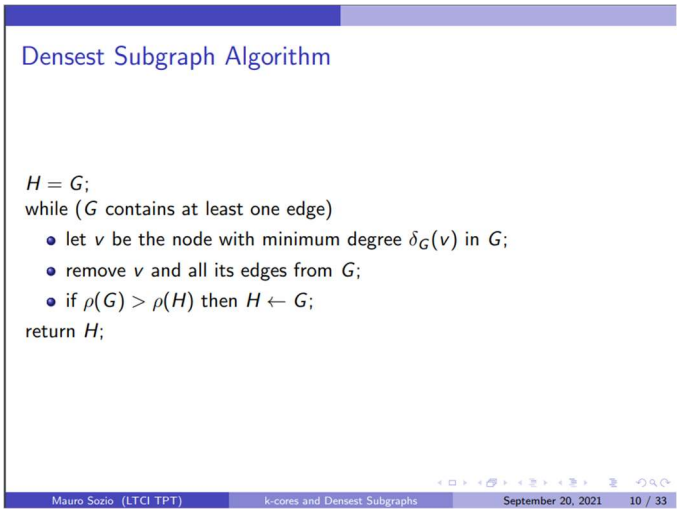
MITRO209 : Project : Report :Quentin RAYNAUD

The objective of this project was to implement the 2-approximation algorithm for the densest subgraph problem, showed in class :



I decided to work with python, as it is the language I master the most. The difficulty was that I had to implement the algorithm in a complexity O(N+M), where N denotes the number of nodes of G, and M the number of edges of G.

I) Overview on the code

The code is split in several parts. Firstly, I imported two modules : timeit and matplotlib. The first module is used to compute the running time of the algorithm, and the second one is used to plots the results to verify that the complexity of the algorithm.
The graphs that I used were implemented as adjacency lists, on .txt files or .csv files.
Then there is the main function : densest_subgraph. It takes a string in entry, that corresponds to a link to the file of the graph we want to study. It returns a list containing N the number of nodes in the graph, including the isolated ones, M which is the number of edges, best_density which is the density of a densest subgraph, and best_iter, which is the number of the iteration where the program found the best subgraph. Indeed, the pseudo-code above shows that the program iterates to remove all nodes from G. To have information about H, we have to note the number of the best iteration.

Then, this is the part of tests. I tested the code on 5 different graphs, as asked. I then computed the running time of the algorithm for each of these graphs.

Finally, I ploted the running time in function of N+M.

II) The main function

The main plot was the densest_subgraph function. I used the open() function of python. The complexity of this function is O(M) : Indeed, the complexity of this function is linear in the number of characters in the file. But the number of characters per line is the same for all lines, and there is exactlty M lines in every file. Moreover, I save in E the number of edges (At this point, we have E = M).
To compute the algorithms, I had to gather data in structures. A list of the nodes ranked by degrees, a list containing the degree of each node, and a list of the neighbours for each node. During the execution of the while loop, I will update these lists as I remove nodes : I will either assign values to adresses in the lists, or remove values. However, the

complexity of deleting values in lists is in O(n), where n is the number of elements in the list. Thus I chose to used dictionnaries. Dictionnaries use hash table internaly for storage and access operations, so all my operations in dictionnaries will be in O(1).

So I implemented the 3 lists as described earlier as dictionnaries.

```
    nodes_by_degree = {i:dict() for i in range(E)} #dictionnary where the key i is a dictionnary of the nodes of degree i. If the degree of a node is i, it is a key
associated to the value 1 in the i-th dictionnary
    degree_of_nodes = {i:{} for i in range(E)} #dictionnary where the key i represents a node, and the value represents the degree of this node
    neighbours = {i:{} for i in range(E)}# dictionnary where the key i represent a node, and the value is a dictionnary of all the neighbours of the node i. If a node is a
neighbour of i, it is a key associated to the value 1 in the i-th dictionnary
```

The complexity of initialising these 3 dictionnaries is O(M), as E = M
I then re-open the file using the open() function. I fill these dictionnaries, reading lines 1 by 1. When the code read a line, for example « 0 2 », it does :

Look if the node 0 was already encountered. If yes, increase its degree by one, add 2 to his neighbours. Else initialize its degree to one, and add 2 to his neighbours. It does the same operation for the node 2.
We also use this loop to update N the total number of nodes, and max_node the number of non-isolated nodes.

