



**SORBONNE  
UNIVERSITÉ**

**M1 Informatique – STL  
Rapport Conception Pratique d'Algorithmes**

**Noms, prénoms**

BRAHIMI Lounes

## 1. TME1 Handling a large graph:

### 1. Le nombre de nœuds et d'arêtes :

L'algorithme s'occupant de faire ce calcul dans le projet est nommé « compteur », il opère en lisant le fichier représentant le graph, ligne par ligne en comptant le nombre d'arêtes, il marque les nœuds sources et cibles dans un tableau et calcule entre-temps l'id maximal, à la fin de la lecture du fichier, on parcourt le tableau jusqu'à l'id maximal en comptant le nombre de marquage qui représente le nombre de nœuds.

### Complexité :

L'algorithme parcourt une fois le fichier, ce qui est équivalent au nombre d'arêtes, puis l'algorithme parcourt le tableau selon l'id maximal, conclusion la complexité est linéaire  $O(\text{nombre arêtes})$ .

### Temps D'exécution :

```
Fichier : "com-amazon.ungraph"
Nombre de noeud : 334863
Nombre d'arcs : 925872

Process returned 48 (0x30)   execution time : 0.361 s

Fichier : "com-lj.ungraph"
Nombre de noeud : 3997962
Nombre d'arcs : 34681189

Process returned 51 (0x33)   execution time : 9.564 s

Fichier : "com-orkut.ungraph"
Nombre de noeud : 3072441
Nombre d'arcs : 117185083

Process returned 52 (0x34)   execution time : 52.859 s
```

## 2. Charger un graph en mémoire :

### List of edges :

### Structures utilisées :

Une structure représentant un arc avec un nœud source et un nœud cible :

```
typedef struct {
    unsigned long source;
    unsigned long cible;
} arc;
```

Une structure représentant la liste de tous les arcs :

```
typedef struct {
    unsigned long nombreNoeuds;
    unsigned long nombreArcs;
    arc *arcs;
} arclist;
```

### Algorithme :

La fonction s'occupant de charger le graph sous forme de liste d'arcs est nommée « readarclist » dans le projet, l'algorithme lie le fichier ligne par ligne en créant à chaque fois un arc d'en la source est la valeur à gauche du fichier et la cible la valeur à droite, enfin il stocke cet arc dans la structure représentant l'ensemble de tous les

arcs en calculant le nombre de nœuds et le nombre d'arcs avec l'algorithme vu dans la première section.

### Complexité :

L'algorithme parcourt une fois le fichier, ce qui est équivalent au nombre d'arêtes, donc la complexité est linéaire  $O(\text{nombre arêtes})$ .

### Temps d'exécution :

```
Fichier : com-amazon.ungraph
Nombre de noeuds: 334863
Nombre d'arcs : 925872
Exemple arc 10 : 2 ==> 170507

Process returned 0 (0x0)   execution time : 0.327 s

Fichier : com-lj.ungraph
Nombre de noeuds: 3997962
Nombre d'arcs : 34681189
Exemple arc 10 : 0 ==> 2947898

Process returned 0 (0x0)   execution time : 9.732 s

Fichier : com-orkut.ungraph
Nombre de noeuds: 3072441
Nombre d'arcs : 117185083
Exemple arc 10 : 1 ==> 12

Process returned 0 (0x0)   execution time : 42.763 s
Press any key to continue.
```

### Adjacency matrix :

#### Structures utilisées :

Une structure contenant la liste des arcs et la matrice qui est de type booléenne, un 1 dans une case signifie l'existence d'un arc entre le nœud à l'indice de la ligne et le nœud à l'indice de la colonne, vu que l'id des nœuds peut dépasser le nombre de nœuds un système d'indexation a été mis en place, c'est pour cela qu'un tableau d'indices est présent dans la structure.

```
typedef struct {
    unsigned long nombreNoeuds;
    unsigned long nombreArcs;
    unsigned long id_max;
    unsigned long *indices;
    arc *arcs;
    bool *mat;
} adjmatrix;
```

#### Algorithme :

Une première fonction « readarclistMat » s'occupe de lire le fichier, de charger les arcs, de compter le nombre de nœuds, le nombre d'arcs, de marquer les indices et de sauvegarder l'id maximum.

La fonction « indexation » affecte un indice unique et séquentiel à chaque nœud du graphe.

Enfin la fonction « mkmatrix » qui parcourt la liste des arcs, vu que le graph est indirect, elle marque la case représentant l'arc « uv » et la case représentant « vu ».

## Complexité :

La fonction « readarclistMat » a une complexité  $O(\text{nombre d'arcs})$ , l'indexation a comme complexité  $O(\text{nombre de nœuds})$  et l'algorithme générant la matrice est en  $O(\text{nombre d'arcs})$ , donc la complexité totale est en  $O(\text{nombre d'arcs})$ .

## Remarque :

La mémoire requise pour stocker une matrice étant très importante (exemple : pour le fichier amazon : nombre de nœuds \* nombre de nœuds = 112133228769) la méthode n'est pas très efficace.

File : matrice	Fichier	Edition
Nombre de noeuds : 7	1	2
Nombre d'arcs : 9	1	3
Construction de la matrice :	1	4
Construction Finit.	1	5
0 1 1 1 0 0	2	4
1 0 0 1 0 1 0	2	9
1 0 0 0 1 0 0	3	5
1 1 0 0 0 1 0	4	9
1 0 1 0 0 0 0	9	10
0 1 0 1 0 0 1		
0 0 0 0 0 1 0		
Process returned 0 (0x0) execution time : 2.463 s		

## Adjacency array :

### Structures utilisées :

Une structure contenant la liste des arcs, un tableau indices pour l'indexation, le tableau « adj » contient les voisins, le tableau « cd » permet de délimiter les indices des voisins d'un nœud au sein du tableau « adj », on peut aussi l'utiliser pour obtenir le degré d'un nœud.

```
typedef struct {
    unsigned long nombreNoeuds;
    unsigned long nombreArcs;
    arc *arcs;
    unsigned long id_max;
    unsigned long *indices;
    unsigned long *cd;
    unsigned long *adj;
} adjlist;
```

## Algorithme :

Une première fonction « readedgelist » s'occupe de lire le fichier, de charger les arcs, de compter le nombre de nœuds, le nombre d'arcs et de sauvegarder l'id maximum.

L'algorithme permettant le stockage dans une liste d'adjacence est implémenté dans la fonction « mkadjlist », l'algorithme commence par mettre en place l'indexation car l'id des nœuds peut dépasser le nombre total de nœuds présent dans le graph, en même temps il génère un tableau représentant le degré de chaque nœud, ce dernier est utilisé pour calculer le vecteur « cd » représentant dans la structure, enfin les voisins d'un nœud « u » sont stockés dans le vecteur « adj », l'indice dans lequel ils sont stockés est calculer en utilisant le cd de ce nœud-là et de son degré qui s'incrémente à chaque ajout.

## Complexité :

La fonction « readedgelist » a une complexité  $O(\text{nombre d'arcs})$  de même que l'algorithme générant la liste d'adjacence, conclusion la complexité est en  $O(\text{nombre d'arcs})$ .

## Temps d'exécution :

```
Fichier : com-amazon.ungraph
Nombre de noeuds : 334863
Nombre d'arcs : 925872
Construction de la liste d'adjacence

Process returned 0 (0x0)   execution time : 0.544 s
```

```
Fichier : com-lj.ungraph
Nombre de noeuds : 3997962
Nombre d'arcs : 34681189
Construction de la liste d'adjacence

Process returned 0 (0x0)   execution time : 12.239 s
```

```
Fichier : com-orkut.ungraph
Nombre de noeuds : 3072441
Nombre d'arcs : 117185083
Construction de la liste d'adjacence

Process returned 0 (0x0)   execution time : 46.675 s
```

## Conclusion :

Les 3 stockages sont linéaires en temps, mais la matrice d'adjacence est très couteuse en espace, ce qui n'est pas le cas de la liste d'adjacence qui est tout aussi rapide que la liste d'arcs avec beaucoup plus de données (exemple les voisins d'un nœud sont accessibles directement) ce qui permet plus de manipulation sur le graph.

