4BIM

# Theoretical Computer Science Project

Hanâ LBATH, Emilie MATHIAN

January 15, 2019

# Contents

# 1 Exercise 1

## 1.1 Data import

All the program associated with the project is saved in a Jupyter notebook named `Projet_info_theorique.ipynb`. First we have to load the data:

- The file `GRN_edges_S_cerevisia.txt`, is loaded as a Pandas data frame object, named `edges`, this data frame contains two columns. The first one refers to the transcription factors and the second one to their target genes.
- The second files `net4_transcription_factors.id` containing the transcription factors id, is loaded in a data frame named `transcription_factors_id`.
- In the two previous files, each gene is identified by an id, the real locus named could be find in the file `net4_gene_ids.tsv`, this one is loaded is loaded in data frame called `gene_id`. Owing to this supplementary information, we can find a gene of interested in database such as NCBI.
- With the incredible increasing amount of biological data, generating by NGS, computer scientists created a tool to store gene informations in a comtutable form. Owing to the gene ontology format, each gene is described according to the molecular function of the gene product, the biological component where the gene product performs a function and by the main biological process accomplished by the molecular activity. Like this, the file named `go slim mapping.tab.txt` contains :
  - the common gene name,
  - its SGD id ( Saccharomyces Genome Database)
  - its Aspect, this field contained a single letter which could be :
    P (= biological Process), F (= Functional process) or C (cellular Component),
  - its GO id,
  - its object type, that it's to say the type of the gene product, (eg. tRNA_gene, or protein).

  All these information are stored in data frame named `gene_go_mapping`.

## 1.2 Network visualization

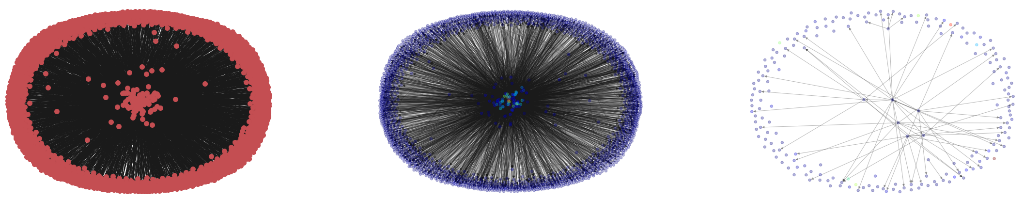


Figure 1: Network visualization : a) all nodes b) all nodes the color represent their degree (yellow high degree, navy blue low degree), c) subset of nodes with their degree. (cf: function `graph_with_degree|))`

With these first representations we understand the complexity linked to the visualization. The first comments that could be infer of the graph [1a] is the high number of nodes. Moreover we notice that there is a few number of nodes with high degree. That is why we decide to distinguish nodes according to their degree, this is represented in the graph [1b]. This representation offers a global visualization of the network but it is still difficult to read. That is why we could draw the same graph for a subset of nodes of interest [1c]. According to these representation we could say that few gene are involved in regulation, and those who have this function (= transcription factor) interact with an high number of gene.

According to this last remark we could decide to represent one transcription factor and the genes regulated by this one. This representation could give an idea of the influence of a transcription factor. Nevertheless this visualization is restrictive because it removes all other regulations that could regulate the transcription of the neighbors of the transcription factor of interest, and its own regulation.

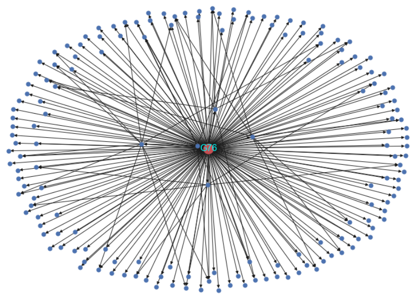


Figure 2: representation of 'G76' and its neighbors (cf: function `egocentric_graph|)`.

To overcome this problem, we could choose for one transcription factor to represent the network resulting of DFS algorithm. This is more or less readable according to the centrality of the transcription factor, that's why we could display or not the labels. Indeed depending on the transcription factor's degree and it's interaction with other transcription factor this representation could more or less complex.

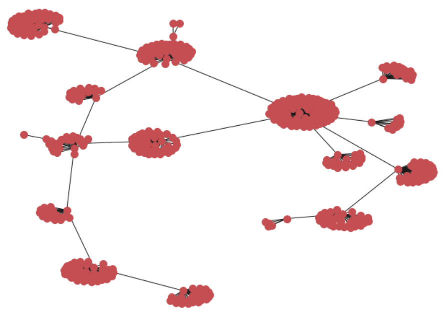


Figure 3: representation DFS network with 'G87' as start node. (cf : function `DFS_graph|)`

We could display network according selecting one Louvain community. This representation allows to display a subset of nodes that are more linked together than to others nodes. We chose to display this representation to illustrate the exercise 2.

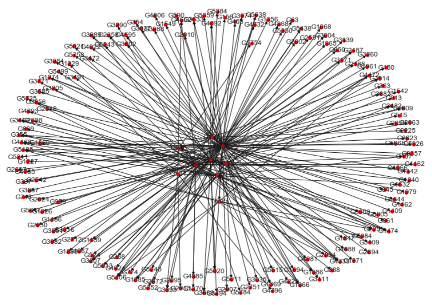


Figure 4: representation of one Louvain community. (cf: function `louvain_community_graph|)`

Finally, we chose to represent the network between a subset of edges of interest. This last representation allows a specialist to have an easy representation of selected interactions.

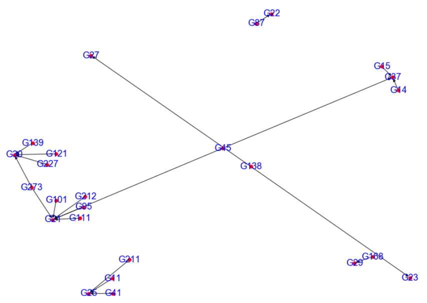


Figure 5: Network create given a list of edges.

