Mitro 209 : Graph and Data partitioning, Project

Ducottet Rémi

31 january 2023

Teacher : Maro Sozio

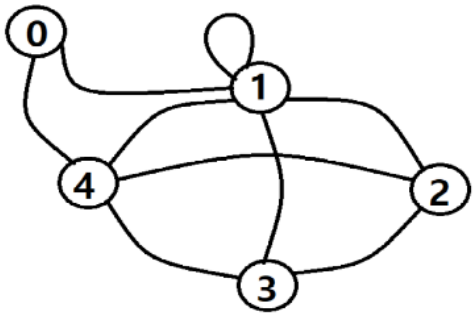
I. Before running the algorithm

Before executing the algorithm, you must download the libraries numpy, sklearn, matplotlib, pathlib, pandas and timeit.

If you want to execute this algorithm on one of your csv files, first you must make sure the name of the columns of your csv file are “id_1” and “id_2”. Your csv file must be placed in the same location as the attached python file. You just have to change the line 59 from filename=”test.csv” to filename=”your_file_name.csv”. You can now run the algorithm and it will compute a densest subgraph in (I hope) a linear time.

Once you run the script, it may take some time if the graph is very big, but it will return at some point a little text in the terminal. This little text will give the initial number of nodes, of edges and the initial density. Then it will tell the time needed to compute all this. Finally it will tell the densest density and the number of nodes in the densest subgraph. This density will be compared to the density of the clique graph with the same number of nodes and the algorithm will compute the clique density. The remaining nodes are available in the script, I did not want to overwhelm the console so I did not print it.

All in all, a graph will be plotted to show that the algorithms had indeed linear time.



II. What does the algorithm do ?

Let assume we want to compute a densest subgraph of this graph (image on the left). This graph would be represented by a csv file like you can see on the image on the right.

To read this csv_file, my algorithm creates at first a temporary_list, which is composed of two lists. temporary_list[0] represents the first node of an edge and temporary_list[1] represents the second node. In our case, we would have temporary_list=[[1,1,2,2,1,3,4,1,0],[2,3,4,3,4,4,0,1,1]]. This object is only the csv file, read by the computer. It allows us to know the number of edges M (that is the size of the first list temporary_list[0]) and the number of nodes N (the maximum of all the nodes with a shift of +1 as there is a node 0).

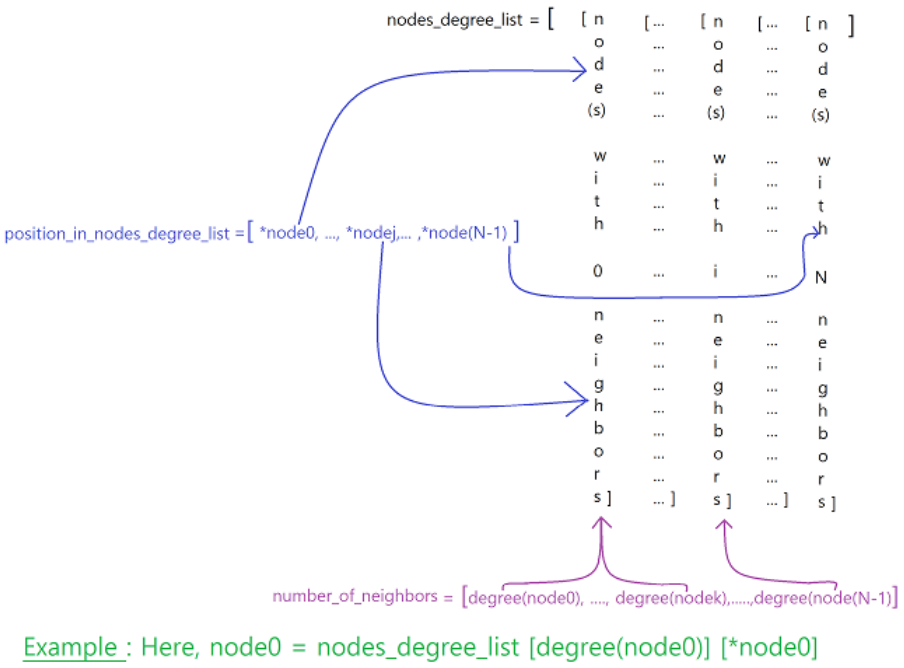
id_1	id_2
1	2
1	3
2	4
2	3
1	4
3	4
4	0
1	1
0	1

From our temporary_list, we want to compute an adjacency list (named adjacencyList) that will represent our graph. Thanks to this representation, we will easily access the neighbors and the degree (number of neighbors) of each node. The adjacencyList is a list of list of nodes where each sublist of nodes represents the neighbors of one node. For instance adjacencyList[0] is a list only composed with the neighbors of the node 0, in our example it is 4 and 1. As a result, the

graph above would be represented by the adjacencyList=[[4,1],[2,3,4,1,0],[1,4,3],[1,2,4],[2,1,3,0]].

