MITRO 209: Project

Aurélien Castre - acastre@enst.fr

January 16th 2022

1 Context and algorithm

In this project we propose a linear-time implementation of a greedy algorithm used to compute an approximation to the densest subgraph problem.

Given a graph $G = (E_G, V_G)$ we define its average degree density to be $\rho(G) := \frac{|E_G|}{|V_G|}$. We then try to find a subgraph H of G with maximum average degree density using the following algorithm:

Algorithm 1 Greedy Densest Subgraph

```
H ← G
while G contains at least one edge do
    v ← minimal-degree node in G
    remove v and all its edges from G
    if ρ(G) > ρ(H) then
        H ← G
    end if
end while
return H
```

Note that we could add a condition to make the algorithm run a bit faster (though without changing its computational complexity). In fact, we can stop whenever we reach an upper bound of the average degree density $\left(\rho(H) \leq \frac{|V_H|-1}{2}\right)$ and return the maximum between the current maximum and the upper bound. The advantages of this addition will be discussed in section 5.

2 Implementation

We propose an implementation of the algorithm in Python 3.8.10 using the NumPy 1.21.2 library. In the following, the names of variables in the Python file will be written using `this font`. Let also N and M respectively be the number of nodes and edges of a given graph G.

First, we convert the file containing all the edges of the graph into an adjacency list called `adj`. We chose this representation of the graph because it allows very simple access to the neighbors of a given node. The `initGraph` function saves the NumPy array of Python lists

`adj` because reading a huge text file in Python takes a lot of time and memory. Note that this function also saves another NumPy array called `degrees`.

Updating `adj` every time we remove a node in the graph would be too computationally expensive so we use a combination of arrays and pointers that we update at every step of the algorithm to retain all the information we need. Here is an exhaustive list of all the data structures we will use:

- `degrees`: an int NumPy array of length N indexed by the nodes and containing their degree,
- `erased`: a bool NumPy array of length N indexed by the nodes and containing 0 if the node hasn't yet been erased, 1 otherwise,
- `vertices`: a Python list NumPy array of length N indexed by all possible degrees and containing the list of all nodes of each degree,
- `locations`: an int NumPy array of length N indexed by the nodes and containing the index of each node `i` in the list `vertices[degrees[i]]`,
- `mind`: an int pointer in $\{0, \dots, N-1\}$ pointing to the first non-empty list of `vertices`.

The implementation is divided in two steps: 1. updating the data structures after the deletion of a minimal-degree node and 2. updating the pointer `mind` and computing the new density.

1 After the deletion of a node of minimal degree, we save its index in `imin`. We can then go through the list of its neighbors using `adj[imin]`. For each neighbor which hasn't yet been erased we then update its degree and move it to the appropriate list in `vertices`.

2 Finding the new first non-empty list in `vertices` is then easy. If we encountered a node of degree equal to `mind` during step 1 we need to go back one step for we know that `vertices[mind-]` won't be empty. Else, we just look for the first non-empty list in the array, starting from `mind`. This works because at each step we know that the degree drop of each node is at most 1 since we delete only one node.

3 Computational complexity

In this section we will show that the implementation detailed in section 2 is in $\mathcal{O}(N + M)$.

First, we argue that all operations in step 1 are performed in constant time. The sensitive step is to move a node `j` in the correct list of `vertices` after deletion of another node. For that, we need to:

- access its degree, this is done by `degrees[j]`,
- access its location in the list, this is done by `locations[j]`,
- swap it with the last item of the list `vertices[degrees[j]]` so we can retrieve it in constant time.

As the Python functions `list.pop()` and `len()` run in constant time, we deduce that the step 1 executed N times runs in $\mathcal{O}(M)$.

Second, even if it's tempting to think that executing step 2 N times will never be better than $\mathcal{O}(N^2)$, we show that it is actually in $\mathcal{O}(N)$. This is simply because the pointer `mind` can only

go back one step at each iteration. This implies that the maximum number of steps it will take will never be greater than $2N$, which is enough to conclude.

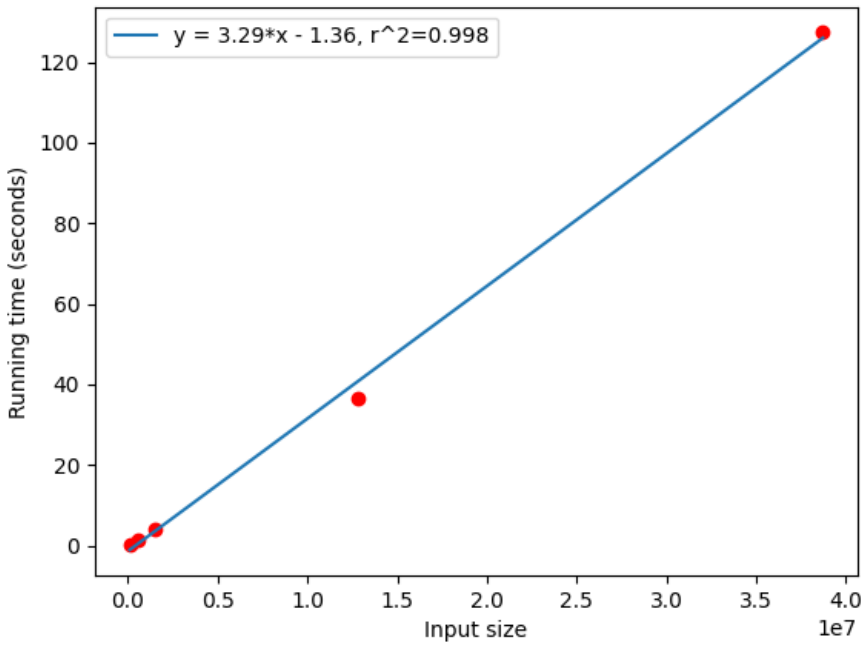
In the end, adding both step 1 and step 2 we indeed get a computational complexity of $\mathcal{O}(N+M)$.

4 Results

In this section we present the data and the results used to demonstrate the conclusions of section 3. All the graphs are from `http://snap.stanford.edu/data/index.html`, here are the details we will need:

	Source	Number of nodes	Number of edges
Graph 1	Social circles: Facebook	4,039	88,234
Graph 2	Deezer	54,573	498,202
Graph 3	DBLP collaboration network	425,957	1,049,866
Graph 4	Autonomous systems by Skitter	1,696,415	11,095,298
Graph 5	LiveJournal social network	4,036,538	34,681,189

In the following plot, we plotted the running time of the algorithm as a function of the size of the input graph, that is $N + M$. We then used some simple NumPy code to perform a linear regression and printed the Pearson correlation coefficient.



Here are the results of the computations performed for each graph:

	Density	Number of nodes	Running time (seconds)
Graph 1	77.3	202	0.209
Graph 2	16.2	7353	1.42
Graph 3	56.5	114	4.09
Graph 4	89.2	431	36.6
Graph 5	191	386	127

5 Conclusion

The scatter plot of section 4 clearly demonstrates the linearity of the implementation, which is reassuring.

However, it doesn't seem like the addition of section 1 would be really useful. In fact, the subgraphs usually have very few nodes at the end of the algorithm, as section 4's results show.