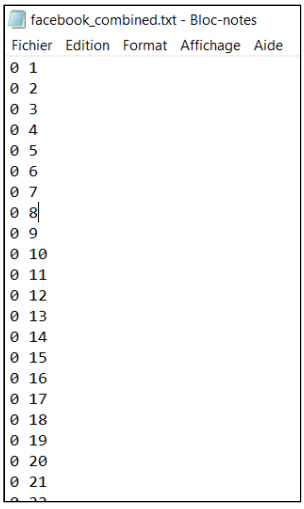
Introduction remarks :

I chose to program in **C**, in order to make the algorithm faster.
I tried my best to make the reading of the input file and the building of the data structures linear as well.
My program could probably still be a bit enhanced, with minor optimizations, but this is the architecture I wanted to build.

I - Input :

I consider an undirected graph, with n vertices, and m edges. The vertices are encrypted with positive integers, between 0 and n-1.
My input file is a text file where each line encodes an edge of the graph, using the numbers of the concerned vertices and a spacebar. If there is an edge between the vertices encrypted by p and q, then one of the line in the document is : "p q".
For example, the first edges of the graph *facebook_combined* look like this.



To compute a document, I have to specify
- the location of the input file
- the number of vertices of the graph
- the number of edges of the graph

For example, for the graphe wiki-topcats, I have to execute this line in a shell.

```
$ ./prgm.exe data/wiki-topcats.txt 1791489 28511807
```

II - Data structures :

I defined two structs, in order to manipulate the smallest variables possible.

```
25  typedef struct Sommet Sommet;
26  typedef struct Liste_chainee Liste_chainee;
27
28  struct Sommet {
29    int nom;
30    int degre;
31    Liste_chainee* adjacents;
32  };
33
34  struct Liste_chainee {
35      int sommet;
36      Liste_chainee* next;
37  };
```

Sommet stands for a vertex.
It contains :

- its name (an integer between 0 and n-1),
- its degree
- a pointer to list of its adjacent vertices

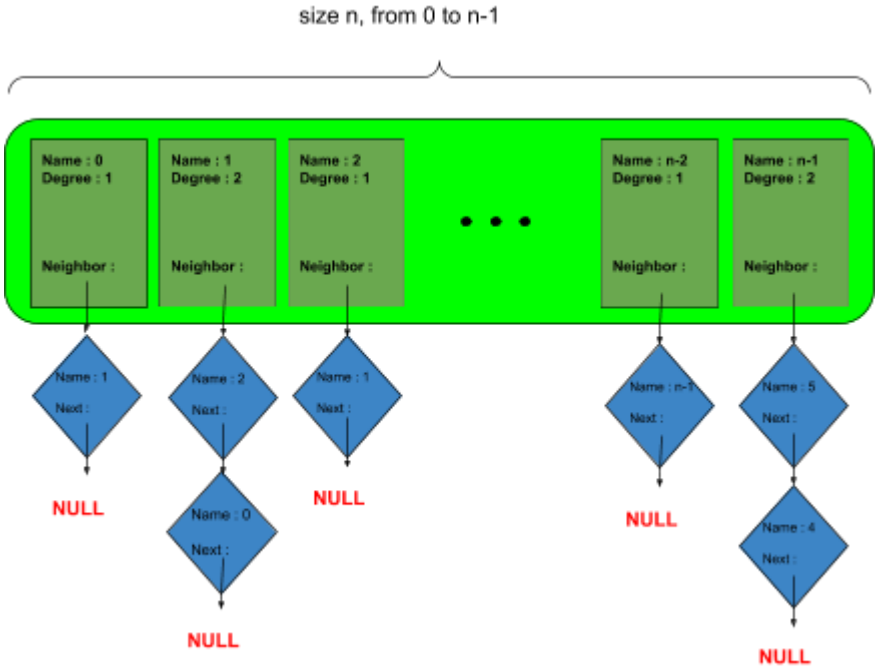
Liste_chainee is the simplest implementation of a list I could imagine. It contains :

- the number (equivalent to the name) of a vertex
- a pointer to the next link of the list

```
40  //Declaration of my 2 principal data structures
41    Sommet* sommets = (Sommet*)malloc(nbr_sommets * sizeof(Sommet));
42    Liste_chainee* tri_sommet = (Liste_chainee*)malloc(nbr_sommets * sizeof(Liste_chainee));
```

