MITRO 209: Project

Baptiste Daumen - baptiste.daumen@telecom-paris.fr

31 January 2023

1 Note

The algorithm and all the tests are given in the Jupyter Notebook 'Mitro 209 Project - Baptiste Daumen'.
The graphs used have been downloaded from .txt files. To create adjacency lists from these files was very long. So I create adjacency lists from theses files only one time and I saved them in .pkl files. This is why I sent .txt and .pkl files.
All the code to make the save in .pkl is in comment on the Jupyter Notebook.
The algorithm creates the adjacency lists from .pkl, what is quicker.
Moreover I make the choice to write code of the algorithm in a single function. It is for 2 reasons. First to simplify the timing of my algorithm during tests. And because I think it's easier to see how the algorithm works in this configuration.

Enjoy your reading !

2 Presentation of the project

A classic problem in graph theory is to find the densest sub-graph of a graph. An algorithm to find a good approximation of the densest sub-graph(s) [1] is :

```
$H = G$;
while ($G$ contains at least one edge)
    • 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 $H \leftarrow G$;
return $H$;
```

The goal of this project is to find an implementation of the algorithm in linear time complexity depending on the graph size. Indeed, with huge graphs (millions of nodes and edges), an algorithm with time complexity worse than linear becomes unusable.

Thus, in this report, I'm going to present the algorithm that I have implemented.

3 Implementation

In this part, I'm going to present how I implemented the algorithm to find an approximation of a densest sub-graph. I will more precisely explain the structure of data that used.

3.1 Structures used

The graph is encoded with an adjacency list. The vertices are noted with numbers between 0 and $|V| - 1$. This representation gives us a size of the graph equals to $|V| + |E|$, with V the set of vertices and E the set of edges.
Thus, the complexity looked for is $\mathcal{O}(|V| + |E|)$.
The only data structure used is the list.

In the algorithm we initialise three lists :

vertices_erased : list of vertices treated by the algorithm. The list is sorted in the reverse order on which the vertices have been treated

vertices_by_degree : list of lists containing the vertices with the degree i, i being the index of the list in vertices_by_degree

vertices_data : a list of lists of size 3, the index corresponds to the number of the vertex. The shape of the list is
(actual degree of the vertex, 0 or 1 depending if the vertex has been treated by the algorithm or not, the index of the vertex in the list of vertices_by_degree where the vertex appears)

3.2 Explanation of the process

The first part of the code is the initialisation. We initialise all the data we need.

We compute the number of edges, the number of vertices, the density of the graph G. Then we initialise the three lists as described.

Finally, we compute the first minimum degree (min_degree), in order to choose the first vertex treated by the algorithm.

The second part is the description of one iteration of the while loop :
In the list vertices_by_degree[min_degree], we take the last vertex as the vertex treated at this iteration and we remove it from the list. For this, we use the function pop(). Let's note this vertex v.
We add v to the lists of treated vertices. Then in vertices_data[v], we update the field corresponding to the fact that the vertex has been treated, so the second field of vertices_data[v] becomes 1.
Then, we have to modify the properties of its neighbours. We iterate on all the neighbours of v. Let w a neighbour.

We have to make many operations on w. We want to erase w for the lists corresponding to its actual degree and add it in the list of its degree minus 1.

Let's describe the operations :
1. In the list vertices_by_degree[degree(w)] : we exchange w and the last vertex, noted x.
2. x has a new index in the list of elements of the same degree. It corresponds to the index of w. So we change its last field of vertices_data[x] by those of w.
3. We remove the last vertex (w) of vertices_by_degree[degree(w)] with pop()
4. We add w to the list vertices_by_degree[degree(w)-1] because the degree of w decreases of 1
5. We decrease the degree of w of 1 in vertices_data[w][0]
6. We change the data of vertices_data[w][2] by its new index in the list of vertices of same degree. It is at the end so it is len(vertices_by_degree[degree(w)]) - 1

These steps corresponds respectively to the lines in the loop on the neighbour of v :

```
1. vertices_by_degree[vertices_data[e][0]][vertices_data[e][2]]
    = vertices_by_degree[vertices_data[e][0]][-1]
2. vertices_data[vertices_by_degree[vertices_data[e][0]][-1]][2]
    = vertices_data[e][2]
3. vertices_by_degree[vertices_data[e][0]].pop()
4. vertices_by_degree[vertices_data[e][0]-1].append(e)
5. vertices_data[e][0] -= 1
6. vertices_data[e][2] = len(vertices_by_degree[vertices_data[e][0]]) - 1
```

In the first operation : the exchange is very important. It allows us to remove the vertex w from vertices_by_degree[degree(w)] in a constant time. Indeed, the function pop() on the list is in constant time when it is used on the last element contrarily to the function remove(), which has a complexity in the size of the list.

After theses operations, it remains to compute the density of the new graph. If the new graph has a better density, we keep in memory the vertices to erase to know a possible densest sub-graph.