### 3. Breadth-first search and diameter:

#### Remarque:

Une implémentation d'une « fifo » a été mise en œuvre, elle permet d'enfiler et de défiler, elle est contenue dans la structure suivante :

```
typedef struct {  
    unsigned long nombre_elements;  
    unsigned long tete;  
    unsigned long queue;  
    unsigned long taille;  
    unsigned long* u;  
} fifo;
```

#### Lower-bound :

#### Structure utilisée :

La structure est composée d'un champ « nœud » faisant référence au nœud le plus loin rechercher et de « dis » qui indique la distance de ce dernier de du nœud d'origine.

```
typedef struct {  
    unsigned long noeud;  
    int dis;  
} resultat_bfs;
```

#### Algorithme :

Pour stocker le graphe c'est la méthode de liste d'adjacence qui a été choisie et cela est dû au fait qu'elle nous permet d'accéder facilement aux voisins des nœuds.

C'est la fonction « lower\_bfs » qui héberge dans le projet l'algorithme s'occupant de calculer la lower-bound, cette dernière prend en entrée la structure graph générée à l'étape précédente et un nœud source, le but de l'algorithme est de calculer le nœud avec la plus longue distance de ce nœud source, L'algorithme parcourt donc tous les nœuds du graphe en calculant successivement leurs distances du nœud source, à la fin de cette étape un tableau représentant ces distances est généré, enfin on parcourt ce dernier tableau en cherchant la distance la plus élevée, une fois trouver elle est retournée avec l'indice du nœud, l'algorithme peut être itéré sur le résultat un nombre « x » de fois pour avoir le meilleur résultat.

#### Complexité :

La lecture du fichier et sa sauvegarde a une complexité  $O(\text{nombre d'arcs})$ .

La première étape du « bfs » (le calcul des distances) ce fait en maximum  $O(\text{nombre de nœuds})$  et la recherche de la plus grande valeur dans le résultat obtenu ce fait en  $O(\text{id maximum d'un nœud})$ , vu que le nombre de nœud est inférieur au nombre d'arcs alors la complexité est en  $O(\text{nombre d'arcs})$  donc linéaire.

## Temps d'exécution :

```
Fichier : com-amazon.ungraph
Nombre de noeuds : 334863
Nombre d'arcs : 925872

distance ==> 47

Process returned 0 (0x0)   execution time : 1.471 s

Fichier : com-lj.ungraph
Nombre de noeuds : 3997962
Nombre d'arcs : 34681189

distance ==> 21

Process returned 0 (0x0)   execution time : 32.625 s

Fichier : com-orkut.ungraph
Nombre de noeuds : 3072441
Nombre d'arcs : 117185083

distance ==> 10

Process returned 0 (0x0)   execution time : 113.477 s
```

## Upper-bound :

La même structure que le lower-bound est utilisée.

## Algorithme :

Tout comme la lower-bound pour stocker le graphe c'est la méthode de liste d'adjacence qui a été choisie.

C'est la fonction « upper\_bfs » qui héberge dans le projet l'algorithme s'occupant de calculer la upper-bound, cette dernière prend en entrée la structure graph générée à l'étape précédente et un nœud source, le but de l'algorithme est de calculer dans un premier temps le nœud au milieu du chemin entre le nœud source et le nœud le plus loin de ce dernier, ensuite il trouve le nœud le plus loin du milieu et enfin trouve le sommet le plus loin de ce dernier. L'algorithme parcourt donc tous les nœuds du graphe en calculant successivement leurs distances du nœud source, à la fin de cette étape un tableau représentant ces distances est généré, enfin on parcourt ce dernier tableau en cherchant la distance la plus élevée, puis le sommet au milieu est détecté, le sommet le plus loin de ce nœud est calculé, enfin on calcule le nœud à la distance la plus élevée de ce sommet et on retourne le résultat.

## Complexité :

La lecture du fichier et sa sauvegarde a une complexité  $O(\text{nombre d'arcs})$ .

Le calcul des distances et du milieu se fait en  $O(\text{nombre d'arcs})$ , de même que la distance la plus longue d'un nœud comme vu dans l'algorithme précédent, la complexité est donc en  $O(\text{nombre d'arcs})$ .

## Temps d'exécution :

```
Fichier : com-amazon.ungraph
Nombre de noeuds : 334863
Nombre d'arcs : 925872

Distance milieu: 16
Distance ==> 47

Process returned 0 (0x0)   execution time : 0.683 s

Fichier : com-lj.ungraph
Nombre de noeuds : 3997962
Nombre d'arcs : 34681189

Distance milieu: 6
Distance ==> 21

Process returned 0 (0x0)   execution time : 18.872 s

Fichier : com-orkut.ungraph
Nombre de noeuds : 3072441
Nombre d'arcs : 117185083

Distance milieu: 3
Distance ==> 9

Process returned 0 (0x0)   execution time : 73.057 s
```

## 4. Listing triangles :

### Algorithme :

Pour stocker le graphe c'est la méthode de liste d'adjacence qui a été choisie, c'est la fonction « liste\_des\_arcs » qui s'occupe de lire le fichier, de charger les arcs de façon à ce que l'id du nœud source soit toujours inférieur à l'id du nœud cible (l'idée est de travailler sur un graphe orienté).

La fonction « adjlist\_dag » permet de gérer l'indexation du graphe orienté, l'algorithme génère un tableau des degrés sortants pour chaque nœud du graphe et stocke les voisins de chaque nœud dans « adj ».

L'algorithme qui calcule le nombre de triangles est implémenté dans la fonction « nombre\_triangle », il parcourt tous les arcs du graph et pour chaque nœud source il parcourt ses voisins du plus grand au plus petit en s'intéressant qu'à ceux qui ont un id supérieur à celui de sa cible, il parcourt ainsi les voisins de la cible et test si les deux nœuds de l'arc ont un voisin en commun, si c'est le cas le nombre de triangles est incrémenté.

### Complexité :

La lecture et la sauvegarde du graph dans la structure de liste d'adjacence ce fait en  $O(\text{nombre d'arcs})$ .



L'algorithme calculant le nombre de triangles, parcourt tous les arcs du graph (n), ainsi que tous les voisins du nœud source supérieur à la cible (du+), enfin il parcourt tous les nœuds cibles avec un id supérieur au voisin courant du nœud cible (dv+).

La complexité est donc en  $O(n \cdot du+ \cdot dv+)$ .

## Temps d'exécution :

```
Fichier : com-amazon.ungraph
Nombre de noeud : 334863
Nombre d'arcs : 925872

nombre de triangles ==> 667129

Process returned 0 (0x0)   execution time : 0.624 s
```

## 2. TME2 PageRank:

### PageRank (directed graph):

#### Structures utilisées :

Une structure pour l'arc.

Une structure modélisant entre autres la liste d'adjacence, la liste de tous les arcs, un tableau indices pour l'indexation, une liste des degrés sortants « d\_out » et de degrés entrants « d\_in » vu que c'est un graph orienté, enfin un array « P » représentant le page rank.

```
typedef struct {
    unsigned long source;
    unsigned long cible;
} arc;

typedef struct {
    unsigned long nombreNoeuds;
    unsigned long nombreArcs;
    arc *arcs;
    unsigned long id_max;
    unsigned long *indices;
    unsigned long *cd;
    unsigned long *adj;
    int *d_out;
    int *d_in;
    double* P;
} graph;
```

#### Algorithme :

Pour stocker le graphe c'est la méthode de liste d'adjacence qui a été choisie, c'est la fonction « liste\_arcs » qui s'occupe de lire le fichier, de charger les arcs, de constituer le tableau des degrés entrants et sortants, ainsi que le marquage des indices.

La fonction « indexation » comme son nom l'indique gère les indices des nœuds vus que l'id d'un nœud peut dépasser le nombre total de nœuds.

La fonction qui calcule le page rank est nommée « power\_iteration » dans le projet, elle prend en entrant la structure graphe chargé, un flottant représentant la probabilité qu'un marcheur se téléporte de façon aléatoire dans le graph et le nombre d'itérations de l'algorithme pour atteindre une convergence, l'algorithme à l'aide de la fonction « initialisation\_P » comme son nom l'indique initialise à 0 le page rank et à  $1/n$  le vecteur « P1 », l'algorithme parcourt tous les arcs (uv) en calculant pour chaque nœud cible  $P[v] : P1[u]/d_{out}(y)$ , ensuite l'algorithme parcourt tous les nœuds du graphs et calcul «  $P \leftarrow (1 - \alpha) \times P + \alpha \times I$  » et finit par normaliser « P », pour l'itération suivante la fonction « permutationVecteurs » est utilisée pour affecter à « P1 » le « P » résultant et en remettant à 0 le « P » pour le prochain calcul.