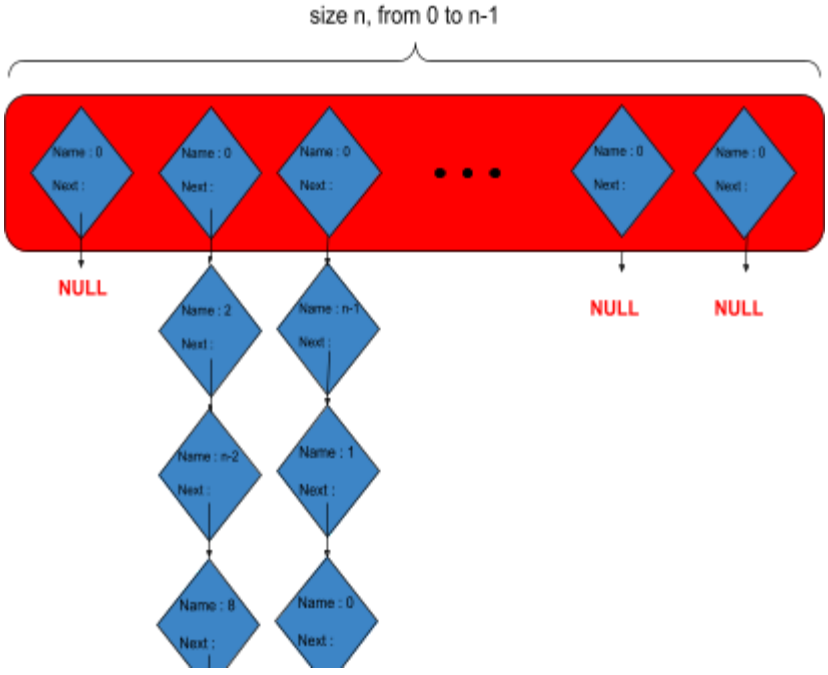
I use two tables, so that I can access their elements in O(1).
sommets stores pointers to every node in a Sommet struct.
tri_sommet will be later used to compute the algorithm.

Organisation of sommets



As I read the input document line by line, I fill **sommets** to store every information on each node.
When I read an edge that contain the nodes a and b :

- If a (resp. b) hasn't already been encountered, a struct of type **Sommet** is allocated in memory, and it's pointer is stored at the index a (resp. b)
- the pointer to a (resp. b) is added at the beginning of the list of the neighbors of b (resp. a)
- the degree of a (resp. b) is incremented by 1



Remark : Notice that this algorithm allows me to organize the data in linear time.

Organisation of tri_sommet

At an index k, this table stores a **liste_chainee** of all nodes of degree k (*at least at its creation*).
After computing **sommets**, I browse every node in it (O(n)), and for each node, I store it at the second element of the right list (every first element is just an empty link).

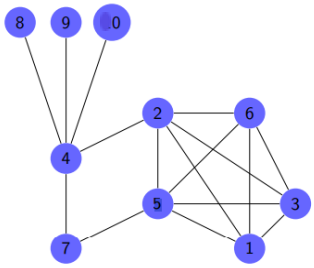
There can begin the computing of the algorithm.

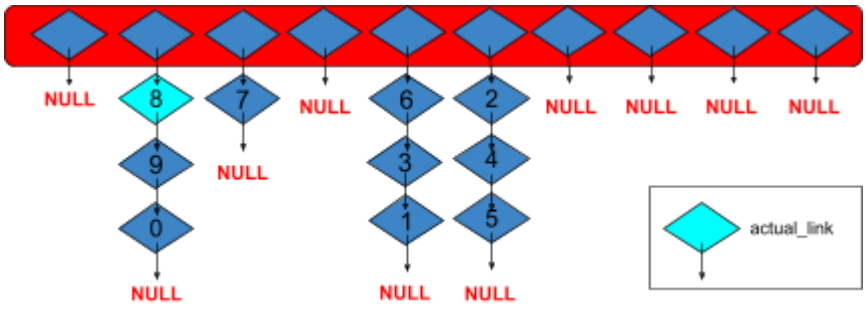
III - The greedy algorithm

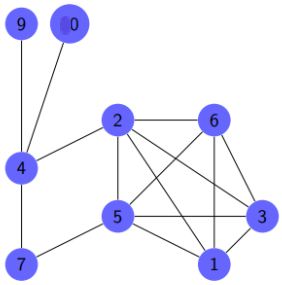
I had 2 variables to store the densest subgraph : **deleted**, a table of size nbr_nodes filled with 0, and **update**, an empty link.