In order to respect the complexity wanted, I followed the advice of the slides and created a list of list of nodes sorted by degree. In this list (named nodes_degree_list), the i-th sublist would be a list of all the nodes that have i neighbors. I tried to represent this list and all the lists I will use in the scheme on the right (it is the black list). With the example of the little graph above, here we would have nodes_degree_list = [[], [], [0], [2, 3], [4],[1]]. This list begins at first with two empty lists because every node has at least 2 neighbors Then the 3rd sublist is [0] as zero is the only node with only two neighbors (shift of one as the first sublist represents the list of the deleted nodes)….

Whereas I won't modify the adjacencyList, the nodes_degree_list will change. If I delete a node from my graph, I must suppress it from his sublist and add it to nodes_degree_list[0]. It won't have any neighbors (a node suppressed will be represented by a node without any neighbors). However I can't make too many operations (like search a node in a whole sublist or il all the sublists), otherwise the algorithm would be too complex. This is why I must keep the position of every node somewhere (this is similar to pointers, but as my list isn't linked, I just note somewhere its position of the node in the list.). To help you see more clearly, here is a little scheme to explain all these list.



To begin with I created the list list_degree that tells the degree of any node (number of neighbors). For the Graph G, we would have number_of_neighbors=[2,4,3,3,5]. This means that the node 0 has list_degree[0]=2 neighbors, and is kept in the list nodes_degree_list[number_of_neighbors[0]].

But now we want to know the position in this sublist of the node 0. To do so, I created another list (position_in_nodes_degree_list) that is composed of the position of the nodes in the sublists of node_degree_list. For example, as node_degree_list[list_degree[3]]=[2,3], 3 is at the second position inside this sublist (or more precisely 1st because there is a shift of -1). So position_in_nodes_degree_list[3]=1.

At each iteration, my algorithm wants to delete a node. The value index_of_smallest_node tells the node with the smallest degree, and tells me in which sublist of nodes_degree_list I have to delete a node.

The degree of the node that will be deleted is saved in the value "degree" because this value helps to compute the density of the densest subgraph.

Deleting a node is an easy thing to do linearly (we just have to place it in node_degree_list[0], which means that it does not have any neighbors anymore). But, we

must warn its old neighbors of this deletion. The neighbors are the nodes in the adjacency list that don't have a null degree (a node with a null degree is a node already deleted). As the neighbors just "lost" a neighbor (the deleted node), their degree is diminished by one so we have to place them in their right new sublist of node_degree_list. Let's consider a neighbor of the deleted node. The only way to suppress it from his sublist in O(1) would be to pop it out. Yet it may not be the last element of the list so I must exchange it with the last element and only then I can pop it out. Once it is popped out, I can append it to its right new sublist.

At the end of every iteration, the density of the new graph is computed and compared with the old density. If the new density is better, it is saved in the ariable density.

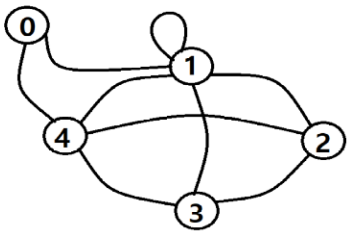
Every deleted node is saved in a list named deleted_nodes. Thanks to this list, once we know the optimal density, the list of the nodes of the densest subgraph is computed in nodes_of_the_densest_subgraph.

All in all, a graph that computes my complexity is plotted.