## Complexité :

La lecture et la sauvegarde du graph dans la structure ce fait en  $O(\text{nombre d'arcs})$ .

La fonction d'indexation elle en  $O(\text{nombre de nœuds})$ .

L'algorithme « power\_iteration », parcourt tous les arcs, puis parcourt tous les nœuds de façon successive et tout cela « t » fois, « t » étant très petit il est négligé, donc l'algorithme a une complexité en  $O(\text{nombre d'arcs})$ .

## Temps exécution :

```
Fichier : alr21--dirLinks--enwiki-20071018
Nombre de noeuds : 2070486
Nombre d'arcs : 46092177
====Highest PageRank====

United States
PR : 0.003732
United Kingdom
PR : 0.001619
2006
PR : 0.001404
Erina High School
PR : 0.001403
Germany
PR : 0.001399

====Lowest PageRank====

Aberdeen (disambiguation)
PR : 7.311536e-008
Animal (disambiguation)
PR : 7.311536e-008
Antigua and Barbuda
PR : 7.311536e-008
Batak Pony
PR : 7.311536e-008
Demographics of American Samoa
PR : 7.311536e-008

Process returned 0 (0x0)   execution time : 34.925 s
```

## Expérimentation de la convergence :

Pour  $t = 1$

```
United States
PR : 0.005932
Germany
PR : 0.002601
England
PR : 0.002433
Erina High School
PR : 0.002425
Canada
PR : 0.002135
```

$t=5$

```
United States
PR : 0.003732
United Kingdom
PR : 0.001619
2006
PR : 0.001404
Erina High School
PR : 0.001403
Germany
PR : 0.001399
```

$t=10$

```
United States
PR : 0.003649
United Kingdom
PR : 0.001596
2007
PR : 0.001366
Sound Affects
PR : 0.001366
2006
PR : 0.001362
```

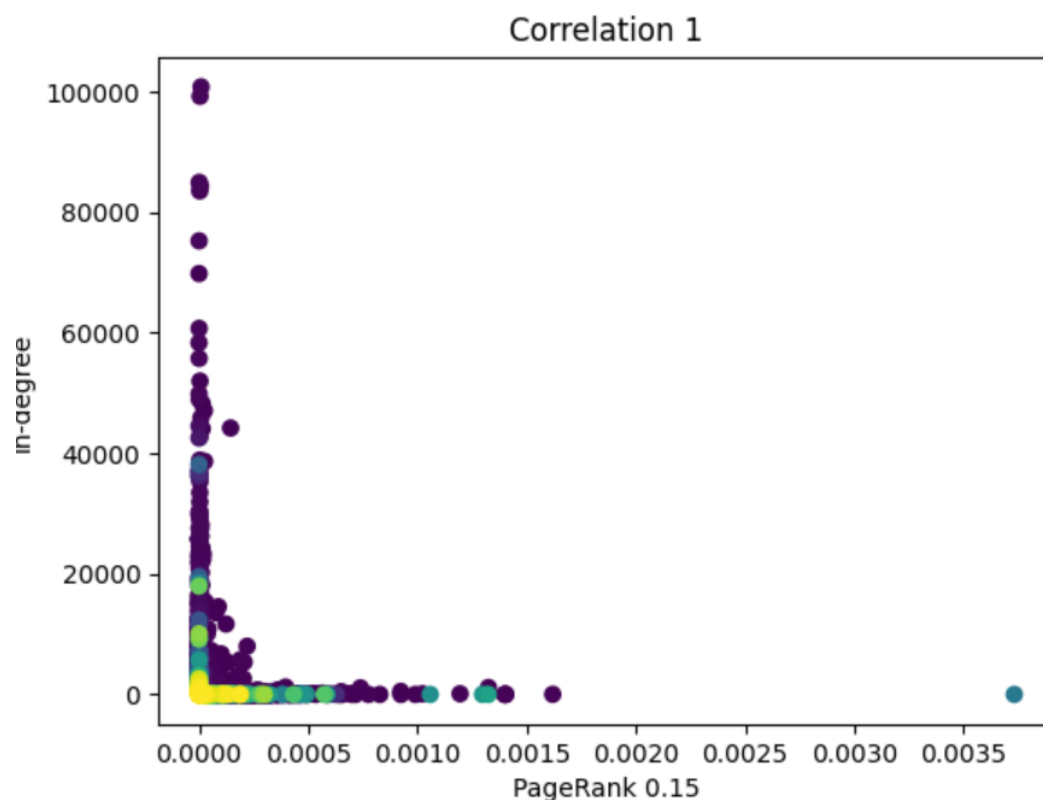
$t=50$

```
United States
PR : 0.003634
United Kingdom
PR : 0.001591
Germany
PR : 0.001363
Erina High School
PR : 0.001360
2006
PR : 0.001356
```

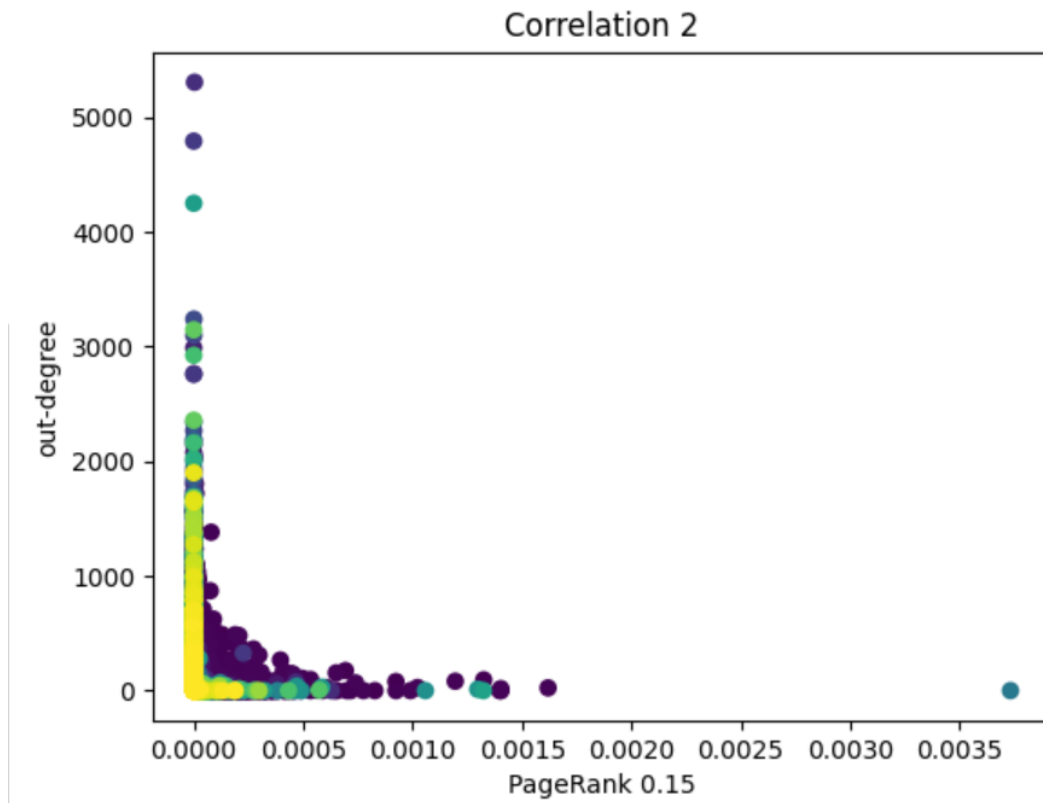
On peut conclure qu'à partir de 5 itération la convergence est visible.