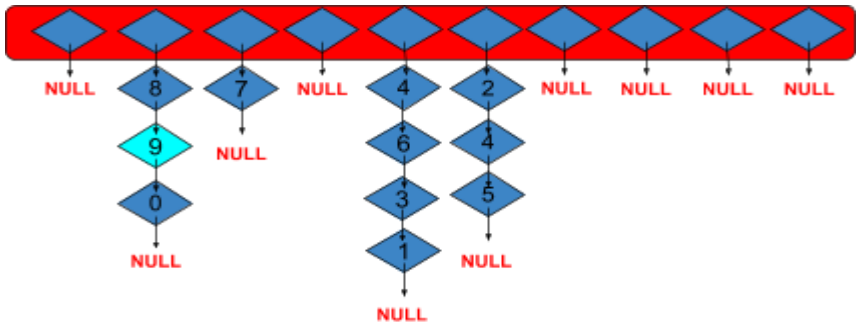
```
iter ← 0

While there is at least one vertex:
    actual_link ← tri_sommet[iter]
    While actual_link isn't the NULL link:
        //the deleted or isolated nodes don't have to be reprocessed
        If sommets[actual_link.sommet].degre > 0:
            //We set the new density
            nbr_vertices -= sommets[actual_link.sommet].degre;
            nbr_nodes -= 1;
            new_density = nbr_vertices/nbr_nodes;
            //we will now update the neighbors
            neighbors ← sommets[actual_link.sommet]).adjacents;
            While neighbors isn't the NULL link:
                sommets[neighbors.sommet].degre -= 1;
                //they are 3 possibilities
                //(i) the neighbor  node has already been deleted or isolated: its degree has already been set to -1,
                or 0. We ignore it
                if(sommets[neighbors.sommet].degre <= 0):
                    neighbors ← neighbors.next;
                    continue;
                //(ii) the neighbor node's degree is equal to the degree of the actual link's node, or greater by one
                //we insert a link for the neighbor node as the direct successor of the actual link : it will be the next
                deleted node
                if(sommets[neighbors.sommet].degre <= sommets[actual_link.sommet].degre):
                    insert a new link for this node as the direct successor of the actual link in tri_sommet
                //(iii) the node's degree is strictly greater than the degree of the actual link's node +1
                if(sommets[neighbors.sommet].degre > sommets[actual_link.sommet].degre):
                    insert a new link at the beginning of the list in tri_sommet of index of its new degree;
                neighbors ← neighbors.next;
            sommets[actual_link.sommet].degre = -1;
            add a new link for node actual_link.sommet in update;
        //if the node hasn't been deleted before : we delete it
        If sommets[actual_link.sommet].degre == 0:
            sommets[actual_link.sommet].degre = -1;
            add a new link for node actual_link.sommet in update;
        if the new density is greater than the actual density:
            Store it, with its associated vertices and nodes numbers;
            Set to 1 the nodes in deleted stored in update
            Empty update
        actual_link ← actual_link.next;
    iter ← iter + 1;
```

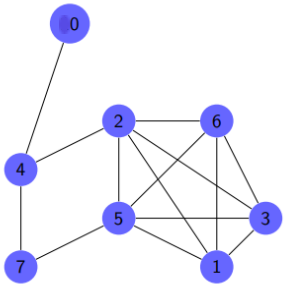
I'll illustrate **tri_sommet** at the beginning of the 3 first iterations with the graph given in example. (I replaced 10 by 0)