III. Example of the algorithm with pseudo-code

To illustrate how the algorithm works, Here is the pseudo code written down, followed by the example of 1 iteration of the algorithm on the graph above.This way the evolution of the values in these lists will be clearer.

```
index_of_smallest_node=1
for every npde in the graph :
    while no node has a degree of index_of_smallest_node, this index increases
    delete a node that had this degree and updates his 3 lists position, degree_list and number_of_neighbors
    for every neighbor not deleted yet of this node
        invert it with the element at the end of his sublist of degree_list and update his pointers
        pop it out of this sublist, and place it in the sublist one index smaller
        update his pointers in the lists position and number_of_neighbors
    compute the new_density and density= new_density if new_density>density
    index_of_smallest_node-=1
```



Let's show an example of execution of the algorithm on this graph

Step 1

deleting node 0

index_of_smallest node = 2

nodes_degree_list=[[ ] , [ ] , [0] , [2 , 3] , [4] , [1]]

position_in_nodes_degree_list = [0,0,0,1,0]

number_of_neighbors = [2,5,3,3,4]

position of node 0

Step 2

nodes_degree_list=[[0] , [ ] , [X] , [2 , 3] , [4] , [1]]

position_in_nodes_degree_list = [0,0,0,1,0]

Step 3

neighbors of node 0 = { 1 , 4 }

First let's supress node 1

nodes_degree_list=[[0] , [ ] , [ ] , [2 , 3] , [4 , 1] , [X]]

pop()

position_in_nodes_degree_list = [0,1,0,1,0]

+ append()

number_of_neighbors = [0,4,3,3,4]

Step 4

Let's then supress node 4

nodes_degree_list=[[0] , [ ] , [ ] , [2 , 3] , [4 , 1] , [ ]]

position_in_nodes_degree_list = [0,1,0,1,0]

number_of_neighbors = [0,4,3,3,4]

Step 5

index_of_smallest_node = 3

nodes_degree_list=[[0] , [ ] , [ ] , [2 , 3 , 4] , [1 , 3] , [ ]]

pop()

position_in_nodes_degree_list = [0,0,0,1,2]

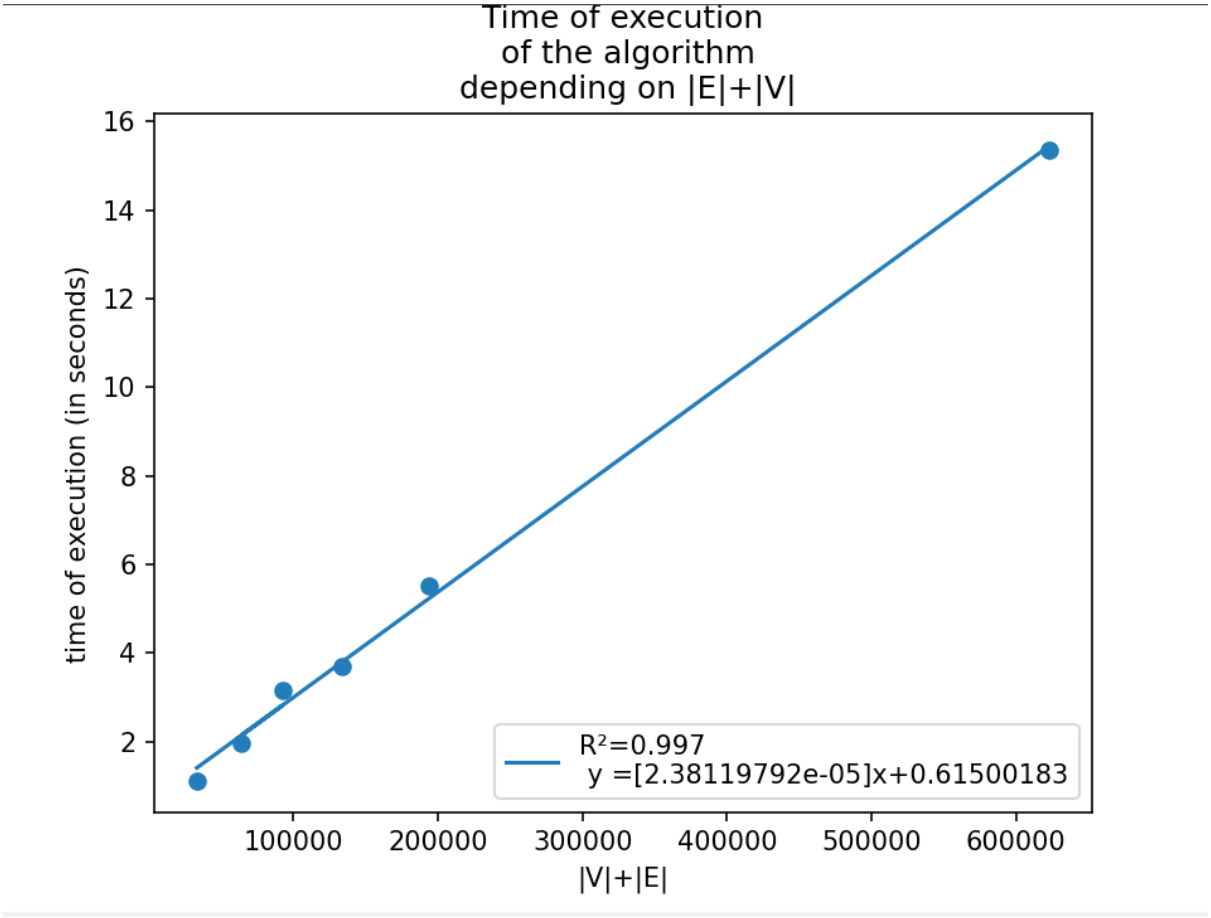
+ append()

