# MITRO209 Project Report

Thiago F Cesar

January 10, 2020

## 1 Introduction

In this document, we detail the implementation of two approximate algorithms for the densest subgraph problem. We derive their complexity and also present simulation results.

## 2 2-Approximation Greedy Algorithm

### 2.1 Data structures and algorithm implementation

In this subsection we detail the data structures used to implement the algorithm and how they are used to obtain a linear time complexity.

#### 2.1.1 Data structures

Figures 2.1 and 2.2 show the data structures used to implement the algorithm.

The graph data structure is the first structure to be built. Each node has its adjacency list which contains its neighbors. It's easy to see that its creation has both $O(n + m)$ time and space complexity: the allocation of the adjacency list is $O(n)$ both in time and space, while the inclusion of all the edges has total time and space complexity of $O(m)$. This structure is immutable, meaning that is not changed during the execution of the algorithm. At the end of the program, however, we need to delete it, which takes $O(n + m)$ operations.

The degree data structure is built using the graph data structure. We start allocating two arrays of size $n$: the list array and the pointer's array. In order to add each node to this structure, we calculate its degree and add the node's cell to the appropriate list. We also save the cell's address in the pointer array (i.e. in position $j$ of the pointer's array, we find the address of the cell corresponding to node $j$) and increment the value of $n$. As we have to go through the adjacency lists of each of the nodes when building this data structure, the total time complexity is $O(n+m)$. The space complexity however is just $O(n)$.

The degree data structure is a mutable data structure, due to the fact that at each step of the algorithm we will need to remove a node from it. We'll assume firstly that the `start` variable will be always kept updated, meaning that it will always contain the smallest integer $i$ such that `list[i]` is not empty. We'll later prove that it's possible to keep it updated in linear time.

#### 2.1.2 Algorithm implementation

The main idea of the algorithm is to use the degree data structure to get the node with smallest degree at each step and remove if from the data structure. In order for the structure to represent a valid graph, we'll also need to update its neighbors, as their degree will have been decreased by one. Because we remove a node in each one of the steps, we can easily see that the algorithm takes $n$ steps to finish. We'll also need a way of remembering the state of the graph at each one of the steps, something that will be detailed in a following subsubsection.

Because `start` is kept updated, it's possible to remove a node with smallest degree in constant time. Suppose we remove node $i$ from the structure. The value of $n$ must then be decremented by one and `pointers[i]` is changed to null. We also need to lower the degrees of each one of the neighbors of the removed node. Consider the node $j$, which was a neighbor of $i$. We claim that we can decrease its degree by one in constant time. Indeed, as we maintain

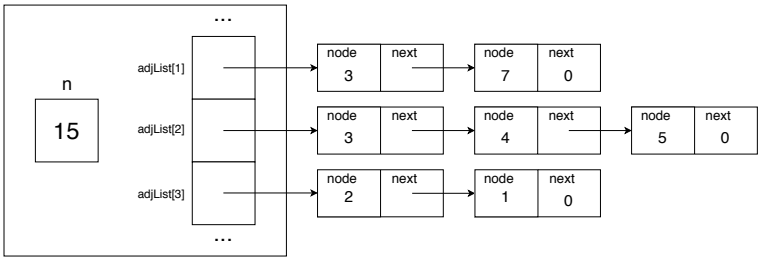


Figure 2.1: Graph data structure.

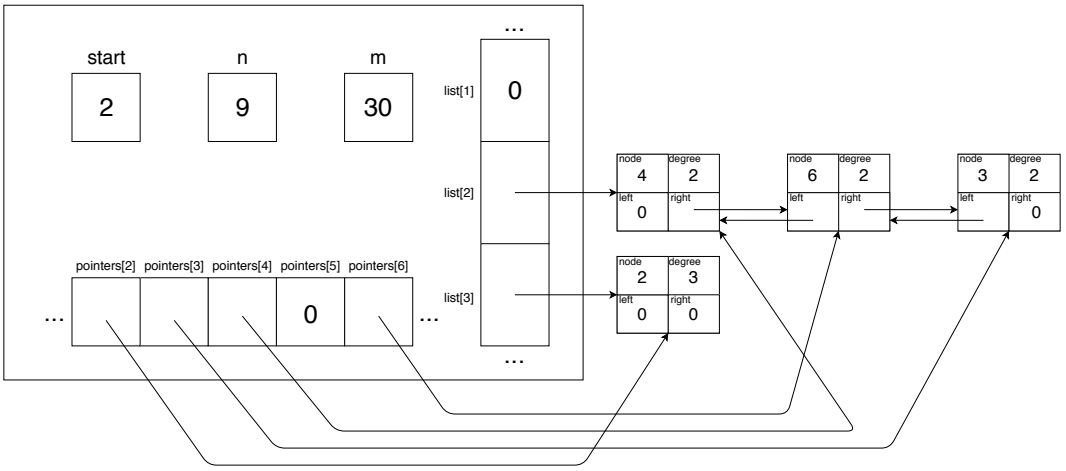


Figure 2.2: Degree data structure.