## 1.3 Networks informations and metrics

### 1.3.1 General informations

The total networks that could be created using the list of edges given by `GRN_edges_S_cerevisia.txt` files contains :

- 1994 nodes
- 3949 edges
- the average in-degree is 1.98
- the average out-degree is also 1.98

These information have been generated by the command `nx.info(G)`. According to them we could calculate the density, which equals $9.9e-4$. According to this global metric we could notice that the gene regulation network is very sparsley connected. To calculate the density we could apply the following algorithm :

```
1: function DEGREE DISTRIBUTION($A$)
2: "'A is the adjacency matrix of a graph $G =< V, E >$. |V| is the number of nodes and |E| the
   number of edges"'
3:     $|E| = sum(A)$
4:     $Max_|E| = |V|(|V| - 1)$
5:     $D = 2|E|/Max_|E|$
       Return $D$
6: end function
```

The time complexity of the previous algorithm is :

$$\mathcal{O}(|V|^2)$$

Indeed we have to sum the adjacency matrix, of size $|V|^2$ . So we have to do $|V|^2$ operations. Then we could assume that the cost to calculate the maximal number of edges is constant, because we could access in constant time to the size of the adjacency matrix, which gives us the number of vertices.

### 1.3.2 Degree distribution

Firstly we decided to present the degree distribution, using three indexes : the in-degree distribution , the out-degree distribution and the general degree distribution. This information allows us to distinguish different types of genes and their relative frequency in the network. For example we could visualize the proportion of gene that an important out-degree, these genes are important regulation factor. Reciprocally we could see the proportion of gene that have an high number of in-degree, those genes are finely regulated.

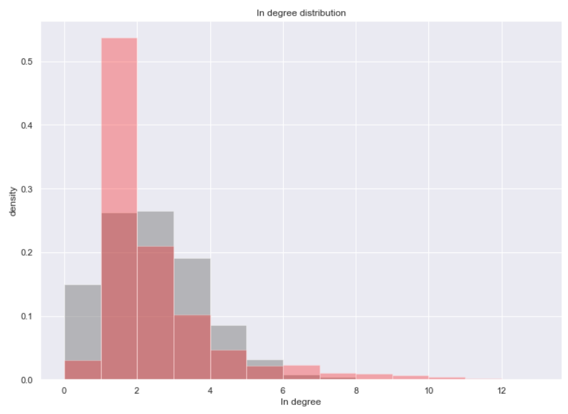


Figure 6: In-degree distribution, the red histogram represent the network, and the grey one a random network create by using Erdos Renyi definition. (cf : function Hist_degree)

The in-degree distribution shows that most of the genes are regulated by 1 to 4 transcriptions regulators, this is expected given the average in-degree. Besides, the proportion of nodes with an in-degree equal to 1 is higher than the one expected in a random network. We notice that a non-negligible number of gene are not regulated. This is biologically interesting because either these genes have a constitutive expression and so don't need any transcription factor, or we could postulate that the transcription factor of these gene haven't been discovered yet. A few number of genes have an high degree distribution. The gene with the highest in-degree (12) is 'YDR527W', this gene interacts with RNA pol II and is involved in transcription regulation.

To get a readable plot of the out-degree distribution, we had to remove all nodes with an out-degree equal to 0, which are target genes. Then if we analyze the distribution induced by the remaining nodes, we note that transcription factors regulated a number of gene extremely variable. Most of these transcription factors (60%) control the transcription of a single node. Besides, the out-degree distribution of the biological network is shifted to the left compare to the random network. However we also notice that few transcription factors regulates a very high number of genes. The transcription factor with the highest degree regulates 218 genes.

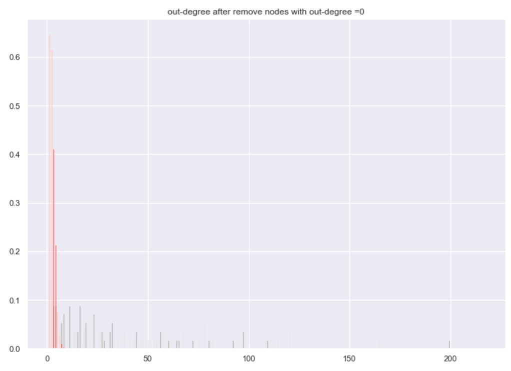
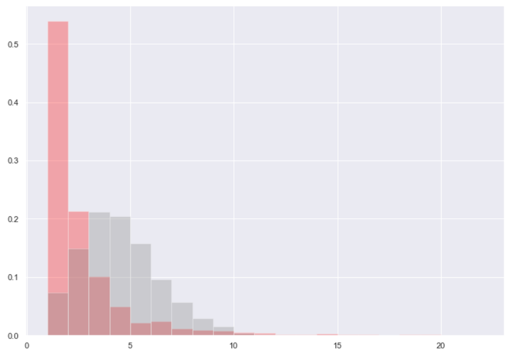


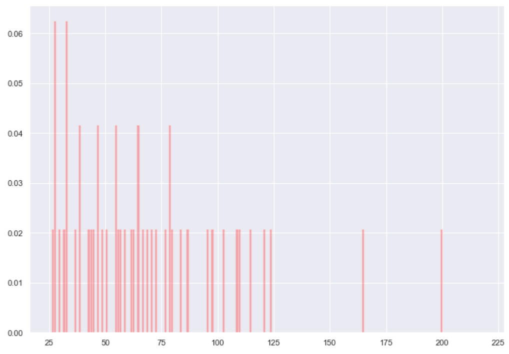
Figure 7: Distribution of out-degree after removing the node with an out-degree = 0. In red distribution of our biological network, in grey distribution of a random network.

Finally we represent the general degree of distribution. Firstly, to get a better visualization we decided to removes all nodes with a degree higher than 25. We observed that the distribution of the biological network is shifted to the left compare to the random one. Secondly we represent only the genes with a degree with a degree over than 25. In this case we observe that the proportion of nodes keeping for the random network is negligible. In conclusion the regulation gene network is characterized by :

- A high number of genes with a low degree of interaction. This proportion is highly greater than the one expected in random case.
- Few genes have a very high degree of interaction. This proportion is over the one expected in random case.



