Finding densest subgraphs in linear time

Anh-Vu Nguyen
Télécom Paris
anh-vu.nguyen@telecom-paris.fr

1 INTRODUCTION

We know that finding the optimum densest subgraph can be computed in polynomial time. In this exercise, we try to implement an algorithm that computes an approximation of the densest subgraph in linear time of the input. We are going to implement a 2-approximation greedy algorithm.

2 LINEAR TIME IMPLEMENTATION OF A 2-APPROXIMATION ALGORITHM

2.1 The algorithm

Algorithm 1: 2-approximation greedy algorithm

```
Result: H
H = G;
while G contains at least one edge do
    Let v be the node with minimum degree $\delta_G$(v) in G;
    Remove v and all its edges from G;;
    if $\rho(G) > \rho(H)$ then
        H ← G;
    else
end
```

2.2 Linear time implementation

For clarity, we are going to explain the implementation in Python. We will later give an implementation in C++ that runs even faster. The code can be segmented in 3 steps, the initialization which produces all the necessary variables and data for the greedy algorithm, the previously described algorithm, and saving the data.
Note that the python code runs the algorithm on all graphs found on the relative directory graphs/. It saves the results in the folder saved_results.

3 JUSTIFICATION

This part is referencing the code, it is easier to read it with the code in the file 2-approx_alg.py.

3.1 Step 1 - initialization

Before running the Algorithm 1, we need to process the data.

`def read_file(path)`

This step requires reading the txt or csv file containing all the edges in the graph, each edge being on a different line. We store all the edges in a list, each edge being a pair of nodes. By iterating through the lines, we also store the node with the max id assuming that the graph contains all the nodes between 0 and the node with max id. Each step is constant in time thus a linear time complexity $O(|E|)$.

`list_adjacent(edges,max_node)`

This function computes the adjacency list by creating a list of length the number of nodes and by iterating through the edges in constant time at each step. Thus a time complexity in $O(|V|) + O(|E|) = O(|E|)$.

`degree_function(list_adj)`

This function gives a list with the degree of each node. It returns D were D[i] is the degree of node i.It iterates the adjacency list with constant time at each step. Thus a time complexity in $O(|E|)$.

`list_nodes_by_degree(D)`

This function returns L were L[i] is a list of nodes of degree i. It creates a list of length below |V| and iterates through the list of degrees D with each step that runs in constant time. Thus a time complexity in $O(|E|)$.
We also create a list of removed edges and nodes, we also store $|V|$ and $|E|$ that will be updated.
In conclusion the initialization step runs in linear time $O(|E|)$.

3.2 Step 2 - Greedy algorithm

This part is referencing the code, it is easier to read it with the code.

•*while G contains at least one edge do :*
The stopping criterion can be replaced by checking if all the nodes have been removed.

```
while number_nodes != len(removed) :
```

This can be done in constant time by checking if the number of removed nodes is equal to the initial number of nodes $|E_G|$.

•*Let v be the node with minimum degree in G*
We then remove a node with minimum degree by updating the necessary variables in constant time. For this we pop a node with minimal degree in the list of nodes by degree : L_degrees. Remark that later in the code, we add new nodes to L_degrees. Thus the while loop and the necessity to check if it has been removed already with binary_removed[node] (binary_removed[i] = 1 if node i has been removed).

```
while True:
   node = L_degrees[d_min].pop()#remove node with min degree
   while  d_min<d_max and not L_degrees[d_min] : #update min degree
         d_min+=1
    if binary_removed[node]==0:#if 1, already removed !
         break

removed.append(node) #add node to removed nodes
binary_removed[node]=1   #add node to removed nodes
```

Justification : we only add a node to L_degrees when we remove an edge. So the total number of added nodes at the end of the greedy algorithm is in $O(|E|)$. So the above while loop still maintains a linear complexity.

•*Let v be the node with minimum degree in G*

Then we update the degree of all the neighbors of the selected node. We update the list of nodes by degrees by adding the neighbors to the lists of their degrees-1 which is their new degree. Note that they will then appear multiple times, but because we only consider nodes with min degree is does not change the result.
We also update the min degree and add the edges containing the removed node to the list of removed edges.

```
k=0 #count the number of edges removed to update the density
for neighbour in L_adjacent[node]:
        if not binary_removed[neighbour] :
            k+=1.0
         L_degrees[deg[neighbour]-1].append(neighbour)
            if deg[neighbour]== d_min : d_min-=1
            deg[neighbour]-=1
            e_removed+=[[neighbour,node]]
```

Justification : Everything inside the for loop runs in constant time. Since at each iteration of the greedy algorithm, a different node is selected and because the adjacency list has a number of nodes in $O(|E|)$, this for loop maintains linear time.
•*If the density of G is greater than H, H becomes G*
We then update the number of nodes, edges and the density. The final result is all the removed edges (= all the initial edges because we remove everything at the end) minus the removed edges at the last step where H is updated.

```
#update E and V
    V-=1.0
    E-=k
    if V!=0 and float(E/V) > d  :

        #at the end, e_removed[m:] = H
        m=len(e_removed)
        d=E/V
```

The key trick here is that we do not need to save the updated H when a greater density is found. We only need to know which edges have been removed, which is the same as remembering the length of e_removed.This final part is also in constant time.
Conclusion : The greedy algorithm runs in linear time $O(|E|)$.

3.3 Step 3 - Saving the graph

```
save(graph,path)
```

We save the graph by writing the returned graph of the greedy algorithm in a .txt file (one edge per line). It runs in linear time.

3.4 Conclusion

Each step runs in linear time, thus a time complexity in $O(|E|)$.