## Correlations:

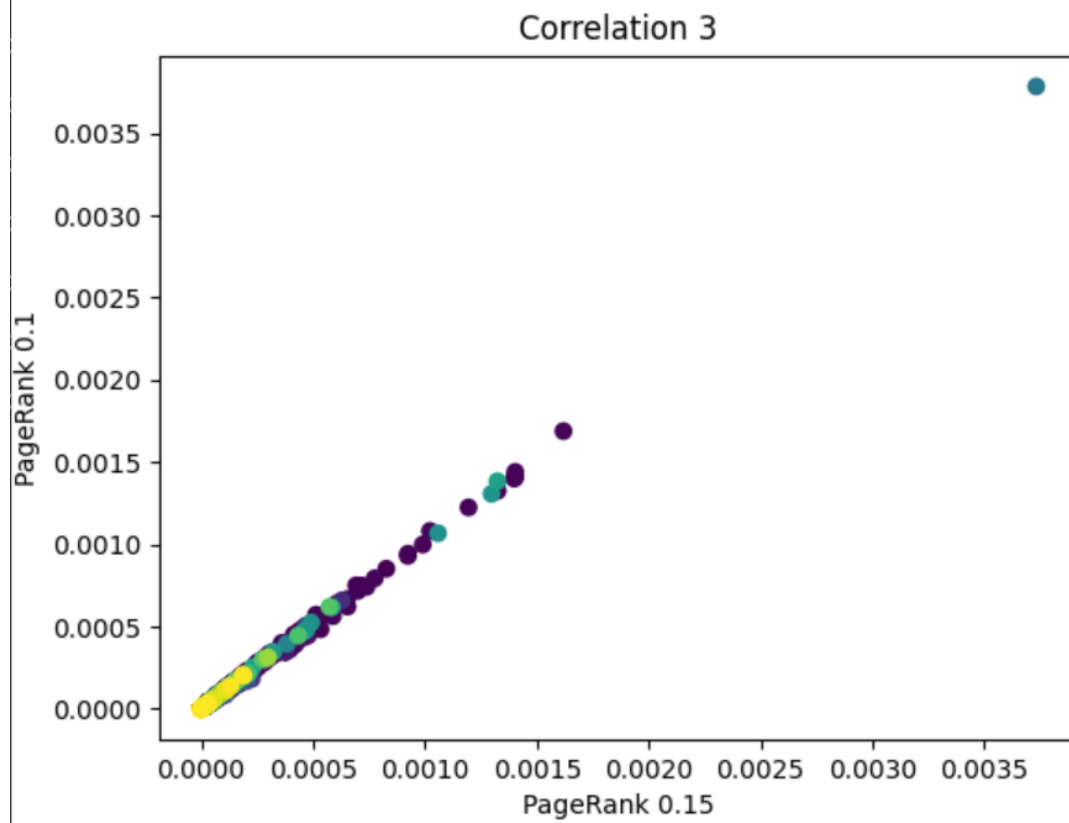
1.  $x$  = PageRank with  $\alpha = 0.15$ ,  $y$  = in-degree:



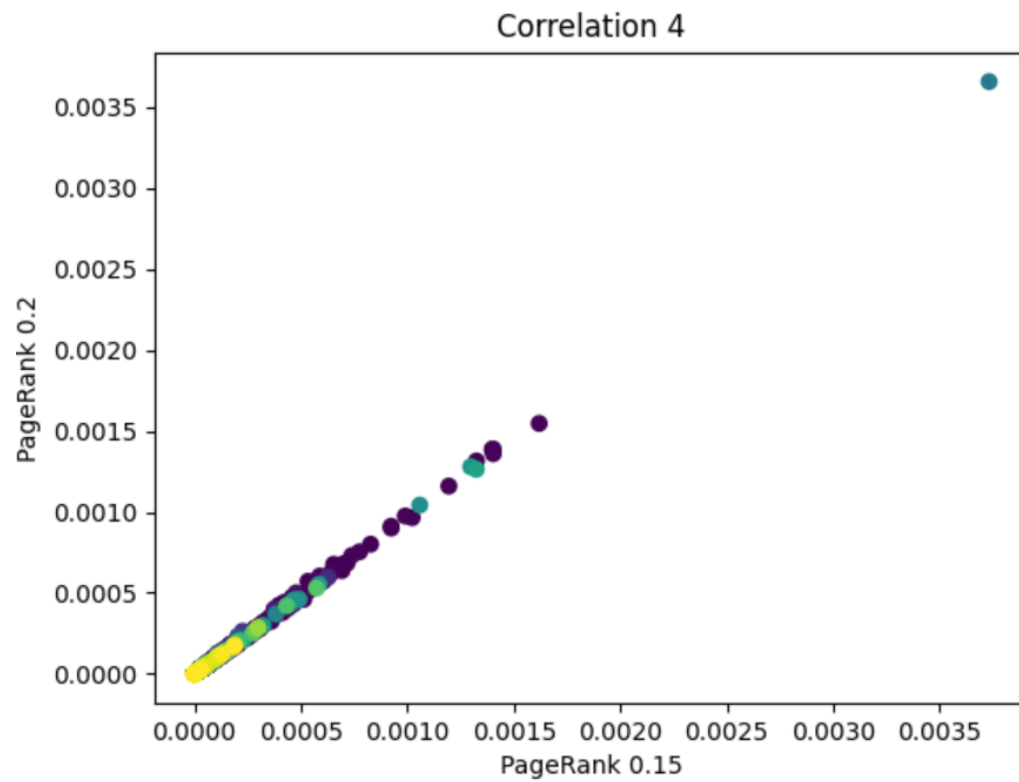
2.  $x$  = PageRank with  $\alpha = 0.15$ ,  $y$  = out-degree :



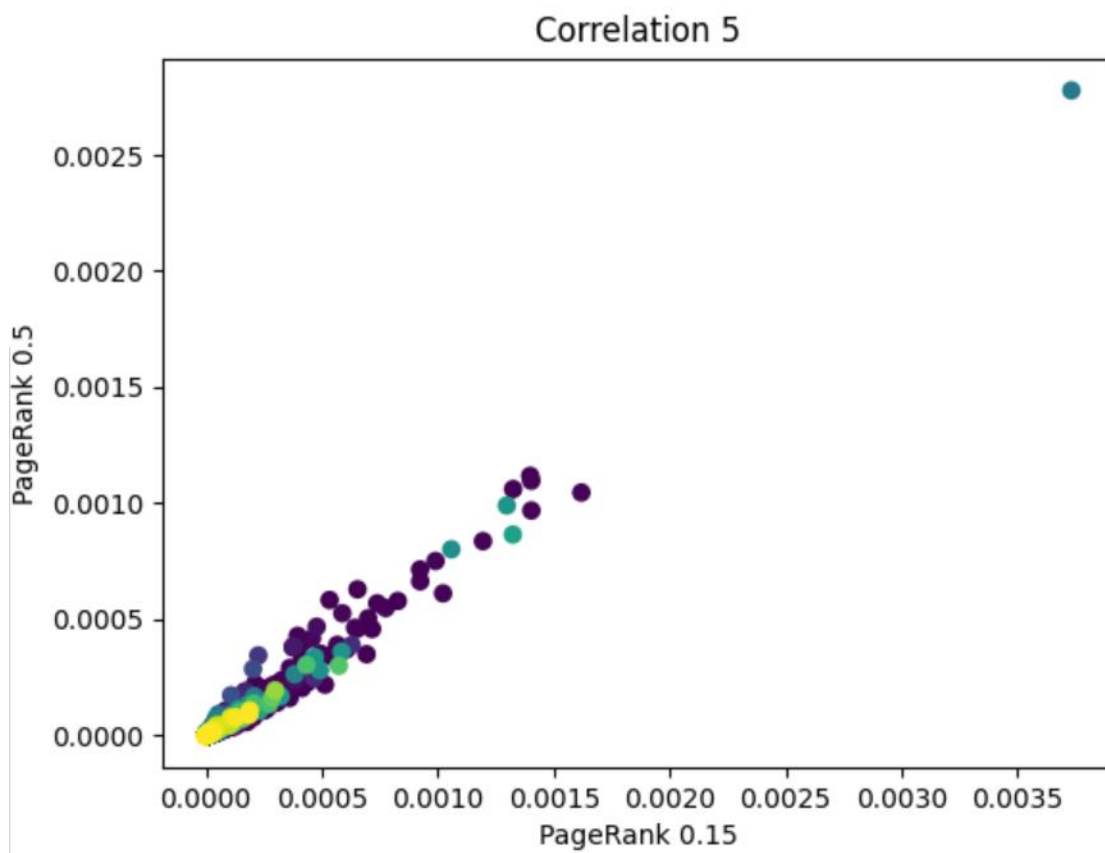
3.  $x$  = PageRank with  $\alpha = 0.15$ ,  $y$  = PageRank with  $\alpha = 0.1$ :



