MITRO209 Projet - Léon Sillano

MITRO209 Projet - Léon Sillano

The project MITRO209 aims to create a 2-approx densest subgraph algorithm in linear time.

We consider a graph $G = (V, A)$, where V represents the vertices and A represents the edges.

The overall algorithm without implementation details is as follows:[1]

- $H = G$
- 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

Some propreties of the graph

Creating such an algorithm requires creating a more elaborate data structure than traditional graph data structures (matrix and adjacency list). However, certain steps are necessarily required:

- At each minimal degree vertex that will be removed from the graph, we must notify its neighbors that it will be removed. Lets calculate the overall complexity of this step.

We know that $\sum_{v \in G} \delta_G(v) = 2.|E|$. Thus, when we examine all the nighbours of all the vertices, we will have have a complexity of $O(2.|E|)$. It respaect the criterion of having a linear time algorithm in $|E|$

1. Data Structure

The choice of data structure is important in this problem. Indeed,

- If we choose an adjacency matrix as the graph implementation structure, its size being $|V|^2$, its construction is therefore at least in quadratic time. It is therefore impossible to use this structure.
- If we choose an adjacency list, even if this structure can be constructed in linear time, we will have problems in the algorithm that consists of selecting the smallest

[1] "k-cores and Densest Subgraphs", Mauro Sozio, Télécom ParisTech

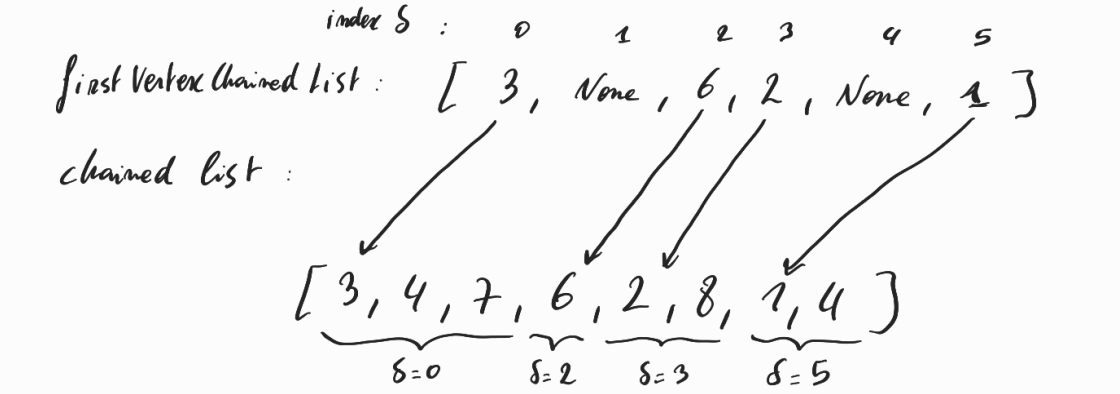
degree vertices (line 3 of the algorithm): we will only be able to do it in linear time. Thus, our algorithm will be at least in quadratic time.

The second implementation is however interesting, but we must create a data structure that allows selecting the smallest degree vertices in constant time. To do this, I decided to use an adjacency list (named `adjacencyList`) to which I added a new structure. This implementation that I chose is contained in the `Graph` class.

This latter, which makes it possible to overcome the problem of selecting the smallest degree vertices, is composed of:

- A **doubly-linked** list containing all the vertices, sorted according to their degree (in increasing order). This list is implemented by two classes created by myself: `ChainedList` and `NodeChainedList`. `ChainedList` store the pointer to the first node of the link list, and each node `NodeChainedList` has a pointer to it father and to it child.
- A list containing at index i the first vertex of the linked list such that its degree is equal to i. This list is called `firstVertexChainedList`. The size of this list is increased by 2. $|E|$ since the sum of the degree of each vertex is 2. $|E|$.