All operations in this part of the code are either assignments, either increments, either accessing adresses, which is O(1) because we are using dictionnaries. Thus all the operations in the open() loop are in linear time. Finally, the complexity of preparing these dictionnaries is O(M), because we saw that opening the file was O(M).
We itinialize the minimum degree by iterating over the nodes. In the worst case, all nodes are linked to all nodes (there is O(n^2) nodes), so the complexity of this operation is O(N)
After these operations, there is the main part of the code : the while loop. In this loop, all edges from the graph G are going to be removed. In addition, we are not going to consider isolated nodes : removing them won't change the dictionnaries neither the density of the graph. Therefore, instead of using a while, as we know the number of nodes, which is max_node, we will use a for loop : we will iterate until max_node-1, where we won't have any nodes lefts in the dictionnaries.

At each iteration, the code computes the density of the graph. If the density is better than the previous one, we update it. We then compute the node of minimal degree.
The iter() returns an iterator in the dictionnary keys, in O(1), because an iterator is just a pointer. We will always have an element in the first key, so the next() will only happen once. Thus we will get the node of minimal degree in O(1).
We immediately delete it from the dictionnary of nodes by degree. This operation is in O(1) because we use a dictionnary
We update the values of the nodes and the edges in the graph.
Then, we need to update all the other nodes. We loop on all the nieghbours of the node of minimal degree. All the operations we do in this loop are either deletions, either affectations, either computations. Computations in dictionnaries are in O(1).
Moreover, we will iterate in total through all the nodes, which number is M. As we do nothing for isolated nodes, the total cost of operations is O(N+M)
The other operations in the loop are about updating the minimum degree. The maximum possible degree is N. But, we will either iterate before the loop, so if it is the worst cas we sawi t is a O(N) operation, or update a little at each iteration. So in the total execution

of the code, we will increment the min_deg at most N times, so updating it is O(N). Thus the complexity of updating the mininimum degree goes in addition to the complexity of updating the lists . And O(N) is O(N+M). Similarly, we earlier did operations in O(M), which is also O(N+M). Thus the complexity of the code is O(N+M).

The nodes of the subgraph are not asked. We could have computed it by doing the same operations, but stopping at best_iter, and collecting the nodes that we remove, to know which nodes should be kept.

Here is the graph where N+M is in the axis x, and the duration is in the axis y.

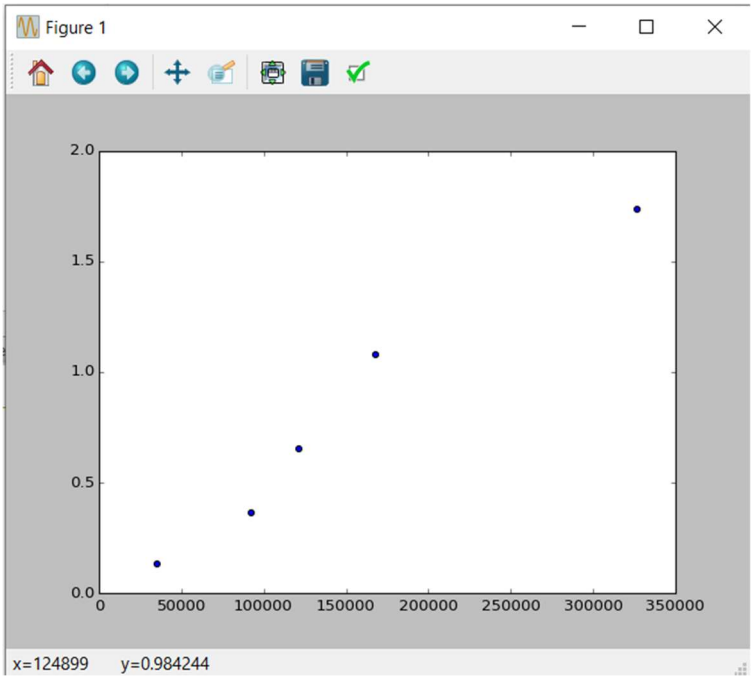


Figure 1 : Time in function of N+M

We see that the points seemes distributed around a straight line, which confirms the linear complexity of our program.

```
[4038, 88234, 77.34653465346534, 3837]
[28279, 92752, 8.286153846153846, 27240]
[7623, 27806, 13.984375, 7376]
[37699, 289003, 29.686567164179106, 36332]
[41772, 125826, 4.8061224489795915, 40542]
```

Figure 2 : Table showing the nodes and edges of the original graph, the density, and the number of nodes in the densest subgraph

III) Other points

The code is not perfect. It does not consider the case where a node is linked to himself. This might cause problems in some graphs. Moreover, the points does not form a perfect straight line because lag might occur in my computer.