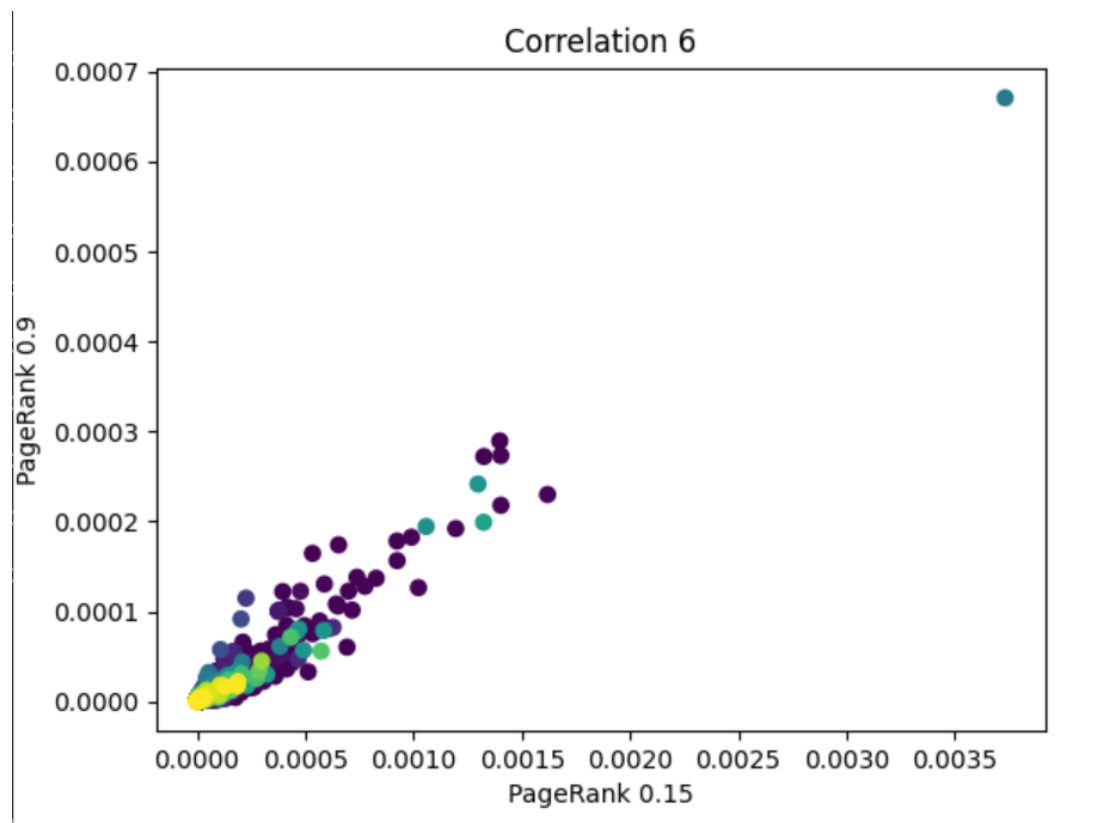
4.  $x = \text{PageRank with } \alpha = 0.15$ ,  $y = \text{PageRank with } \alpha = 0.2$ :



5.  $x = \text{PageRank with } \alpha = 0.15$ ,  $y = \text{PageRank with } \alpha = 0.5$ :



6.  $x = \text{PageRank with } \alpha = 0.15$ ,  $y = \text{PageRank with } \alpha = 0.9$ :



J'ai utilisé une échelle logarithmique car le nombre de points est très important et les valeurs très petites.

Les deux premières corrélations (degrés entrants et sortants avec page rank) sont quasi équivalentes.

Pour la corrélation avec  $x$  et  $y$  des Page Ranks, on remarque bien que quand «  $\alpha$  » de «  $y$  » a une valeur minimale (0.1) ou (0.2) (proche de 0.15) le coefficient de corrélation est quasiment égal à 1, par contre dé que «  $\alpha$  » devient grand (0.5) ou (0.9) on perd cette dépendance, cela est dû au fait que le marcheur se téléporte beaucoup plus fréquemment de façon aléatoire, et de ce fait on a perdu la convergence qu'on avait aux premiers.

## Personalized PageRank :

### Algorithme :

L'algorithme est implémenté dans la fonction « `power_iteration_rooted` » très similaire à « `power_iteration` » déjà vu à la différence qu'un vecteur est passé en argument, il est utilisé pour initialiser « `P1` », si on mettait à 1 ou une valeur importante à l'indice d'un nœud cela fait que le page rank de ce nœud la et des nœuds qui ont le même contexte (donc beaucoup de relations) auront les plus grands page rank et seront donc les nœuds les plus pertinents.

## Complexité :

La lecture et la sauvegarde du graph dans la structure ce fait en  $O(\text{nombre d'arcs})$ .


La fonction d'indexation elle en  $O(\text{nombre de nœuds})$ .

L'algorithme « power\_iteration\_rooted », parcourt tous les arcs, puis parcourt tous les nœuds de façon successive et tout cela « t » fois, « t » étant très petit il est négligé, donc l'algorithme a une complexité en  $O(\text{nombre d'arcs})$ .

## Temps d'exécution :

```
Nombre d'arcs : 46092177
====Highest PageRank====

Magnus Carlsen
PR : 0.150000
Amsterdam
PR : 0.010625
August
PR : 0.010625
Wertheinstein
PR : 0.010625
Anatoly Karpov
PR : 0.010625

Anatoli Karpov
Joueur d'échecs


====Lowest PageRank====

Anarchism
PR : 0.000000e+000
Autism
PR : 0.000000e+000
Albedo
PR : 0.000000e+000
Thomas McCunn
PR : 0.000000e+000
A
PR : 0.000000e+000

Process returned 0 (0x0)   execution time : 22.155 s
```

## TME3 Community detection:

### Simple bechmark:

### Algorithme :

L'algorithme est implémenté dans la fonction « simple\_bechmark » qui prend en argument un flottant « p » représentant la probabilité que des nœuds d'un même cluster soient connectés entre eux et un autre flottant « q » qui représente la probabilité que des nœuds de clusters différents soient lié, dans le cas de notre étude pour générer des communautés « q » doit être très petit devant « p », l'algorithme stocke les 400 id de

nœuds dans un tableau, l'algorithme travaille sur 4 intervalles ([0..100], [100..200], [200..300], [300..400]), il parcourt tous les nœuds avec une probabilité « p » un arc avec le nœud courant comme source est créé directement dans le fichier avec un autre nœud du même intervalle, de la même façon mais avec une probabilité « q » le nœud est lié avec un autre d'un intervalle différent, sachant que le nœud n'est comparé qu'avec des nœuds d'un degré plus important.

