MITRO209 2-approximation algorithm

Mohamed Taha Bayoumi

January 2023

1 Introduction

The aim of this project is to implement the 2-approximation algorithm, seen during the class courses so that it runs in linear time complexity to a given graph's dimensions.

1.1 Densest subgraph problem and 2-approximation algorithm

Before delving deep into the algorithm implementation, as well as the description of time efficiency. We provide some essential definitions to understanding the problem, as well as the 2-approximation algorithm.

1.1.1 Definition

Formally, given an undirected graph $G = (V_G, E_G)$, we define:

- *The density of the graph G*: $\rho(G) = \frac{|E_G|}{|V_G|}$;
- Given a node $v \in V_G$, we denote $\delta_G(v)$ as the degree of the node v;

The objective of our algorithm is to find an approximation to the densest subgraph that we can induce out of the graph G. To do so, we proceed as the *2-approximation* algorithm suggests: (following page)

It is worth noting that the implementation of the algorithm, in a relatively straightforward fashion, could be easily done in $\mathcal{O}(n^2)$, where $n = |E_G|$, or even in $\mathcal{O}(n.log(n))$. However, this project will provide an implementation in a linear time complexity, *i.e*, in $\mathcal{O}(|E) + |V|)$.

2 Implementation choices

2.1 Programming language

The choice of which programming language to use is extremely dependent on the data structures it provides, since implementing the algorithm in linear complexity reveals the importance of choosing adequate data structures, all while

Algorithm 1 2-approximation algorithm

```
INPUT: an undirected graph $G = (V_G, E_G)$
Let: $H = G$
while $G$ contains at least one edge: do
  Let $v$ be the node with minimum degree in $G$;
  Remove $v$ and its edges from $G$;
  if $\rho(G) > \rho(H)$ then
    $H \longleftarrow G$
  end if
end while
return H
```

preferring faster running times. Hence, the code will be exclusively in `C++`, and there will be some code in `Python` to plot the running times.

2.2 Graph representation

The graph representation will be done in `.csv` format, where the two columns' structure represent a set of edges in the graph.

3 Algorithm

3.1 Difficulties

Before describing the algorithm's implementation, we first start by stating the difficulties when trying to maintain a linear complexity:

- **Graph representation**: poses some problems, especially for information retrieval, this is why we went, along some other structures to help retrieve information and update the graph more efficiently, with a variant similar to adjacency list (adjacency hashmap)
- **Deleting nodes**: this procedure would cost a linear time complexity in the length of the considered container. Hence, instead of deleting nodes, we went through a boolean hashmap where we flag every node that is no longer in the graph, and since accessing a value in a hashmap is done in constant complexity, the problem is solved.
- **Accessing a node's degree**: despite the graph implementation, computing nodes' degrees at every iteration of the algorithm is very costly. This is why we opted for a hashmap where the keys are the nodes, the values are their degrees, so that access to a node's degree would be in constant time, and removing the edges of a certain node would be in $\mathcal{O}(\delta_G(v))$.
- **Graph affectation**: a naive affectation of G to H would add up to a quadratic complexity, this is why we opted for changing the parameters of H and G simultaneously when $\rho(G) > \rho(H)$.