IV. Result of the algorithm

Here is the a tab that summarizes the results of my algorithm. For each graph, the algorithm computed the number of nodes in the subgraph, the density of the densest subgraph and it compares it with the clique density. We remark that for most graphs, the clique density of the densest subgraph is quite small (except for the graph facebook combined). So in these big graphs, we succeed to find big densest subgraphs but these subgraphs are not so much connected (as they aren't very dense compared with cliques).

	Initial graph				Densest Subgraph				
Name of graph	Number of nodes |V|	Number of edges |E|	|E| + |V|	Initial density	Number of nodes in the densest subgraph	Density of the densest subgraph	Clique w ith the same number of nodes	Clique density	Average running time (in seconds)
musae_PTBR_edges	1912	31299	33211	16.37	361	31.49	180	0.17	1.12
facebook_combined	4039	88234	92273	21.85	203	76.97	101	0.76	3.17
musae_ES_edges	4648	59382	64030	12.78	574	28.62	286.5	0.1	1.95
musae_FR_edges	6549	112666	119215	17.2	840	34.1	419.5	0.08	3.7
musae_facebook_e	22470	171002	193472	7.61	326	34.9	162.5	0.21	5.53
Email-EuAll	265214	420045	685259	1.58	368	48.38	183.5	0.26	15.36

In the following graph that represents the complexity of my algorithm, we wan see that the complexity seems rather linear given the quality coefficient of my linear regression. These points were found after running 25 times my algorithm on the graphs mentioned abive and after computing the empirical mean running time for every file.



V. Complexity of the algorithm

Let M be the number of edges and N the number of nodes. To begin with, just to read all the data in the csv file, it must be done in O(M) as it reads every edges and creates a list in which it saves these values

To create the adjacency list, first I create a list of n empty lists that represent every node, so O(N). But then, I fill these lists with 2*M values as for every edge, I must add the two nodes in the right sublists. So creating the adjacency list is done in O(N+M).

To create the list nodes_degree_list,that is the list of degrees, I initiate n sublists then I fill them with n values. It is thus an O(N). The lists position_in_nodes_degree_list and number_of_neighbors are just 2 lists of n elements (that are n pointers) so their creation is an O(N).

Let's analyze the mechanism to pick a node. I initialize the int index_of_smallest_node at 1 at first. This int will grow until it finds a minimal degree. At the end of each iteration, this value reduces by one. So during the N-2 iteration of the algorithm (the deletion of the nodes), index_of smallest node reaches in the worst case N at the first iteration (case of a clique), and then slowly decreases until it reaches 1 at the last iteration. Its complexity is a O(N).

During the N-2 iteration, I compute at each step the density, and the new number of nodes or edges in O(1). So all this is an O(N).

I delete N-2 nodes from the graph. Each deletion has consequences on the neighbors of the deleted node. Yet I only pass once over each edge. So the part in which I pop out of a list the neighbors of the deleted node is a O(M), because at most I decrease M-1 (it remains one edge at the end) nodes, and to do so I only use operations that cost me a O(1), hence the pop(). However, I delete N-2 nodes by placing them in nodes_degree_list[0], this costs me an O(N).

When I compute the nodes of my densest subgraph, this is O(1) manipulations on lists of size N, hence an O(N) cost.

All in all, after adding all these complexities I find a O(N+M) complexity. I think that my algorithm is linear.

VI. Critical feedback of my code

This algorithm doesn't notice if all the nodes are present. For example, if I give him a csv file with only one edge that is (0,10), the algorithm will consider that there are 11 nodes and that this graph has a density of 2/11. All the nodes must be present at first, otherwise the algorithm will return some wrong values.

Hence my complexity isn't really a 0(M+N) but an O(M) as I can't have M<N the way I coded my algorithm.