(a)



(b)

Figure 8: Degree distribution: a) for nodes with a degree lower than 25, b) for genes with a degree over than 25

```
1: function DEGREE DISTRIBUTION($A, deg\_type$)
2: ”’A is the adjacency matrix, deg type could be ’in’, ’out’, ’both
3: V is the number of nodes in the graph G”’
4:     if deg_type == ’In’ then
5:         $degree\_dict = \{\}$
6:         for v in V do                                  ▷ $|V|$ iterations
7:             $degree\_dict[v] = sum(A[:, v])$           ▷ $\mathcal{O}(|V|)$
8: # sum the column of A corresponding to v
9:         end for
10:    end if
11:    if deg_type == ’Out’ then
12:        $degree\_dict = \{\}$
13:        for v in V do                                  ▷ $\mathcal{O}(|V|)$
14:            $degree\_dict[v] = sum(A[v, :])$           ▷ O(|V|)
15: # sum the lines of A corresponding to v
16:        end for
17:        if deg_type == ’both’ then
18:            $degree\_didt = \{\}$
19:            for v in V do                              ▷ $|V|$ iterations
20:                $degree\_didt[v] = sum(A[v, :]) + sum(A[:, v])$   ▷ $\mathcal{O}(2|V|)$
21:            end for
22:        end if
23: Return $degree\_dict$
24:
```

According to thes line 5,11 and 16 the number of operations it at least of |V|, the number of vertices. Then for the line 6 and 12 there are |V| operation to to do, it's the cost of the sum. According the line 17 there is at least 2|V| operations to do. So the time complexity of this algorithm is :

$$\mathcal{O}(|V|^2)$$

To obtain this result we assume that the time to reach a column or line is constant. Then we have take into account the time complexity to build the adjacency matrix.

### 1.3.3 Rich club

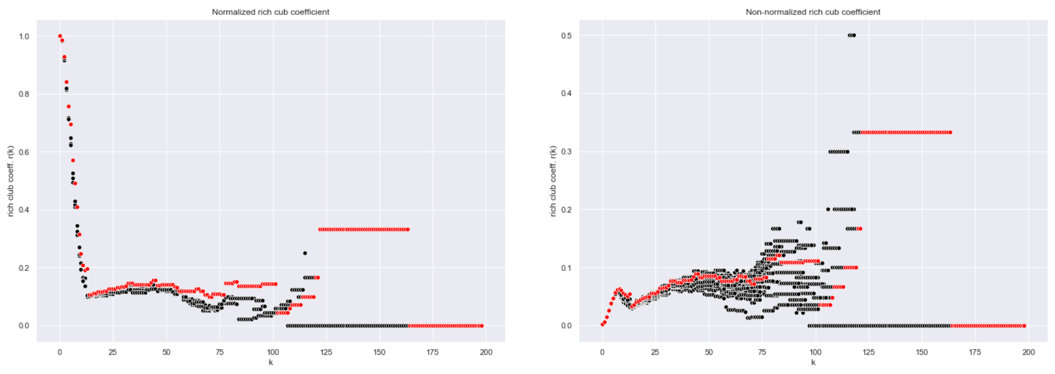
The second metric that we chose to present is the rich club coefficient. We can recall that the rich club coefficient represents how well connected nodes are well connected between themselves. The formula associated to this metric is :

$$\phi(k) = \frac{2|E \geq k|}{|V \geq k|(|V \geq k| - 1)}$$

Where $|E \geq k|$ is the number of edges that linked nodes which have a degree at least equal to $k$. This metric is normalized by the number of possible edges between the remaining nodes. Then the maximal rich club coefficient is equal to 1. This metric could be criticized, because the rich club coefficient automatically increases with the number $|E \geq k|$, that is why we also present a normalized form of this metric defined such that :

$$\rho(k) = \frac{\phi(k)}{\phi_{rand}(k)}$$

Where $\phi_{rand}(k)$ is the rich club coefficient calculated for a random graph generating with $|V \geq k|$ nodes. Hence, if $\rho(k) > 1$, there is a rich club effect.
Biologically the rich club coefficient is interesting. Indeed, it shows if transcription factors regulate themselves.



(a) (b)

Figure 9: The rich club coefficient is calculated on the original network (red), and for 25 random networks (black) : a) normalized rich club coefficient b) non normalized rich club coefficient; for all different degrees identified in the network $k$. The randomization have been done by shuffling the list of target gene, like this we have conserved the degree of each node, but we broke the possible communities.

For genes that have a degree inferior to 125, we can't detect a reach club effect, even if the rich club coefficient is scarcely above the random one. However for genes with a degree between $125 < k < 160$, we detect rich clubs, indeed the rich club coefficients calculated for these degrees is often above those calculated for the random networks. The rich club coefficient for these nodes is around 0.30, which is highly superior to all other nodes. For nodes with a degree superior than 160, the rich club coefficient become again low.
The main information that we could infer from this graph is : there is an high degree of interaction between a class of transcription factors, that have a degree $(125 < k < 160)$, so these transcription factors regulate among themselves.

We could try to write a pseudo algorithm computing the rich club coefficient.

```
function RICH CLUB COEFFICIENT($A, G =< V, E > degree\_dict$)
"'A is the adjacency matrix of the graph G
degree_dict is a dictionnary containing nodes and their degree."'
    $max\_degree = max(degree\_dict.values())$        ▷ $\mathcal{O}(log(|V|))$ if degree_dict is sorted
    $Ek = []$                                          ▷ $\mathcal{O}(1)$
    for k in from 1 to max_degree: do                  ▷ $|E|$ iterations
        for v in degree_dict.keys() do                 ▷ $|V|$ iterations
            if degree_dict[v]<k then                   ▷ $\mathcal{O}(1))$
                remove A's column corresponding to v   ▷ $\mathcal{O}(1))$
                remove A's line corresponding to v     ▷ $\mathcal{O}(1))$
            end if
        end for
    end for
    E = sum(A)                                         ▷ $\mathcal{O}(|v \geq k|^2))$
    Ek.append(E)                                       ▷ $\mathcal{O}(1)$
Return Ek
end function=0
```

This algorithm assumes that the degree distribution dictionary was calculated (using the precedent pseudo code for example), and sorted (using a quick sort algorithm), like this summing we could estimate the time complexity for rich club coefficient such as :

$$\mathcal{O}(3|V||E| + |V|^2 + log(|V|)$$

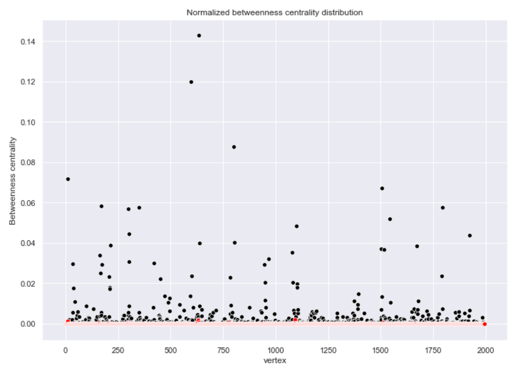
which is equal to $\mathcal{O}(|E||V| + |V|^2)$