An example of the new data structure is presented below: NB. `firstVertexChainedList` contains actually pointers to the vertex of the chained list (`NodeChainedList` object)



In summary, my class `Graph` contains:

- An adjacency list adjacencyListPointers such that for each element in the list, there is associated a unique vertex $v \in V$ presented in this form:

  $$[id_v, \delta_g(v), d(v)]$$

  where id_v corresponds to the name of the vertex, $\delta_g(v)$ is the degree of v in G, $d(v)$ is the list containing the neighbors of v. It contains pointers to the neighboring vertices. Specifically, we have:

```
adjacencyListPointers
>>> [<NodeChainedList at 0x006ACAB90>, <NodeChainedList at 0x056AZE088>,...]

adjacencyListPointers[i].getObject() #NodeChainedList.getObject()
>>>[12,2,[<NodeChainedList at 0x006ACAB90>,<NodeChainedList at 0x056AZE088>]]
```

```
# The vertex 12 has 2 neighbours which are the object 0x006ACAB90 and 0x056AZ
E088.
# We can get the id of the neighbours vertex by using .getObject() on a NodeC
hainedList object
```

- The linked list `ChainedList` possessing the function `getElements()` allowing to retrieve all the elements of this linked list as a list, presented above.
- The list `firstVertexChainedList`, presented above.

2. Implementation of the algorithm & analyse of the complexity

2.1. Graph construction

The graph is constructed from a file containing the edges E For this,

- For each edge $v_1, v_2 \in E$, we store in `rawAdjacencyList` the data `[v1,v2]` and `[v2,v1]`: $O(|E|)$
- We merge vertices with the same id using a linear sorting algorithm based on the fact that we are handling integers: $O(|V| + |E|)$. We constitute `adjacencyList`
- We create the linked list `ChainedList` and the list `firstVertexChainedList` by placing each vertex in the correct place in `ChainedList` and updating `firstVertexChainedList` at each turn. The use of `firstVertexChainedList` allows to find the correct place of each vertex in linear time given that we consider the length of `firstVertexChainedList` to be constant. $O(|V|)$
- We also create a dictionary containing all the pointers of the vertices named `vertexReference`. We consider that searching for an element in a python dictionary is $O(1)$, which allows for linear time for this step: $O(|V|)$
- Finally, we create the adjacency matrix `adjacencyListPointers` containing pointers to the neighboring vertices from `adjacencyList` and `vertexReference`. $O(|E|)$.

In sum, graph construction is $O(|V| + |E|)$.

2.2. The 2-approximation densest subgraph algorithm

At each iteration of the algorithm,

- We retrieve the vertices with the smallest degree (and $\delta(v) > 0$). We use the first index of `firstVertexChainedList` for this. $O(1)$
- We set each of these vertices to $\delta(v) = 0$. It means that thses vertices are being deleted from the graph. This node becomes :

  $$[id_v, 0, d(V)]$$

  However, d(V) doesn't change during the algorithm (thus, it is not empty for a deleted vextex).
- For each neighbor of these vertices, we decrease the degree by 1 if the vertex has a degree greater than 0, otherwise we do nothing (his case corresponds a an already deleted vertex). Then, we replace the neighbour if the degree has changed at the right place in the chained list using `firstVertexChainedList`. $O\big(\delta(v)\big)$

- The vertices which was deleted at this iteration, we replace thme at the right place in the chained list using `firstVertexChainedList`
- Finally, if the density of the obtained graph decreases during this iteration, we remove the vertices that have obtained a null degree, i.e. the vertices that have been removed during this iteration from our current graph.

We place these iterations in a while loop, with the following exit condition: the current graph has no more vertices (and edges), this is characterized by the fact that in the adjacency matrix, each vertex has a null degree.

As said in the propresies of the graph paragraph, given that we iterate one time on each vertex in order to delet it, we will have for all the iterations in the loop about the first step: $O\left(\sum \delta(v)\right) = O(2.|E|)$ beacause with have $|V|$ iterations in the loop.

We have $O(|V|)$ iterations, each iteration is executer in constant complexity. So the overalll algorithm is in $O(|V| + |E|)$.

