the reduced graph with its edges and nodes

The second image shows a trend of the solution for the specified case. As we can see the closeness is very low, meaning that the only few pubblications are similar to each other; the betweeness rapresents a mean value. The degree does not follow any path because the publications are random (in the case of the attachement process the graph would have been logaritmic).

In the third graph we shows 3 authors which have a small distance between them (this means that they share some publications).

1.0.3 Part 3: Generalized Erdos Number

generalized_E_number.py

```
In [8]: import heapq
```

Also in this module, we can find two functions:

- dijkstra:

```
In [9]: #we create a heap structure, through the heapq lib, in order
        #to compute the algorithm faster
        import heapq

        def dijkstra(graph, initial, end=None):
            #we create a dictionary in which we store the visited nodes
            visited = {initial: 0}
            #we create a list, initializing the starting node with weight 0
            h = [(0, initial)]
```

```
        while h:
            #we obtain through the heappop command the value of h and
            #then is popped out of the list
            current_weight, min_node = heapq.heappop(h)
            #for each neighbour(v) of a node we go through the minimum weight path
            for v in graph[min_node]:
                weight = current_weight + graph[min_node][v]['weight']
                #if we find a shorter path using another node we update the weight
                if v not in visited or weight < visited[v]:
                    visited[v] = weight
                    heapq.heappush(h, (weight, v))
        #if no ending node is specified we compute the generalized dijkstra
        if end == None:
            return visited
        #in the case that we specify the ending node we retrieve only the
        #weight of the shortest path
        else:
            return visited[end]
```

We can observe that the running times are close.

```
In [10]: # In this part we do a consideration on the execution time for the
         #dijkstra algorithm that we wrote and the other implemented by

         # the library networkx:
         aris_id = 256176

         import time

         t = time.time()
         print(nx.dijkstra_path_length(G, 20405, aris_id))
         print(time.time()-t)

         t = time.time()
         print(dijkstra(G, 20405, aris_id))
         print(time.time()-t)
```

```
6.747649572649573
0.043797969818115234
6.747649572649573
0.025595903396606445
```

- GroupNumber:

```
In [11]: def GroupNumber(graph, subset_of_nodes):

             GroupNumber = {} #define a groupnumber dictionary {node: {sub_node : shortest_pat
             #we take all the nodes of the graph - the nodes of the subset
```

```
    difference_set = list(set(graph.nodes())-set(subset_of_nodes))

    for idx, node in enumerate(difference_set):
        #initially we set a general weight = to infinite and each time we
        #find a lower weight
        #we substitute the previous value
        weight = float('Inf')

        try:
            #calculate the shortest path between node and subnode and take the min va
            all_paths = dijkstra(graph, node)
            GroupNumber[node] = min(all_paths[x] for x in subset_of_nodes)
        except:
            pass

    #we filter the result in order to delete the keys that have no value
    filtered = {}
    for i in GroupNumber:
        if GroupNumber[i] != {}:
            filtered[i] = GroupNumber[i]
    return filtered
```

Example of use this last part.

```
In [12]: #In this cell, instead, we use the method that calculates the Groupnumber.
         #For example we find 21 nodes connected to Aris and we calculate the group number of
         aris_connected = {}
         i=0
         while len(aris_connected)<21:
             try:
                 aris_connected[G.nodes()[i]] = dijkstra(G, G.nodes()[i], aris_id)
             except:
                 pass
             i+=1


         #we can now take those nodes as the set of subnodes for the computation
         #of the groupnumber

         #to know how much time it takes:
         t=time.time()
         result = GroupNumber(G,aris_connected)
         print(time.time()-t)
         print(result)
```

```
53.908344984054565
{114691: 5.17542735042735, 16388: 0.75, 16390: 4.335, 16392: 3.835, 163849: 4.741176470588235,
```

The execution time for the Group Number, processing 21 subnodes of the reduced dataset, is around 56 seconds. Overall this is a good result considering that we implemented a dijkstra algorithm that has the same velocity of the nx library's one.

Considerations about possible improvements of the code In order to reduce the computational cost of the GroupNumber algorithm we can do several things. First of all, because this is an undirected graph, we can compute just one time the shortest path between 2 nodes and then take same value for the reverse path. Second, we can think to use a different version of the dijkstra algorithm that takes in input a list of nodes instead of a single one. This result in a multi-source dijkstra that assumes that all the nodes of the subset are already visited and assigns to them a weight cost of 0. Then for every subset's node we see what nodes of the graph are reachable (that are not visited yet). At the end we follow the path that has the lowest weight and use the single-source dijkstra from that node to the others.