### 1.3.4 Betweenness centrality

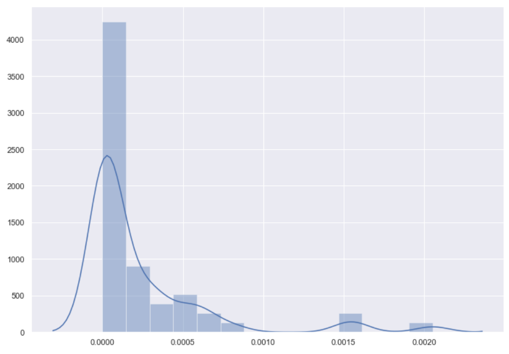
The next metric used to describe our gene regulation network is the betweenness centrality. This metric is calculated such that :

$$b_v = \sum_{V_s \neq v \neq V_t} = \frac{\sigma_{V_s,V_t(V)}}{\sigma_{V_s,V_t}}$$

Where $\sigma_{V_s,V_t(V)}$ is the number of shortest path between $s$ and $t$ passing through v, and$\sigma_{V_s,V_t}$ is the number of shortest path between $s$ and $t$. This metric could be normalized by dividing result by the maximal number of paths.
This metric could be useful to identify the most central genes in a network. These genes are probably important transcription factors but they could also be transcription factors that are highly regulated. Like this we could identify the genes that don't act as transcription factors, the genes which are important transcription factors and a third category, not described by the degree distribution, the transcription factors that are highly regulated.



(a)



(b)

Figure 10: a) Representation of the normalized betweenness centrality, original data are represented in (red), and random data in (black). The random network is generating by shuffling the target list. b) Distribution of the betweenness centrality on the original data.

Most of the genes have a betweenness centrality equal to zero. This is a reasonable result such that there is only 114 different transcription factors among 2048 genes. This result is highly lower by the one expected in random case. The distribution of the betweeness centrality infers the same conclusions.

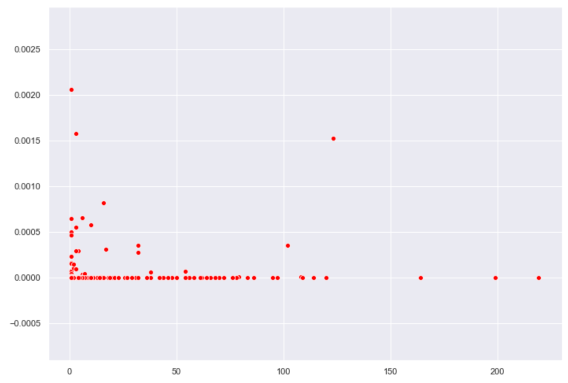


Figure 11: Representation of the betwenness centrality in function of the nodes' degree. According to this graphic the betweeness centrality is not correlated with the degree. Like this the genes with the highest degree are not necessarily those who have the hightest betweenness centrality

## 1.4 Pearson degree-degree correlation

The last metric that we decide to present is the Pearson degree-degree correlation. the formula of this metric is :

$$r = \frac{1}{\sigma_q^2} \sum_{k,l} k \times l \times (e_{kl} - q_k q_l)$$

Given a edge $< v_i, v_j >$,e$_{kl}$ represents the probability that the ends of $< v_i, v_j >$ have a degree k and l, $q_k$ is the probability that one end of $< v_i, v_j >$ has a degree k, and reciprocally for $q_l$, $\sigma^2$ is the variance of the distribution $q_k$. The Pearson degree-degree correlation allows to characterize a graph according 3 category :

- if $r = 0$, the graph is said neutral. The nodes with a low dregree could be connected indifferently to high degree node or low degree nodes .
- $r < 0$, the graph is say disassortative. The nodes with a low degree are more likely to be connected with nodes with a high degree.
- $r > 0$, the graph is assortative. The graph edges tend to connect nodes with a similar degree.

The biological meaning of a assortative network is that the transcription factor tend to have more links between themselves than with others genes. In a disassortative network, the transcription factors tend to have more links with their targets than links between themselves. Finally a neutral network means that transcription factor are linked with others transcription factors and also with others genes.
The assortativity on the gene regulation network is :

$$r = 0.02699$$

The assortativity calculated is close to zero, so transcription factor are linked with others transcription factors and also with others genes (targets).
We tried to write a pseudo algorithm to calculate the assortativity.