a list of pointers to the cells in the degree data structure, we can get to each cell in constant time. If the pointer is null, then the node has already been removed of the structure and we don't need to do anything. If that's not the case, then we can get to the cell and, using the pointers `left` and `right`, rearrange the list in order to remove $j$, with a total constant cost (note that this is only possible because we're using a doubly linked list). Because each of the cells in the list stores its own degree, we also know $j$'s old degree (let's suppose that it had a degree of $l$ when $i$ was in the graph). Using this data, we know that we need to add $j$ to the list `list[l-1]`, which can be done in constant time. Finally, we decrease $m$ by the degree of the removed node (node $i$ in the example).

As this operation needs to be performed to each one of the nodes in $i$'s adjacency list, the total cost of updating $i$'s neighbors is $O(\delta(v_i))$. Thus, the cost associated with each one of the steps of the algorithm is $O(1 + \delta(v_i))$. Because a step is executed for each node in the graph, we can conclude that the total cost is $O(n + m)$.

Now we prove that we can keep `start` updated in linear time. Each time we decrease a node's degree we also check if the new degree is lower then `start`, and in this case we update its value (this is a constant time operation). It's clear that at each step of the algorithm, `start` can only be decremented by one (it's impossible for a node that stays in the graph to have more then two edges removed in the same step). Therefore, the number of times we decrease `start` is bounded by $n$. We claim that the total cost of the following operations is linear: before each one of the steps we check whether `start` points to a empty list, and if that's the case we increment it until we find a

non empty list. Indeed, because `start` can never be bigger then $n$, and because the number of times we decrease `start` is bounded by $n$, then the number of times we increase `start` is bounded by $2n$, which is $O(n)$.

When the algorithm finishes, all the cells in the degree data structure will have been deleted, and therefore in order to delete this structure we only need to free the vectors, a $O(n)$ operation.

#### 2.1.3 Remembering the steps and printing the final solution

A naive way of remembering all steps at the end of the algorithm would be to make a copy of the graph at each step. One way to get around this is as it follows. We'll make two arrays of size $n$: one will be called `removalList` and will store which node was remove at step $j$, and the other will be called `densitiesList` and wil store the density after step $j$ (note that, because we keep the values of $n$ and $m$ updated on the degree data structure, we can calculate the density in $O(1)$). Updating each of these arrays with the information related to the current step has constant time complexity.

To find the step in which we found the graph with the highest density we only have to go through the densities array and find the index that maximizes the density, which we'll call $k$. Then we'll make a third array of size $n$ called `nodeIsInSolution` and initialize all the positions to one. Next, we'll go thought the `removalList` and, for each node $j$ that was remove before or during step $k$, we'll make `nodeIsInSolution[j]` equal to zero.

In order to print the solution on the output file, we go through the graph data structure and print the edges whose both ends are on the final solution (this is constant to check, using the `nodeIsInSolution` array). However, there is a small problem: each edge appears two times on the graph data structure. If $\{i, j\}$ is an edge, then $i$ appears on the adjacency list of $j$ and $j$ also appears on the adjacency list of $i$. Because we don't want an edge to appear two times on the output file, we can't print both edges. One workaround is to print only the edges $\{i, j\}$ in which $i > j$ (or $i < j$ alternatively).

It is clear that all of the operations made to print the final solution are linear in $n$ or $m$.

### 2.2 Simulation results

If the time complexity is indeed $O(n + m)$, then we should expect the number of operations to be a function on the form

$$f(m, n) = a \cdot n + b \cdot m + c \ .$$

If we assume the graphs to be connected, then we know that $m \geq n - 1$. In this case

$$f(m, n) \leq a \cdot (m + 1) + b \cdot m + c \ ,$$

which implicates that the time complexity is $O(m)$. We therefore can confirm the fact that the time complexity is linear by making a plot of the runtime by the number of edges.