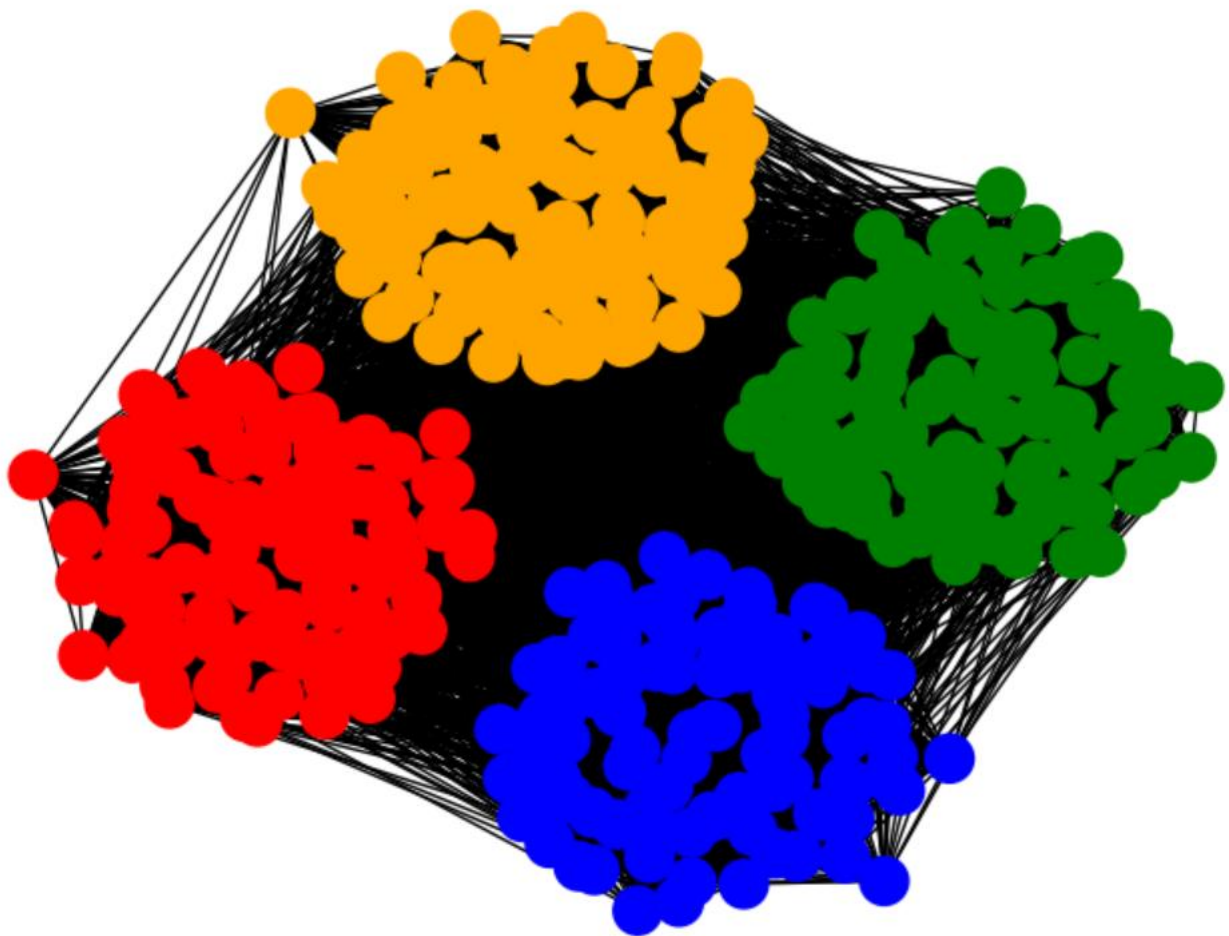
### Complexité :

L'algorithme parcourt tous les nœuds et les compare avec les nœuds d'un plus grand id, donc la complexité est en  $O(\text{Nombre de nœuds})$ .

### Temps d'exécution :

```
p = 0.50, q = 0.03
```

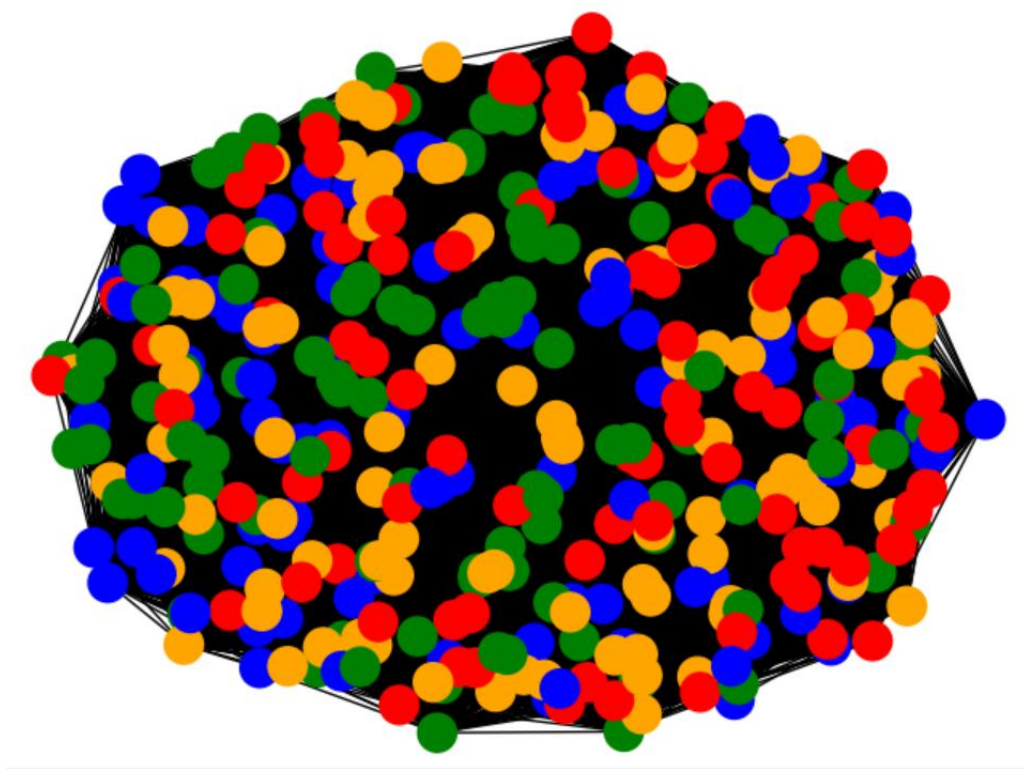
```
Process returned 0 (0x0)   execution time : 0.161 s
```





Augmenter la valeur de  $p$  a pour effet de rassembler les communautés ce qui créer des clusters séparés dans le visuel, augmenter la valeur de  $q$  rapproche les nœuds de différents clusters, ce qui mélange un peu les communautés.

Exemple contraire au dernier graph  $p = 0.20, q = 0.50$  :



## Label propagation :

### Structures utilisées :

Une structure pour l'arc.

Une structure modélisant entre autres la liste d'adjacence, la liste de tous les arcs, un tableau indices pour l'indexation, un tableau

« id\_nodes » stockant les identifiants affectés aux nœuds, un autre « nodes » qui permet d'avoir l'identifiant original d'un nœud à partir d'un indice, « voisins » enregistre le nombre de voisins de chaque nœud id,

« marquage\_noeuds » est utilisé dans le prochain exercice.

```
typedef struct {
    unsigned long source;
    unsigned long cible;
} arc;

typedef struct {
    unsigned long nombreNoeuds;
    unsigned long nombreArcs;
    arc *arcs;
    unsigned long id_max;
    unsigned long *indices;
    unsigned long *id_node;
    unsigned long *nodes;
    unsigned long *cd;
    unsigned long *adj;
    unsigned long *voisins;
    int *marquage_noeuds;
} graph;
```

## Algorithmes :

Pour stocker le graphe c'est la méthode de liste d'adjacence qui a été choisie, cela se fait à travers la fonction « readedgelist », qui lie le fichier et stocke entre autres les arcs.

La fonction « mkadlist » est utilisé pour charger la liste d'adjacence (« adj », « cd » ...), elle s'occupe aussi de gérer l'indexation, initialiser les identifiants des nœuds et sauvegarder les nœuds (ils seront utilisés plus tard pour permettre l'indexation inverse).

La fonction indexation permet de transformer le tableau « nodes » de sorte à ce qu'on puisse remonter d'un indice vers un nœud spécifique.

Enfin, l'algorithme label de propagation est implémenté dans la fonction « label\_propagation », il commence par trier les nœuds de façon aléatoire et pour ça la fonction « shuffle\_nodes » a été implémentée, ensuite il parcourt tous les nœuds avec cet ordre aléatoire et lui affecte l'id de ses voisins le plus fréquent, le processus est répété tant qu'il existe un nœud qui n'a pas comment identifiant le plus fréquent de ces voisins.