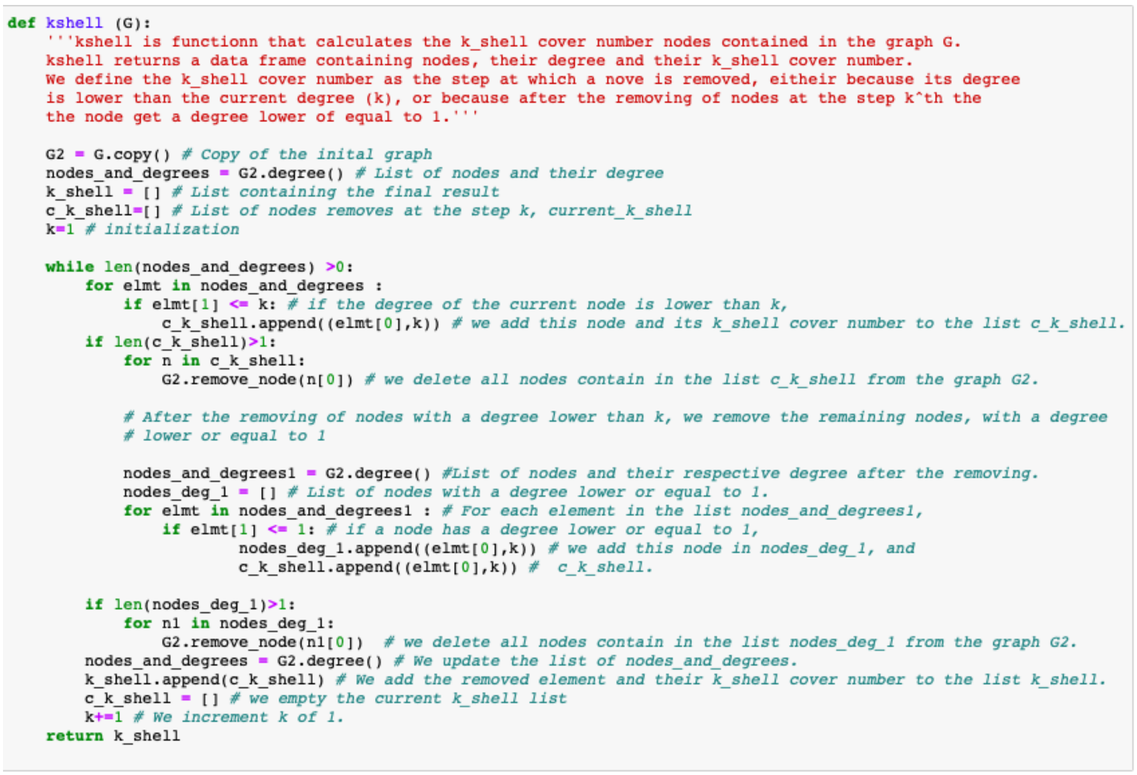
```
1: function DEGREE DEGREE PEARSON COEFFICIENT($G =< V, E >, degree\_dict$)
2: ”’ the graph G contains a list of edges E
3: E is a list of edges such that $< v_i, v_j >$; degree_dict is a dictionary that associated each node
   to its degree.”’
4:     $edges\_degree\_list = []$
5:     for e in E do ▷ |E|$iterations$     $(k, l) = (degree\_dict(e[0], degree\_dict(e[1])$ ▷ $\mathcal{O}(2)$

6: # The time complexity assumes that the access to a element in a dictionnary is constant
8: # (k,l) is a tuple with the degrees of $v_i$ and $v_j$
9:
10:        $edges\_degree\_list.append((k, l))$                                        ▷ $\mathcal{O}(1)$
11:
12: Return Stat Pearson (edges_degree_list[0] , edges_degree_list[1])
13: # Calculation of the Pearson correlation of coefficient between the first elements and the second
    elements of edges_degree_list
14: =0
```

```
1: function STAT PEARSON COEFFICIENT($X, Y$)
2: ”’ X and Y are two vectors, N is the number of elements.”’
3:     $Mean_X = sum(X)/N$                                  ▷ $\mathcal{O}(N)$
4:     $Mean_Y = sum(Y)/N$                                  ▷ $\mathcal{O}(N)$
5:     $d_X = (X - Mean_X)$                                 ▷ $\mathcal{O}(N)$
6:     $d_Y = (Y - Mean_Y)$                                 ▷ $\mathcal{O}(N)$
7:     $num = sum(d_X * d_Y)$                               ▷ $\mathcal{O}(N + (N - 1))$
8:     $d_X^2 = (X - Mean_X)^2$                             ▷ $\mathcal{O}(2N)$
9:     $d_Y^2 = (Y - Mean_Y)^2$                             ▷ $\mathcal{O}(2N)$
10:    $denom = Sqrt(sum(d_X^2)sum(d_Y^2))$                 ▷ $\mathcal{O}(3N + 1)$
11:    $r = num/demon$                                      ▷ $\mathcal{O}(1)$
12: Return r
13:
```

The `Stat Pearson` algorithm has a global time complexty equal to $\mathcal{O}(N)$. Like this if `Stat Pearson` takes as entry two lists of $|E|$ elements, `degree degree Pearson coefficient` have a global time complexity of $\mathcal{O}(2|E|) = \mathcal{O}(|E|)$.

## 1.5 K-shell

The k-shell algorithm allows to identify central and dense cores of a graph. Our version of k-shell algorithm is describes bellow:



```python
def kshell (G):
    '''kshell is functionn that calculates the k_shell cover number nodes contained in the graph G.
    kshell returns a data frame containing nodes, their degree and their k_shell cover number.
    We define the k_shell cover number as the step at which a nove is removed, eitheir because its degree
    is lower than the current degree (k), or because after the removing of nodes at the step k^th the
    the node get a degree lower of equal to 1.'''

    G2 = G.copy() # Copy of the inital graph
    nodes_and_degrees = G2.degree() # List of nodes and their degree
    k_shell = [] # List containing the final result
    c_k_shell=[] # List of nodes removes at the step k, current_k_shell
    k=1 # initialization

    while len(nodes_and_degrees) >0:
        for elmt in nodes_and_degrees :
            if elmt[1] <= k: # if the degree of the current node is lower than k,
                c_k_shell.append((elmt[0],k)) # we add this node and its k_shell cover number to the list c_k_shell.
        if len(c_k_shell)>1:
            for n in c_k_shell:
                G2.remove_node(n[0]) # we delete all nodes contain in the list c_k_shell from the graph G2.

            # After the removing of nodes with a degree lower than k, we remove the remaining nodes, with a degree
            # lower or equal to 1

            nodes_and_degrees1 = G2.degree() #List of nodes and their respective degree after the removing.
            nodes_deg_1 = [] # List of nodes with a degree lower or equal to 1.
            for elmt in nodes_and_degrees1 : # For each element in the list nodes_and_degrees1,
                if elmt[1] <= 1: # if a node has a degree lower or equal to 1,
                        nodes_deg_1.append((elmt[0],k)) # we add this node in nodes_deg_1, and
                        c_k_shell.append((elmt[0],k)) #  c_k_shell.

        if len(nodes_deg_1)>1:
            for n1 in nodes_deg_1:
                G2.remove_node(n1[0])  # we delete all nodes contain in the list nodes_deg_1 from the graph G2.
        nodes_and_degrees = G2.degree() # We update the list of nodes_and_degrees.
        k_shell.append(c_k_shell) # We add the removed element and their k_shell cover number to the list k_shell.
        c_k_shell = [] # we empty the current k_shell list
        k+=1 # We increment k of 1.
    return k_shell
```

Figure 12: Description and code of k-shell algorithm.

According to the graph [13], we could see that the dense cores are associated to high value of k-shell number. Like this most of the nodes associated to a low k-shell number are in the corona. This representation is nearly equivalent to the one give in [1b]. Indeed the k-shell is a representation of the degree distribution.