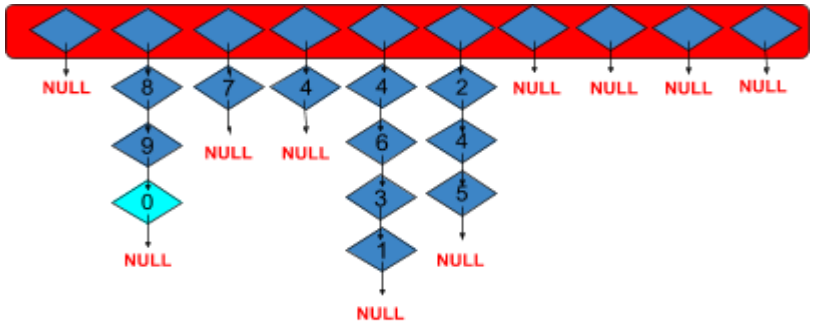












IV - Proof of linearity

We see that I sort of duplicate the links of neighbors nodes in **tri_sommet**.
δ(v) denotes the degree of a vertex v in the whole graph, n the number of vertices and m the number of edges.
As I used tables and only insert links at the beginning of lists, every operation inside my loops are in O(1). I'll now show that the total number of operations is linear.

$\sum_{i=1}^{n} \delta(v_i) = 2m$

Every vertex v will be duplicated in the worst case δ(v) times.
We know this equality So when browsing **tri_sommet**, I will in the worst case meet each node 2m + n times.

But I'll delete a vertex only the first time I meet it. So I browse its neighbors' list only once, in order to update them. In the worst case, I'll have to update δ(v) vertices for each vertex. This occurs in the worst case the n first times I meet this vertex. In total, another 2m operations. I finally add the browsing of empty values in **tri_sommet**, in O(n).

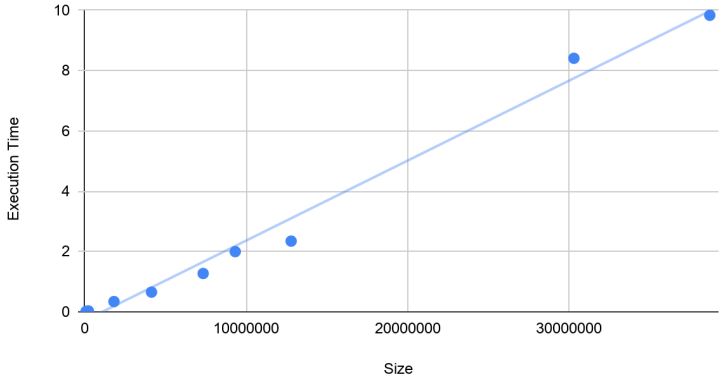
In conclusion, the complexity of the algorithm is strictly inferior to 4m + n, so linear in n+m, the size of the graph.

V - Results

I measured and plotted the execution time of the central algorithm for some graphs.

	Nodes	Edges	Size	Execution Time	Density	Nodes of subgraph	Edges of subgraph
facebook_gouv.txt	7057	89455	96512	0,013	35,116183	241	8463
as-skitter.txt	1696415	11095298	12791713	2,346	88,687961	407	36096
facebook_combined.txt	4039	88234	92273	0,009	77,346535	202	15324
com-lj.ungraph.txt	4036538	34681189	38717727	9,835	190,984456	386	73720
twitch_fr.txt	6549	225332	231881	0,032	68,254005	874	59654
com-youtube.ungraph.txt	1157827	2987624	4145451	0,654	45,548519	2195	99979
wiki-topcats.txt	1791489	28511807	30303296	8,409	67,855989	12624	856614
flickr.edges	215495	9114557	9330052	1,997	268,97855	979	263330
lastfm.edges	136420	1685524	1821944	0,341	106,614583	864	92115
myspace.edges	854498	6489736	7344234	1,272	15,031932	118064	1774730

Execution Time by Size



Remark : I didn't have the time to recompute all the graphs, but I got better computing times using the computers of the school, with Unix.

VI - Final remarks

Here are some hints to reduce the complexity of my implementation :

- change **tri_sommet** into a list of lists, to reduce its length : with a table, most of its values are empty
- don't duplicate the nodes in **tri_sommet**. This can be done by adding for every **Liste_chainee** a pointer to the previous link, and in every node in **sommets** a pointer to its link in **tri_sommet**.

These two enhancements would lead to a complexity in the worst case in 2*m.

I first couldn't run the algorithm for large graphs. I found out that I couldn't create a table with `Sommet* sommets[ N ];` if N was strictly greater than 519276.
As I wrote at the beginning of this report, I had to use malloc to resolve this static definition.