In order to test our program, we ran it on the datasets ca-AstroPh, com-amazon, com-dblp, douban, flixster, hyves, loc-brightkite_edges, loc-gowalla_edges. We adapted the files in order to provide the graph's $n$ and $m$ on the first line. For each one of the files, we ran the code ten times and we report on Figure 2.3 the average runtime of each one of them. A linear fit shows that the algorithm runs indeed in linear time.

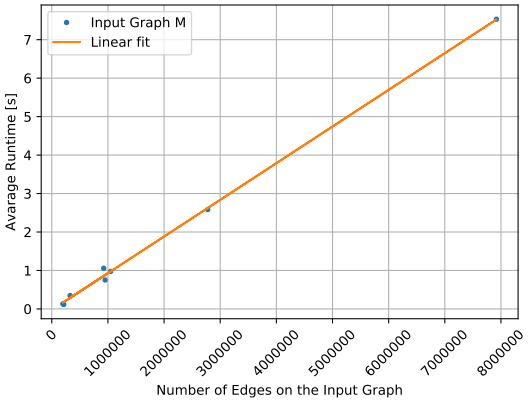


Figure 2.3: Empirical time complexity.

| | ca-AstroPh | com-amazon | com-dblp | douban | flixster | hyves | loc-brightkite_edges | loc-gowalla_edges |
|---|---|---|---|---|---|---|---|---|
| $n$ | 1302 | 24990 | 114 | 1985 | 388 | 670 | 219 | 559 |
| $\delta$ | 29.55 | 3.83 | 56.5 | 13.60 | 50.37 | 33.48 | 40.56 | 43.80 |

Table 1: Number of vertices ($n$) and density ($\delta$) of the subgraphs found by the greedy algorithm.

# 3 (2+$\epsilon$)-Approximation Streaming Algorithm

## 3.1 Data structures and algorithm implementation

### 3.1.1 Data structures

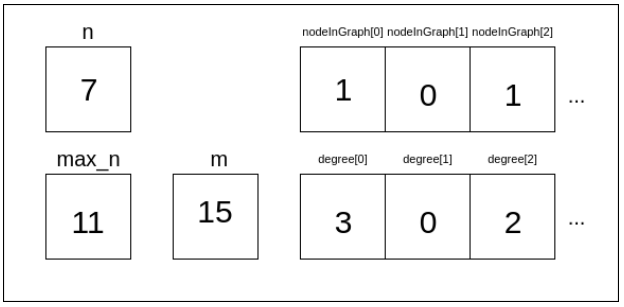


Figure 3.1: Data structure.

Figure 3.1 illustrates the data structure used to store the graph for this algorithm. Because this is a streaming algorithm, we can't store the whole graph, only the degree of each one of the vertices. Since at each step we create a new graph with some of the vertices removed, in the moment the new graph is created we also give it a boolean vector that indicates which vertices are part of the graph.

Thus, when the graph file is read, the function which adds the edges checks whether both ends of the edge are in the graph before counting it (in other words, for the edge $e = \{i, j\}$ it checks if both `nodeInGraph[i]` and `nodeInGraph[j]` are 1). If the edge is indeed in the graph, then the values of `degree[i]`, `degree[j]` and `m` are incremented by one.

The variable `max_n` indicates the size of the original graph. That is, as vertices are removed, the value of `n` gets smaller, but the value of `max_n` stays constant. It is important to preserve this value because it is the size of the vectors `nodeInGraph` and `degree`.

### 3.1.2 Algorithm implementation

We start the algorithm creating a graph whose `nodeInGraph` vector is initialized with all ones. This way, the first graph read is exactly equal to the one on the file.

At each step, we start by reading the graph file. We then use the values of `n` and `m` stored on the data structure to calculate the graph's density. Next, we use a function which creates a `nodeInNewGraph` array and calculates a `new_n` integer from a graph and a bound value. It works by creating a array of size `max_n` with all zeros and then going through the `degree` vector and, for each vertex $i$ whose degree is greater then the bound value, it changes the value of `nodeInNewGraph[i]` to one and increments `new_n`. At each iteration, we use $(2 + \epsilon) \cdot \delta$, in which $\delta$ represents the graph's density at the current iteration, as the bound value. The `nodeInNewGraph` is then used on the following iteration to decide which nodes stay on the graph. The algorithm stops when the value of `new_n` is zero.