Finally, we have to compute the new minimum degree. For making it, we are looking for in the vertices_by_degree list the first list not empty. We start our research at the index min_degree - 1 because the minimum degree can only decreases of one or increase until the number of vertices remaining.

4 Complexity

In this section we are going to prove that the algorithm has in the worst case a linear complexity in time. So it is in $\mathcal{O}(|V| + |E|)$.

First remind that the initialisation of a list, the access to an element in a list, the function append(), the function pop() (used on the last element of the list) and the computation of the length of a list in Python are operation made in $\mathcal{O}(1)$ time.

The step of initialisation in the code is in $\mathcal{O}(|V|)$. We only do loop on the set of vertices and operation in constant time.

The second step asks more study to compute the complexity. First notice that all operations made, except the two loops, are constant time operation. Indeed it are fundamental operation on lists and integers.

The first loop on the neighbours of v makes $\delta(v)$ iterations, with $\delta(v)$ the degree of v.

Let i the index of the iteration of the while loop. Let's note u_i the minimum degree of this iteration. The second loop has $u_i - u_{i-1} + 1$ iterations because when we find a list not empty we break the loop.

The main while loop has $|V|$ iterations. Indeed we iterate until the number of edges is equal to 0. So it could remain in the graph only vertices of degree 0. However, it is impossible because vertices of degree 0 are the first treated by the algorithm. So we treat all the vertices of the graph. It is why we are doing $|V|$ iterations.

Without taking in count constant time operations, which have low complexity compared to loops, the second step of the algorithm has the complexity:

$$\sum_{i=0}^{|V|-1} \delta(i) + \sum_{i=1}^{|V|-1} u_i - u_{i-1} + 1$$

We know that $\sum_{i=0}^{|V|-1} \delta(i) = 2 \times |E|$. Then,

$$\begin{aligned}
&\sum_{i=1}^{|V|-1} u_i - u_{i-1} + 1 \leq |V| \\
&\Leftrightarrow \sum_{i=1}^{|V|-1} u_i - u_{i-1} \leq 0 \\
&\Leftrightarrow \sum_{i=1}^{|V|-1} u_i \leq \sum_{i=1}^{|V|-1} u_{i-1} \\
&\Leftrightarrow \sum_{i=1}^{|V|-1} u_i \leq \sum_{i=0}^{|V|-2} u_i \\
&\Leftrightarrow u_{|V|-1} \leq u_0
\end{aligned}$$

This last inequality is true because $u_{|V|-1} = 0$.
Thus, $\sum_{i=0}^{|V|-1} \delta(i) + \sum_{i=1}^{|V|-1} u_i - u_{i-1} + 1 \leq 2 \times |E| + |V|$

Finally, the complexity time is $\mathcal{O}(|V| + |E|)$ for this step.
The algorithm has a total complexity in the worst case of $\mathcal{O}(|V| + |E|)$.

5 Tests

Let's test our algorithm on graphs. I choose 5 graphs from the website Stanford university [3].

1- Graph Social circles: Facebook : 4039 nodes and 88234 edges

2- Graph EU email communication network : 265214 nodes and 420045 edges

3- Graph Gnutella peer-to-peer network, August 31 2002 : 62586 nodes and 147892 edges

4- Graph Slashdot social network, November 2008 : 77360 nodes and 905468 edges

5- Graph Deezer Europe Social Network : 28281 nodes and 92752 edges

The algorithm returns the number of node in the densest sub-graph and its average degree density. Here are the results for each graph :

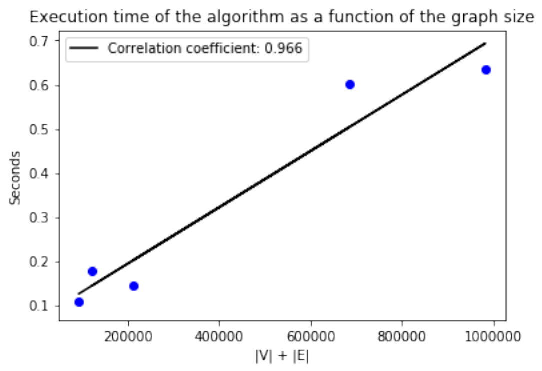
1- (202, 77.34653465346534) in 0.109 second

2- (205730, 0.9317412502717307) in 0.603 second

3- (8656, 4.906697395069488) in 0.145 second

4- (3262, 25.695055594984623) in 0.633 second

5- (150, 8.533333333333333) in 0.179 second



We have a very good correlation coefficient. The graph matchs with our theoretical complexity. Indeed the time complexity is linear as a function of the size of the graph.

References

[1] *Mauro Sozio : k-cores and Densest Subgraphs - slide 10*
`https://ecampus.paris-saclay.fr/pluginfile.php/1419484/mod_resource/content/1/Densest.pdf`

[2] *Mauro Sozio : Mitro 209 - Project*
`https://ecampus.paris-saclay.fr/pluginfile.php/1419486/mod_resource/content/0/TextLab.pdf`

[3] *Stanford university : Stanford Large Network Dataset Collection*
`http://snap.stanford.edu/data/index.html`