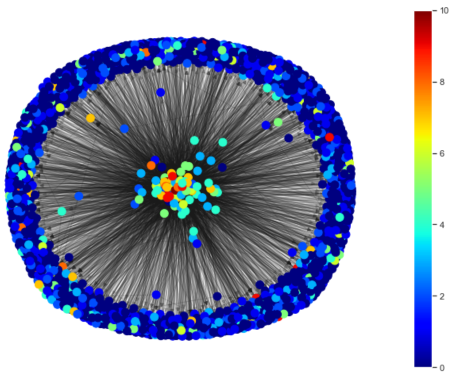
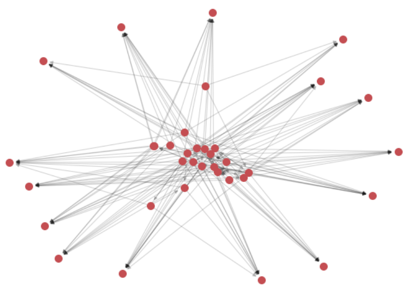
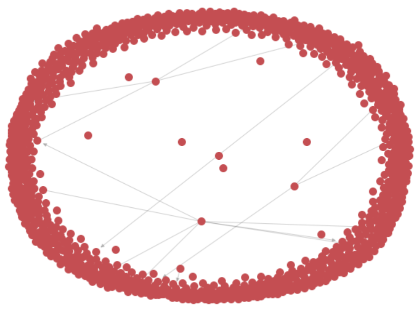


Figure 13: Representation of the networks: the color associated to each node represents the k-shell number. The value of the k-shell number is given by the bar drawn on the left side.

We could also understand the meaning of the k-shell by representing the k-core [14a] and the k-shell [14b]. Owing to these representations we could highlight that the transcription factors are highly linked together, whereas the target genes share are more independent.



(a)



(b)

Figure 14: a) Representation of the k-shell core : remaning nodes after the $8^{th}$ step b) Representation of the k-shell : removing nodes before the step 2.

## 2 Exercise 2

### 2.1 The Girvan-Newman Algorithm

The Girvan-Newman algorithm suggests to detect communities by detecting the edges "between" comunities, i.e. the edges that connect different comunities. In order to do so, the edge betweenness metric is used. The edge betwenness of an edge can be defined by " the number of shortest paths between pairs of vertices that run along it. If there is more than one shortest path between a pair of vertices, each path is given equal weight such that the total weight of all of the paths is unity" [1]. The algorithm consists of the following :

1. Calculate the edge betwenness for all edges

2. Remove the edges with the highest edge betweness

3. Recalculate the edge betwenness of all edges affected by the removal.

4. Repeat from step 2 until there are not any edges left.

The time complexity is $\mathcal{O}(|E|^2|V|)$. Indeed, the calculation of the edge betweenness can be done in $\mathcal{O}(|E||V|)$ [2]. Since we eventually have to calculate the egdge betwenness for each edge, the time complexity is $\mathcal{O}(|E|^2|V|)$.

## 2.2 The Louvain method

The Louvain method [3] aims to extract the community structure of a graph $G =< V, E >$ by maximizing the modularity which is defined as :

$$Q = \frac{1}{2m} \sum_{ij} [A_{ij} - \frac{k_i k_j}{2m}] \delta(c_i, c_j)$$

Where $m$ is the number of edges, $A$ is the adjacency matrix, $k_i$ is the degree of the node $i$ , and $c_i$ is the type of the community of the node $c_i$. $\delta(c_i, c_j)$ is set equal to one if $i$ $j$ belong to the same community , and to zero otherwise.

In order to do so, this method proposes the following steps :

1. Assign each node to its own community.

2. For each node $v$, we consider each of its neighbor $k$ and evaluate whether removing $v$ from its community and putting it into $k$'s community would improve the modularity. After considering every neighbor of $v$, we put $v$ into the community of the neighbor that had induced the highest positive gain in modularity. In case of tie, an arbitrary assignment is made. If there is no positive gain, then $v$ remains in its own original community.
   We repeat this step until no more improvement in modularity is possible. We have thus reached a local maximum of the modularity.

3. We now build a new graph where each node is one of the communities previously detected. The edges between those nodes are then assigned a weight, which is the sum of the weight of the edges between the nodes in the corresponding communities. Finally, selfloops are added to each node, which correspond to the edges between the nodes in the community (i.e. the new node).

4. We repeat step 2 and 3 until there is no more improvement in terms of modularity. We have thus reached a maximum of the modularity.

This algorithm is meant to be used with sparse graphs. In this case, the time complexity of the Louvain algorithm is $\mathcal{O}(|E|)$. Indeed [3], running through step 2 the first time takes up most of the total running time. In addition, this is roughly done in $\mathcal{O}(|V|\langle k \rangle)$, where $\langle k \rangle$ is the average number of neighbors per node. According to [4], $\langle k \rangle = \frac{2|E|}{|V|}$, hence the time complexity.
If, in step 2, for each node a random neighbor is considered, instead of every neighbor, then this algorithm runs in $\mathcal{O}(|V|log(\langle k \rangle)$ [4].

In any case, the louvain algorithm works well with large sparse graphs [3] and provides us with a hierarchy of communities at different scales, since it is essentially detecting "communities of communities".

## 2.3 Select the best partition of Girvan Newman

The best partition returned by the Girvan Newman method is the one which maximizes the modularity. To get this partition we have to follow the following procedure :

1. For each partition return by Girvan Newman method (we have at most |E| different partitions) do :

2. Calculate the modularity such that :

$$Q = \frac{1}{2m}\sum_{ij}[A_{ij} - \frac{k_i k_j}{2m}]\delta(c_i, c_j)$$

   Where $m$ is the number of edges, $A$ is the adjacency matrix, $k_i$ is the degree of the node $i$ , and $c_i$ is the type of the community of the node $c_i$. $\delta(c_i, c_j)$ is set equal to one if $i$ $j$ belong to the same community , and to zero otherwise. The modularity represents the proportion of edges that belong to the same community less the number of edges between the group $c_i$ and $c_j$ in a random graph. The time complexity to find the modularity of a given partition is $\mathcal{O}(|V^2|)$, in the view of the number of elements i the sum.

3. Find the best modularity : - We could sort the modularity list and find the maximal value. Using a quick sort algorithm it could be done with a time complexity of $\mathcal{O}(|E|log(|E|))$, since we have at most $|E|$ element in the list.

4. Select the best partition, using indexes.

This procedure have a final cost of O(|E||V|$^2$).

In addition, the Girvan-Newman algorithm is in $\mathcal{O}(|E|^2|V|)$. Therefore, given the size of our biological network, we cannot apply the Girvan-Newman method. That is why we will use **Louvain method**, which is in $\mathcal{O}(|E|)$.