As the iterations go by, the graphs that are produced are stored in a stack. This allows us to easily recover the graph whose density was the largest. Next, we get the `nodeInGraph` vector of the graph found and go through the graph file one last time. For each edge, we verify whether both of its ends have boolean value 1 on the `nodeInGraph` array, and if that's the case we print the edge on the output file.

We have seen in class that the algorithm takes $\lceil \log_{1+\epsilon} n \rceil$ iterations to finish. As the operations done in the initialization and finalization phases and inside the iteration's loop all have linear complexity with respect to $n$ or $m$, we conclude that the total time complexity is $O((n+m) \log n)$. The space complexity, however, is just $O(n \log n)$, because at each iteration we create a new data structure whose size is $O(n)$.

### 3.2 Simulation results

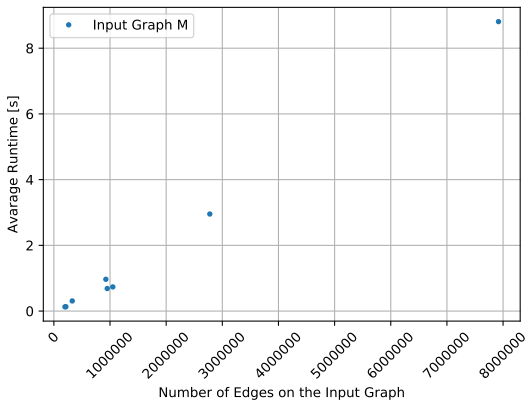


Figure 3.2: Empirical time complexity.

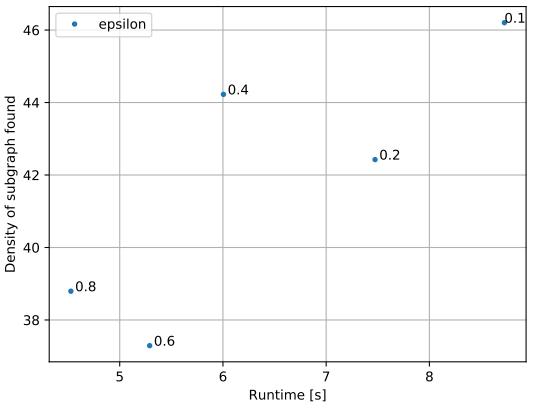


Figure 3.3: Impact of $\epsilon$.

Firstly, we ran the algorithm with $\epsilon = 0.1$ on the datasets ca-AstroPh, com-amazon, com-dblp, douban, flixster, hyves, loc-brightkite_edges, loc-gowalla_edges. We adapted the files in order to provide the graph's $n$ and $m$ on the first line. For each one of the files, we ran the code ten times and we report on Figure 3.2 the average runtime of each one of them. Using a similar argument to the one used in subsection 2.2, we can simplify the time complexity of $O((n+m)\log n)$ to $O(m \log m)$, which allows us to make a 2D plot.

It is true that the points seem to trace a line, which would give us a better time complexity then the one calculated before. However, one can verify using a graphing tool that the graph of $x \log x$ looks pretty linear when looking at a large interval. Though that the data given by this graph is inconclusive regarding the complexity function, it nevertheless allows us to check that the algorithm runs relatively efficiently and that the run times are feasible.

Next, we used the flixster dataset to observe how the behavior of the algorithm changes with $\epsilon$. Figure 3.3 shows the densities and running times for $\epsilon \in \{0.1, 0.2, 0.4, 0.6, 0.8\}$. As expected, the running time increases when $\epsilon$ decreases, which is due to the fact that the number of iterations is $\lceil \log_{1+\epsilon} n \rceil$, a number that increases when $\epsilon$ decreases.

However, the fact that the density doesn't strictly decrease with respect to $\epsilon$ is a bit counterintuitive. To make sense of this result, we need to remind ourselves that we never claimed that the density will always increase with $\epsilon$. We only saw (in class) that the density found will be contained inside the interval $[\frac{\rho(O)}{2(1+\epsilon)}, \rho(O)]$. Therefore, though when $\epsilon$ gets smaller it becomes more likely to obtain a larger density, there is no reason for this to always be the case. Fortunately, we at least verify that there is a general tendency for the density to increase as the value of $\epsilon$ gets smaller, which is something that we expected.

## 4 Conclusion

On this document we have successfully explained the implementation of the greedy and the streaming algorithms for the densest subgraph problem. Simulation results were provided as to confirm theoretical results.