3.2 Code description

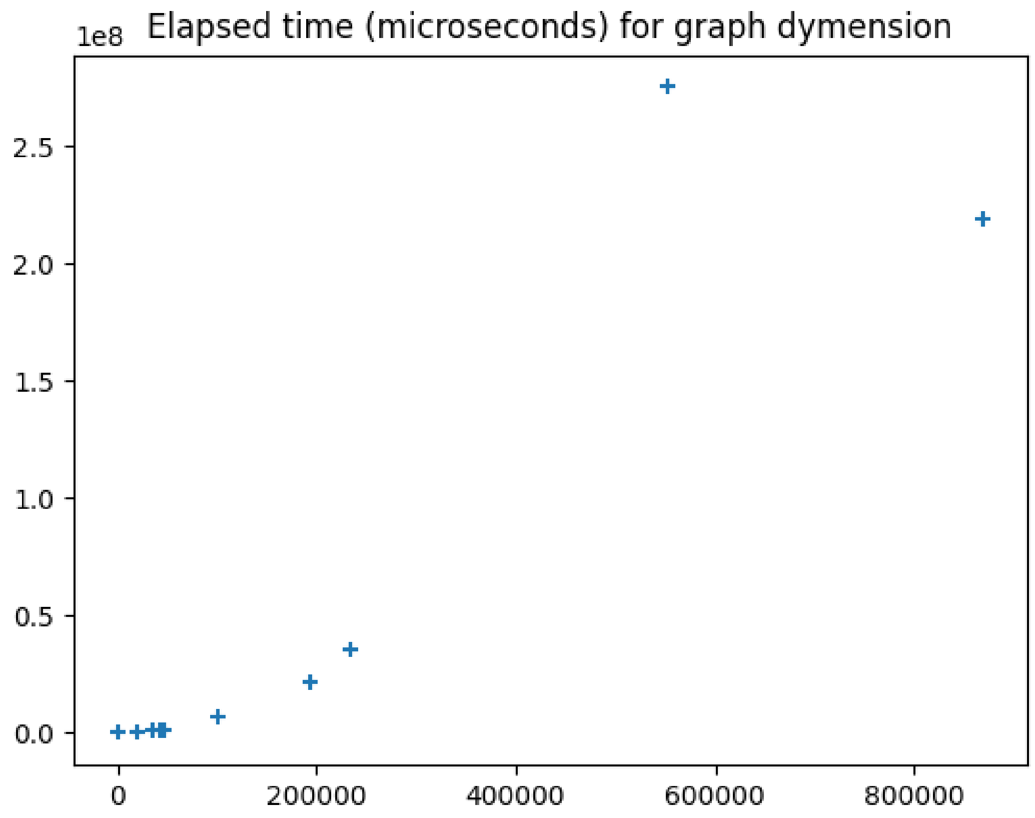
In this implementation, graphs will be implemented through the class `Graph`, which contains:

- `degreeVector`: an unordered map`<key = degree, value = vector of nodes >` that makes accessing the node with the minimum degree more efficient;
- `mini`: an integer that refers to the minimum degree possible in a graph;
- `degree`: an unordered map`<key = node, value = degree >` that makes accessing a node's degree constant;
- `neighbors`: adjacency hashmap;
- `inducedGraph`: an unordered map`<key = node, value = boolean: true if it is in the graph, false otherwise >` that serves for flagging deleted nodes;

We present a brief description of what the code does:

- First we check if the graph still has at least one edge, to do so, `mini` is always updated so that it lands on the minimum degree value, and starts increasing as the algorithm runs;
- If `mini` corresponds to a value where `degreeVector[mini]` has at least one node that has not been flagged yet in `inducedGraph`:
  - The node `w` in question (not flagged yet), is now flagged;
  - We subtract 1 from `degree[v]` where `v in neighbors[w]` ;
  - Compare density values, if it is greater than before deleting `w`: the previous changes are made to G and H simultnaeously, otherwise, they are made only to G;
  - Neighbouring nodes to `w` are being affected to their new corresponding degrees in `degreeVector`, and `mini` takes the minimum value of these updated nodes' degrees, as well as its value, in case a degree becomes of a degree lesser than `mini`;
- While updating G, there are two hashsets: `waitListElimination` and `waitListDegree` that account for nodes to be flagged as eliminated (resp. have an updated degree) between two consecutive affectations of G to H. After each affectation, these hashsets are cleared and built anew until another affectation takes place.

This implementation ends up scanning all nodes of the graph, and neighbors of every node, all while performing constant time operations (since all containers are hashed). Hence, the complexity is in $\mathcal{O}\left(|V_G| + \sum_{v\in V_G} \delta_G(v)\right) = \mathcal{O}\left(|V_G| + |E_G|\right)$. Below is the plot of elapsed time for 10 graphs:


Elapsed time (microseconds) for graph dymension
1e8
2.5
2.0
1.5
1.0
0.5
0.0
0
200000
400000
600000
800000