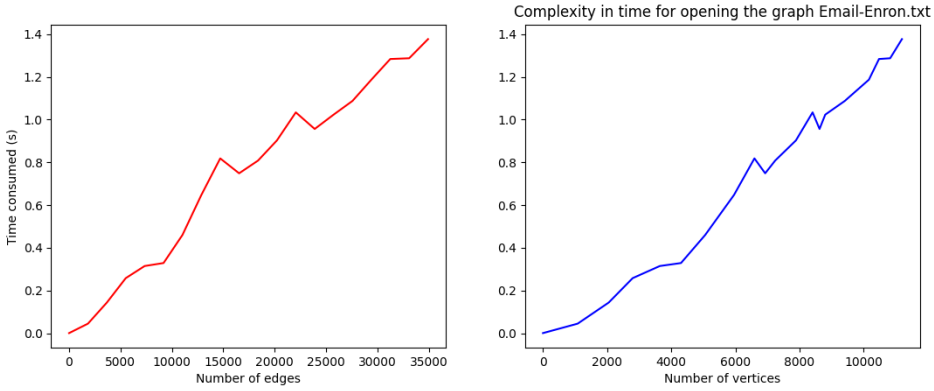
At the same time in the loop, I calculate at each iteration the density of the subgraph. If the sugraph H obtained after the iteration has a higher density than the previous one G, I keep H for the following iteration. Actually, to conserve the linearity of the algorith, I use the list of deleted vertices named `vertexDeletedForDensestSubraph`, which I update by the list `vertexDeletedCurrent` when the density increases. Then I do the difference (with `Graph2.getDifference()`) of the list containing all the initial vertices `allVerticesList` with the list of vertices which are not in the densest subgraph `vertexDeletedForDensestSubraph`. To do that, I do a linear sort on the two list, the I itrerate on both list at the same time to get the diffrerence.

3. Results

To visualize the results, I use a function that allows me to open a graph by limiting it to the `nb_rows` first degrees `fromCsvLimitRow()`. Thus, I can display two functions:

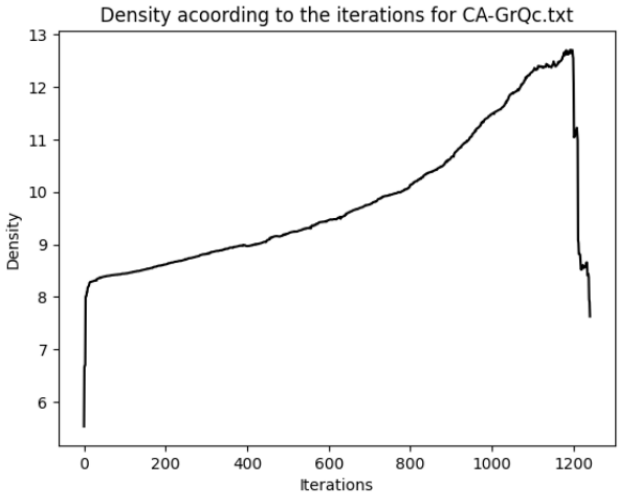
- Time consumed $= f\left(nb_{edges}\right)$, we have $nb_{edges} = nb_{rows}$
- Time consumed $= f(nb_{vertices})$, I calculate for each graph the number of vertices using $nb_{vertices} =$ `len(adjacencyList)`

I applied my algorithm on the data `Email-Enron.txt`



We see that the algorithm is in $O(|E|)$ (red function), and in $O(|V|)$ (blue function). The algorithm is thus working in linear time for this example. Actually, when we consider a clique, given that $|E| \approx |V|^2$, we will have the complexity of the algorithm in $O(|V|^2)$, so non linear in the number of vertices. This constitute the worst case of the algorithm. But on average, the algorithm is linear in the sum of vertices and edges.

Finally, I decided to plot the variation of the density according with the iterations for the graph:`CA-GrQc.txt`



We see that the density rises along with the iterations to reach the it max, which is the 2-approx max density of the graph, then it decreses when we delete vertices with high degree (which created the 2-approx densest subgraph)

Conclusion

The implementation of the 2-approx densest subgraph is in $O(|V| + |E|)$ **on average**. The worst case, given by a clique, is in $O(|V|^2 + |E|)$.