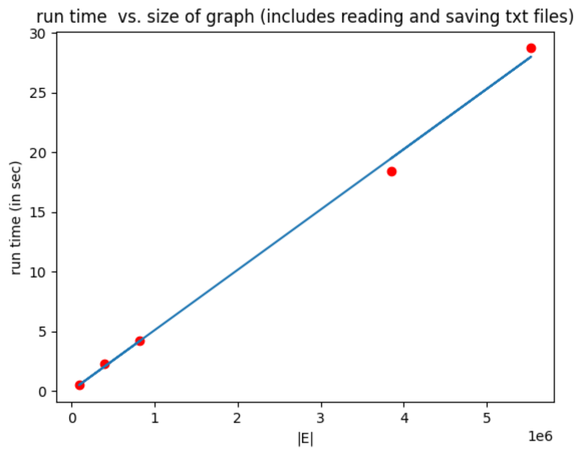

Figure 1: Run Time with initialization and saving

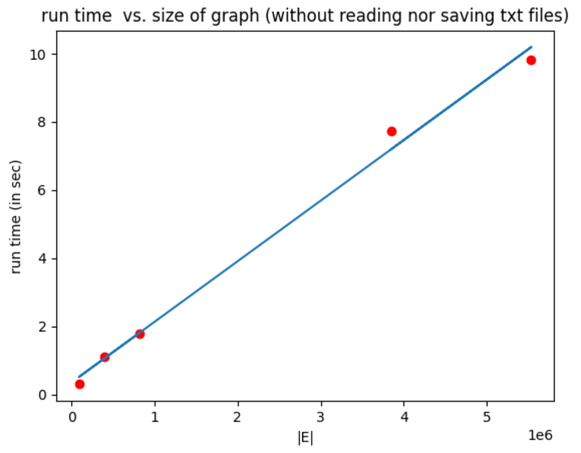

Figure 2: Run Time without initialization or saving

4 PERFORMANCE NUMBERS OF THE PYTHON CODE

The 5 chosen graphs range from 88234 to 5533214 edges they are in **table 1**. As expected, the run time performance seems to be linear of the number of edges as shown in **figure 1** and **figure 2**

Table 1: Graphs

edges	graph
819090	artist_edges.csv
396100	CA-AstroPh.txt
88234	ego-Facebook.txt
5533214	roadNet-CA.txt
3843320	roadNet-TX.txt

Table 2: Performance with initialization and saving steps

run time (sec)	graph
4.231691360473633	artist_edges.csv
2.25187635421752	CA-AstroPh.txt
0.5404579639434814	ego-Facebook.txt
28.74393320083618	roadNet-CA.txt
18.406732797622	roadNet-TX.txt

5 RESULTS NUMBERS

The number of edges and density of the returned graphs are listed in **table 3**.

Table 3: Density and number of edges of the results

graph	density	number of edges
artist_edges.csv	58.13167053364269	100219
CA-AstroPh.txt	59.102918586789556	76952
ego-Facebook.txt	77.34653465346534	15624
roadNet-CA.txt	3.3485469780709574	18782
roadNet-TX.txt	3.4909090909090907	576

6 C++ VERSION

The main advantage is that we can tightly manage our memory by using pointers and by freeing the memory whenever a variable or list if not required anymore. This is particularly important for massive datasets where memory constrains can limit the amount for processed data.
Because the code is not interpreted but compiled, it also runs faster. Here the python code as been transcribed in C++. We use Vectors to replace the lists and delete them with delete[].
Note that the C++ version does not save the graph and that it requires **C++17** and has only been tested on windows.

It gives the same results as the Python code but is much faster ! (figure 3)
Output of the c++ code :

```
graphs\artist_edges.csv
duration (sec) : 0.028027
n and  m :50515   819090
RESULT : density = 58.1317 edges =  100219
======================
graphs\CA-AstroPh.txt
duration (sec) : 0.0127
n and  m :133280   396100
RESULT : density = 59.1029 edges =  76952
======================
graphs\ego-Facebook.txt
duration (sec) : 0.002734
n and  m :4039   88234
RESULT : density = 77.3465 edges =  15624
======================
graphs\roadNet-CA.txt
duration (sec) : 0.251625
n and  m :1971281   5533214
RESULT : density = 3.34855 edges =  18782
======================
graphs\roadNet-TX.txt
duration (sec) : 0.177687
n and  m :1393383   3843320
RESULT : density = 3.49091 edges =  576
======================
```

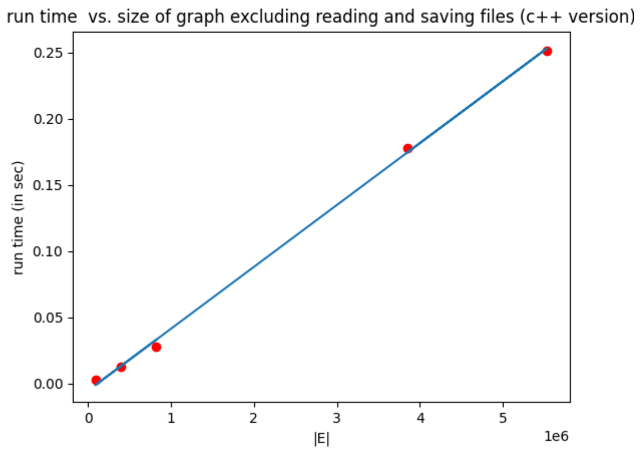

Figure 3: Run Time without initialization or saving

Table 4: Performance without initialization or saving (C++ version)

run time (sec)	graph
0.028027	artist_edges.csv
0.0127	CA-AstroPh.txt
0.002734	ego-Facebook.txt
0.251625	roadNet-CA.txt
0.177687	roadNet-TX.txt

The greddy algorithm part run 4 times quicker on the c++ version. The initialization and saving part are bounded by io throughput (sata SSD vs nvme ssd vs hard drives).