## 2.4 GO composition of each community

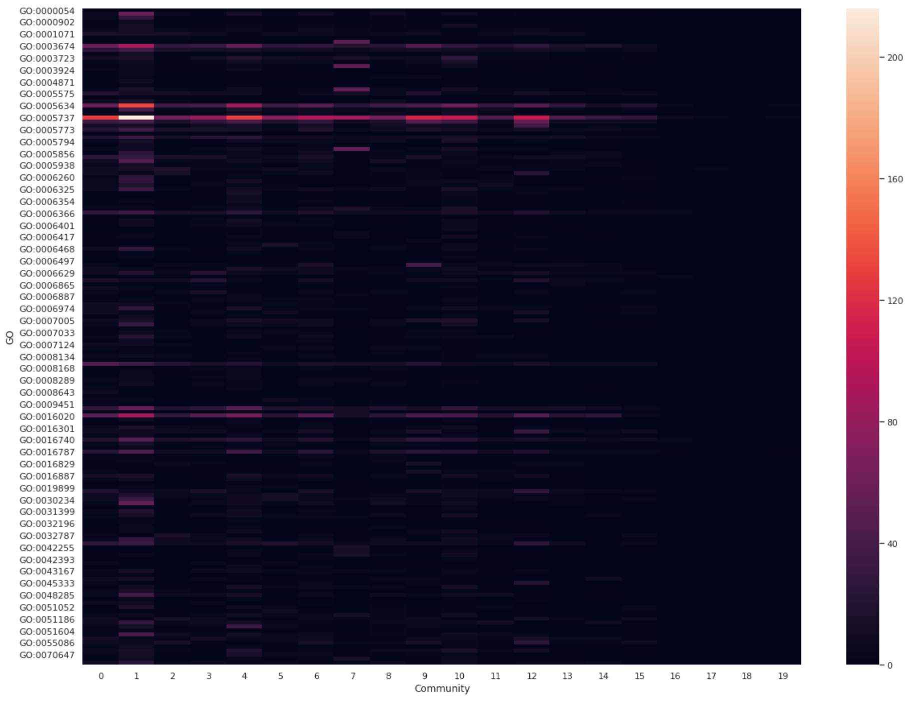


Figure 15: Representation of the contingency matrix, columns are the 20 communities and rows the different gene ontology number.

We can notice in Figure 15 that each community has a different distribution of GO annotations. However, it is quite hard to see which GO annotations discriminate the communities. In addition, we cannot see anything regarding community 16 through 19. This might be because these communities are made up of a small number of genes. Thus, we decided to represent the contingency matrix in terms of proportion of the presence of each GO annotation:

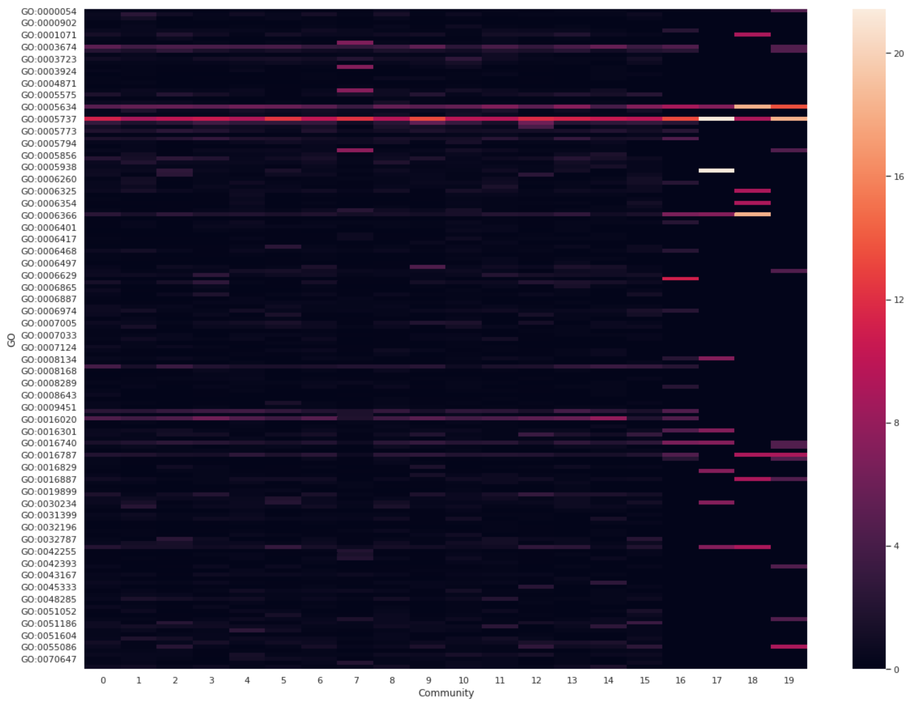


Figure 16: Representation of the contingency matrix in terms of proportion of GO number for each community, columns are the 20 communities and rows the different gene ontology number.

Figure 16 allows us to see the GO makeup of communities 16 through 19. However, the other communities are still hard to read. Nonetheless, we can notice that, in both Figure 15 and 16, GO:0005737 and GO:0005634 are majoritarily present in almost all the communities. This could be because GO:0005737 stands for cytoplasm (i.e. "All of the contents of a cell excluding the plasma membrane and nucleus, but including other subcellular structure") and GO:0005634 for nucleus. These are cellular compartiments where most of the proteins are located for at least part of their "life". Therefore, in order to get a better visual on the fist 15 communities we decided to remove from the contingency matrix the lines corresponding to GO:0005737 and GO:0005634 :