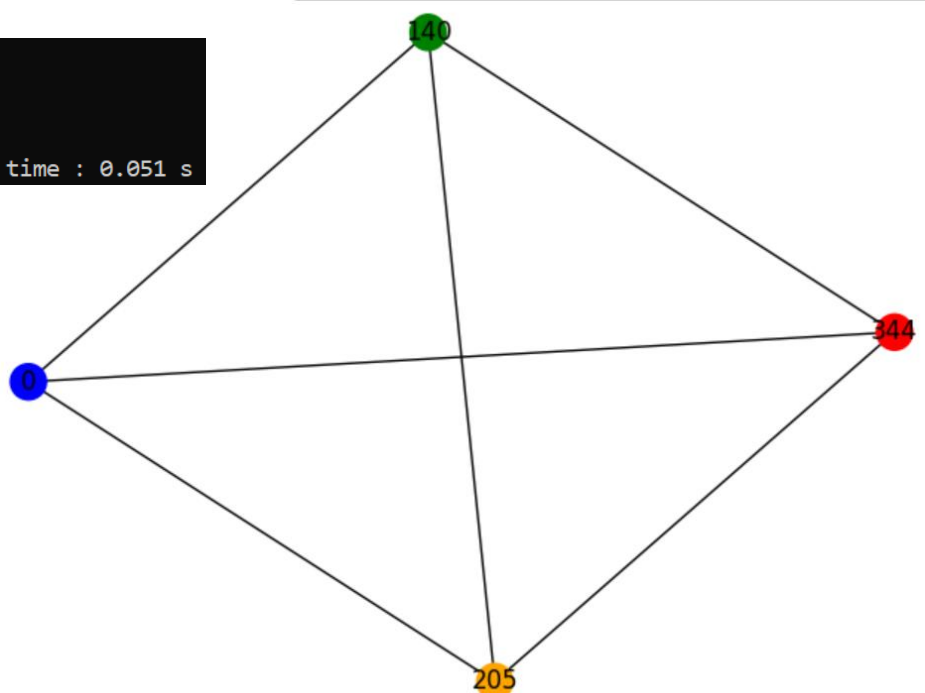
## Complexité :

L'indexation ainsi que la lecture et la sauvegarde du graph dans la structure ce fait en  $O(\text{nombre d'arcs})$ .

La méthode de tri aléatoire utiliser ce fait en temps linéaire  $O(n)$ , ensuite il parcourt tous les nœuds et pour chaque nœud il vérifie l'id de chacun de ses voisins ce qui est équivalent à  $O(2 * \text{nombre d'arcs})$ , l'algorithme itérera un nombre « t » de fois, comme dit au cours, en pratique il a bien été constaté que « t » est très petit, donc la complexité est en  $O(t * \text{Nombre d'arcs})$ .

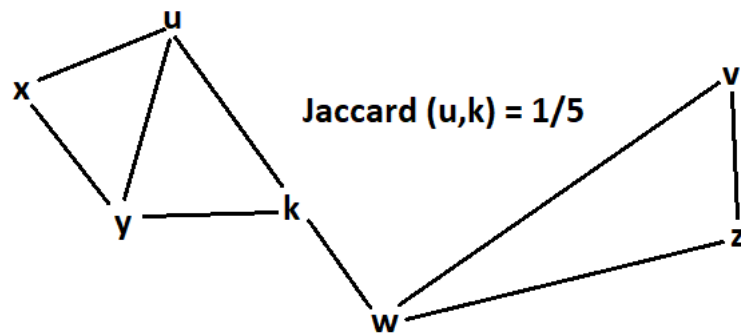
## Temps exécution :

```
Fichier 4clusters  
Nombre de noeuds : 400  
Nombre d'arcs : 13371  
Process returned 0 (0x0)   execution time : 0.051 s
```

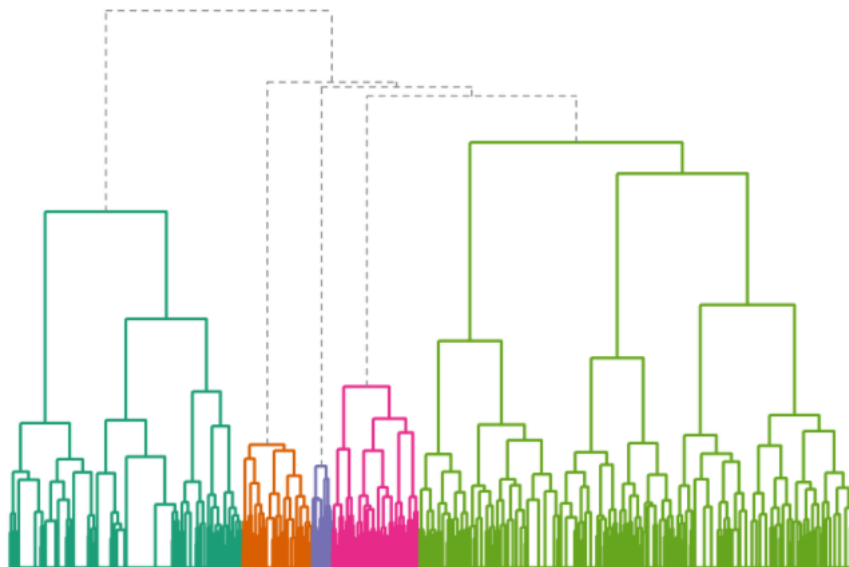


## New algorithm :

Pour cet exercice j'ai fait une implémentation d'une méthode déjà existante, la méthode en question est « la classification ascendante hiérarchique », le but est de donner une valeur pour chaque arc du graph, les arcs avec les plus petites valeurs sont des arcs liants deux clusters, j'ai utilisé pour calculer les valeurs « l'indice de Jaccard » :  $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$ , ainsi je divise le cardinal des voisins en communs entre A et B sur le cardinal de l'union de leurs voisins, plus le résultat est grand plus l'arc a de forte chances d'être dans un cluster et l'inverse est vraie.



Une fois cela fait, on supprime les arcs qui ont une valeur inférieure à un certain x, cela nous permet de détecter des regroupements de nœuds et donc des communautés (exemple on supprime  $\leq 1/5$  pour l'exemple ci-dessus).



## Structure utilisée :

Cette structure permet de stocker les nœuds qui seront sélectionnés dans une étape de l'algorithme.

```
typedef struct {  
    unsigned long source;  
    unsigned long cible;  
} arc_suppr;
```

## Algorithmes :

Pour stocker le graphe c'est la méthode de liste d'adjacence qui a été choisie, cela se fait à travers la fonction « readedgelist », qui lie le fichier et stocke entre autres les arcs.

La fonction « mkadjlist » est utilisée pour charger la liste d'adjacence (« adj », « cd » ...), elle s'occupe aussi de gérer l'indexation, initialiser les identifiants des nœuds et sauvegarder les nœuds (ils seront utilisés plus tard pour permettre l'indexation inverse).

La fonction indexation permet transformer le tableau « nodes » de sorte à ce qu'on puisse remonter d'un indice vers un nœud spécifique.

L'algorithme est implémenté dans la fonction nommée « hiear\_detect », qui prend en argument la structure graph et la valeur flottante à partir de laquelle le découpage sera fait, l'algorithme commence par parcourir tous les arcs du graph un par un, calcul l'union (la somme du nombre de voisins des 2 nœuds auquel on soustrait l'union, ainsi que l'union en parcourant tous les voisins des nœuds et en incrémentant un compteur quand ils ont un sommet en commun), une fois que la valeur de l'arc courant est calculée elle est comparé à celle passée en argument pour vérifier si l'arc doit être supprimé, si c'est le cas il est ajouté dans un vecteur le gardant pour l'étape suivante, ensuite l'algorithme utilise la fonction « bsf » en lui passant le tableau d'arcs à supprimer et un nœud de départ pour affecter un identifiant pour tout nœud du graph, elle leur affecte un identifiant initiale 0, elle donne le même identifiant à tout le monde jusqu'à arriver un arc à supprimer elle donne le même identifiant au sommet source, mais un nouvel identifiant pour le sommet cible de cette façon la première communauté tous ces nœuds auront comme identifiant 0 et l'autre communauté ils auront 1 comme identifiant.

## Complexité :

L'indexation ainsi que la lecture et la sauvegarde du graph dans la structure ce fait en  $O(\text{Nombre d'arcs})$ .

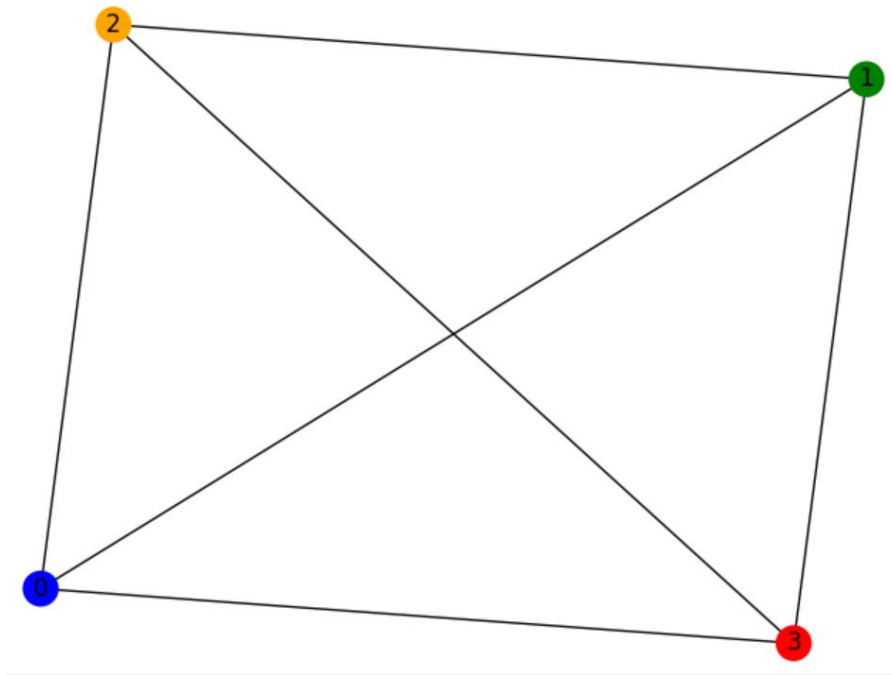
L'algorithme principal parcourt tous les arcs du graph et pour chaque sommet il visite ses voisins donc sa complexité est en  $O(\text{Nombre d'arcs} * \text{Nombre d'arcs})$ .

Le « BFS » parcourt tous les nœuds du graph et pour chacun d'entre eux il visite tous ses voisins donc comme l'algorithme principal la complexité est en  $O(\text{Nombre d'arcs} * \text{Nombre d'arcs})$ .

Conclusion la complexité est quadratique  $O(n^2)$ .

## Temps d'exécution :

```
Fichier 4clusters
Nombre de noeuds : 400
Nombre d'arcs : 13371
====Arcs supprimés : ====
 98 ==> 1070
199 ==> 2031
299 ==> 3182
Process returned 0 (0x0)   execution time : 0.572 s
```



## Experimental evaluation :

### TME4 Densest subgraph:

### k-core decomposition:

### Structures utilisées:

Une structure min heap a été implémentée avec toutes les fonctions permettant de la manipuler :

```
typedef struct{
    int d;
    int c;
    int indice;
    int id_node;
}id_degree;

typedef struct{
    id_degree *arr;
    int count_;
    int capacity;
    int* indexation;
}min_heap;
```

Une structure pour l'arc.

Une structure modélisant entre autres la liste d'adjacence et le degré d'un nœud.

```
typedef struct {
    unsigned long source;
    unsigned long cible;
} arc;

typedef struct {
    unsigned long nombreNoeuds;
    unsigned long nombreArcs;
    arc *arcs;
    unsigned long id_max;
    unsigned long *indices;
    unsigned long *cd;
    unsigned long *adj;
    int *d;
} graph;
```

## Algorithmes :

Pour stocker le graphe c'est la méthode de liste d'adjacence qui a été choisie, cela se fait à travers la fonction « liste\_arcs », qui lie le fichier et stocke entre autres les arcs.

La fonction « mkadjlist » est utilisée pour charger la liste d'adjacence (« adj », « cd » ...), elle s'occupe aussi de gérer l'indexation et de calculer les degrés des nœuds.

L'algorithme est implémenté dans la fonction « core\_decomposition », il commence par insérer les nœuds dans le tas-min, le critère du tri choisi est l'importance du degré d'un nœud, l'algorithme traite tous les nœuds du graph en commençant par le minimum du tas à chaque fois, et pour chaque sommet obtenu il calcule son core-value en comparant le core-value actuel avec le degré du nœud et choisit le plus grand et lui affecte un identifiant unique, il prend soin de supprimer le nœud et tous les arcs dans lesquels il participe ( dans notre cas on le supprime du tas et on le met dans un autre tas pour l'étape suivante), ensuite il parcourt tous les arcs dans le but de charger un tableau qui pour chaque nœud d'indice « i » il contient le nombre de liens avec les nœuds précédents, enfin, il parcourt tous les nœuds pour générer un tableau qui à l'indice « i » a le nombre de liens dans le sous graph induit sur les « i » premiers nœuds, au cours de son exécution dans les différentes étapes il calcule le « core value du graph », « the average degree density », « The edge density » et « the size of a densest core ordering prefix ».

## Complexité :

Le chargement du graphe se fait avec une complexité en  $O(\text{Nombre d'arcs})$ .

L'algorithme parcourt les nœuds pour les insérer, ensuite il parcourt tous les nœuds et décrémente le degré de tous ses voisins donc  $O(N * \text{voisins} * \log)$ .

La complexité est donc en  $O(N * du * \log)$ .

## Temps d'exécution :

```
Fichier : com-amazon.ungraph
Nombre de noeuds : 334863
Nombre d'arcs : 925872

Core value of the graph : 6
The average degree density : 2.764928
The edge density : 1.651383e-005
the size of a densest core ordering prefix : 497

Process returned 0 (0x0)   execution time : 0.768 s
```

```
Fichier : com-lj.ungraph
Nombre de noeuds : 3997962
Nombre d'arcs : 34681189

Core value of the graph : 360
The average degree density : 8.674717
The edge density : 4.339571e-006
the size of a densest core ordering prefix : 377

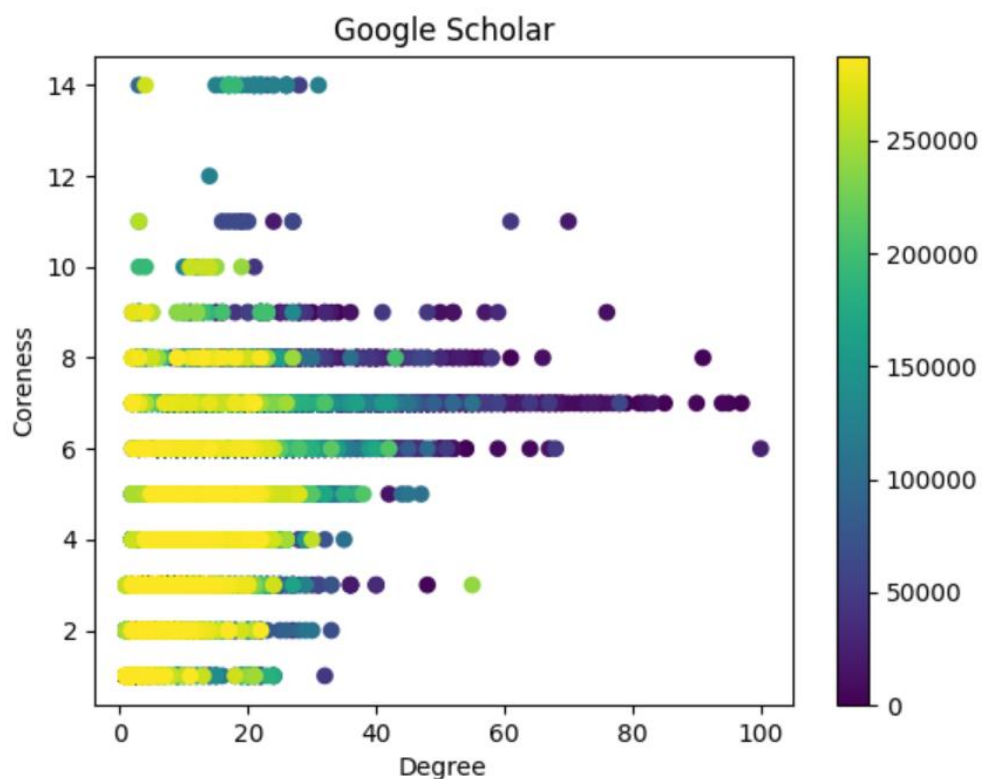
Process returned 0 (0x0)   execution time : 21.519 s
```

```
Fichier : com-orkut.ungraph
Nombre de noeuds : 3072441
Nombre d'arcs : 117185083

Core value of the graph : 253
The average degree density : 38.140719
The edge density : 2.482765e-005
the size of a densest core ordering prefix : 15706

Process returned 0 (0x0)   execution time : 85.889 s
```

## Graph mining with k-core:



```
Fichier : net
Nombre de noeuds : 287426
Nombre d'arcs : 871001

Core value of the graph : 14
The average degree density : 3.030349
The edge density : 2.108619e-005
the size of a densest core ordering prefix : 27

===Generation d'un graph===

Anomalous : Roger Fu
Anomalous : Heekwan Koo
Anomalous : Jinhee Lee
Anomalous : N. R. Mohamad

Process returned 0 (0x0)   execution time : 3.449 s
```