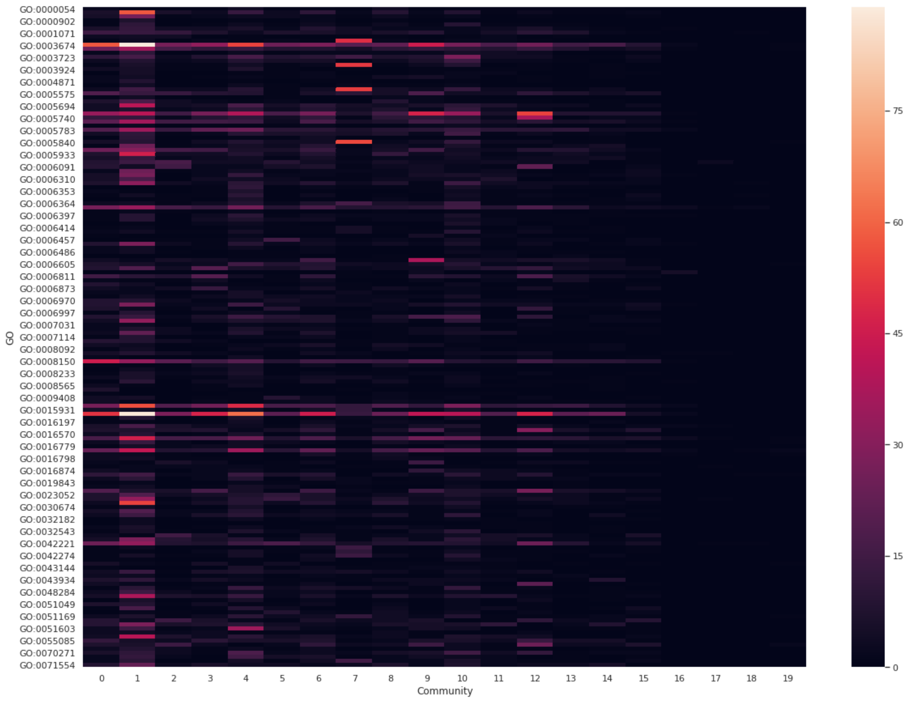


Figure 17: Representation of the contingency matrix , columns are the 20 communities and rows the different gene ontology number. We removed the lines corresponding to GO:0005737 and GO:0005634

We then detailled a few communities with the following website "http://amigo.geneontology.org":

- **Community 1**: It contains 313 genes. It has 89 genes annotated with GO:0003674, which stands for molecular function (i.e. "the cellular chemical reactions and pathways involving a nucleobase-containing small molecule: a nucleobase, a nucleoside, or a nucleotide."). It also has 88 genes annotated with GO:0016020, which stands for membrane, and 58 genes annotated with GO:0000278, which stands for mitotic cell cycle.

- **Community 7**:It contains 112 genes. It has 55 genes annotated with GO:0005840, which stands for ribosome, 53 genes annotated with GO:0005198 (structural molecule activity, which means "the action of a molecule that contributes to the structural integrity of a complex or its assembly within or outside a cell"), 52 genes annotated with GO:0003735 (structural constituent of ribosome, which means "the action of a molecule that contributes to the structural integrity of the ribosome") and 50 genes anotated with GO:0002181 (cytoplasmic translation, which means "the chemical reactions and pathways resulting in the formation of a protein in the cytoplasm. This is a ribosome-mediated process"). This community seems to contain mostly genes related to the function of the ribosome.

- **Community 9**: It contains 130 genes. It has 36 genes that belong to the group G0:0012505, which stands for endomembrane system. This community is also characterized by the gene ontology group named GO:0005739, which stands for semiautonomous, self replicating organelle.

- **Community 17**: According to the matrix [16], the smallest community, which contains 3 genes, is characterized by the GO GO:0006766, which stands for chemical reactions and pathways that involve vitamin. This group is also well represented by the GO GO:0016740 which gathers enzymes that are transferases.

Finally the most relevant information that we could infer from these matrices are :

- The Louvain communities gather genes which belong to the same biological processes.
- Some GO IDs function could be fond in all communities. Indeed these GO IDs represent global functions that could be specified.
- Communities built by the Louvain algorithm are smaller and smaller, that is why reading the most represented GO IDs in the last community we could give us more specific biological processes.
- Finally if we want to characterize each community by a biological process that we want to be the most representative, we would have to remove all GO groups in common between several communities.

## 3 Exercise 4

We want to prove that the following problem is NP-complete:
**INPUT**: Let $G$ be an undirected graph with $n$ vertices.
**QUESTION**: Does there exist a cycle that goes through at least half of the vertices of $G$ ?

In order to do so, we want to show first that this problems is in **NP**:

This problem is a decision problem and its solutions can be "checked" in polynomial time. Indeed, a solution is a cycle of $G$ of size at least $\frac{n}{2}$. So given a solution $C$, we can check $C$'s size with a $\mathcal{O}(1)$ comparison and that $C$ is a cycle of $G$ by verifying that each of its vertices is connected to the next vertex ant that the first and last vertices are the same. This can be done in $\mathcal{O}(n)$ in the worst case (where $C$ has $n$ vertices).

Then, we want to show that there exists a **reduction** from the "Hamiltonian cycle" problem (which is NP-complete) to the problem at hand:

Let $G'$ be an instance of the "Hamiltonian cycle" problem. Let $n'$ be the number of vertices in $G'$. Let $f$ be the function defined such that $f(G') = G$ and $G$ is built such that we have added $n'$ new vertices to $G'$ without connecting them to any vertices (we can notice that this is done in $\mathcal{O}(n')$).
Then: $G'$, a graph with $n'$ vertices, has a Hamiltonian cycle **if and only if** $G$, a graph with $n = 2n'$ vertices, has a cycle that goes through at least half of its vertices. Indeed:

- if $G'$ has a Hamiltonian cycle, then $G'$ has a cycle that goes through at least half of its vertices. Then, by construction, $G$ has a cycle that goes through at least half of its vertices.

- if $G$ has a cycle that goes through at least half of its vertices, then $G$ has a cycle that goes through at least $n'$ vertices. As the $n'$ vertices we added to $G'$ in order to build $G$ were not connected to any vertices of $G'$, then $G$ has a cycle that goes through exactly the $n'$ vertices that constitutes $G'$ in its entirety. Therefore, $G'$ has a hamiltonian cycle.

We have thus built a polynomial reduction from the "Hamiltonian cycle" problem (which is NP-complete) to the problem at hand.

**Thus, the problem is NP-complete**.

## References

[1] M. Girvan and M. E. J. Newman, Community structure in social and biological networks, PNAS June 11, 2002, 99 (12) 7821-7826

[2] M. Girvan and M. E. J. Newman, Finding and evaluating community structure in networks, 2004, PHYSICAL REVIEW E 69, 026113

[3] V. D. Blondel, J-L. Guillaume, R. Lambiotte, E. Lefebvre, Fast unfolding of communities in large networks, 2008, Journal Of Statistical Mechanics-theory And Experiment

[4] V. A. Traag, Faster unfolding of communities: Speeding up the Louvain algorithm, 2015, Phys. Rev. E 92, 032801