Master Parisien de Recherche Opérationnelle

Recherche opérationnelle et Données Massives

Graphes réels de grande dimension

Auteurs :
Guillaume Dekeyser
Thibaut Girard
Matthieu Roux

Responsable :
Maximilien Danisch

École des Ponts
ParisTech

5 avril 2020

Table des matières

1 Introduction

Les codes qui ont été utilisés pour la résolution de ces trois TPs se trouvent à l'adresse suivante : (https://github.com/Saint-Venant/massive_graphs)

2 TME 1 : Gestion de graphes de grande taille

2.1 Exercice 2 : Taille d'un graphe

Les noeuds et arêtes d'un graphes sont comptés de la manière suivante :

```
printf("Reading edgelist from file %s\n", graphPath);
adjlist* g=read_toadjlist(graphPath);
printf("\nNumber of nodes: %lu\n",g->n);
printf("Number of edges: %lu\n\n",g->e);
```

Les résultats que nous obtenons sont :

	email	amazon	LiveJournal	Orkut
n	1005	548552	4036538	3072627
m	25571	925872	34681189	117185083
temps (s)	0.03	1.6	62.2	210.8

TABLE 1 – Lecture des sommets et noeuds

La cinquième instance ne rentre pas en mémoire vive, même avec la structure de données la plus efficace et n'a donc pas été utilisée pour la suite du TP.

2.2 Exercice 3 : Nettoyage des données

On commence par charger le graphe.

```
adjlist* g = read_toadjlist_clean(graphPath, &selfLoops);
```

Il se pourrait qu'il y ait des doublons ou des arêtes d'un sommet vers lui-même. Ces deux cas sont pris en compte et l'on s'est assuré que

— Les arêtes de type (u,u) ne sont pas chargées

— Les arêtes (u,v) sont chargées en arc (u,v) si u<v et en arc (v,u) sinon.

La liste d'adjacence est ensuite faite de la manière suivante : on crée d'abord la liste d'adjacence puis, pour chaque sommet, on trie ses voisins. L'intérêt est que les doublons sont facilement repérables dans la liste triée car ils sont consécutifs.

```
mkadjlist(g);
sortadjlist(g);
```

Les résultats que nous obtenons sont :

	email	amazon	LiveJournal	Orkut
self-loops	642	0	0	0
duplicated edges	8865	0	0	0
temps (s)	0.07	4.2	163.7	546.7s

TABLE 2 – Nettoyage des données

2.3 Exercice 4 : Trois structures de données de représentation d'un graphe

Les trois principales structures de données utilisées pour représenter sont :

— Les matrices d'adjacences : cette structure représente un graphe par une matrice dont le coefficient M_{ij} vaut 0 si i et j ne sont pas voisins et 1 sinon (ou le coût de l'arc (i,j) si le graphe est pondéré). Cette structure de données a une complexité en mémoire en $O(n^2)$ est n'est donc pas utilisable pour des instances de grande taille.

— Les listes d'adjacences : cette structure représente un graphe de la manière suivante : pour chaque sommet, on liste les voisins du sommet considéré. Cette structure de données a une complexité en mémoire en $O(n + m)$. Il y a donc un gain de mémoire, d'autant plus grand que le graphe est peu dense.

— La liste d'arêtes : le graphe est représenté comme une liste de paires (u, v). La complexité en mémoire est en $O(m)$.

Remarque : Lorsque les résultats n'ont pas été reportés, c'est que le programme n'a pas pu allouer la mémoire requise et a donc cessé de fonctionner. La structure de données par matrice d'adjacence atteint donc très vite ses limites et ne sera par conséquent pas utilisée dans la suite du TP. Les deux autres structures, liste d'arêtes et liste d'adjacence, ont des performances très comparables (ce qui n'est pas étonnant car elles requièrent le même ordre de grandeur en mémoire).

	email	amazon	LiveJournal	Orkut
list of edges	0	1	11	23
adjacency matrix	0	/	/	/
adjacency array	1	1	12	39

TABLE 3 – Scalability (temps en s)

2.4 Exercice 5 : BFS

Pour déterminer les composantes connexes, nous utilisons l'algorithme de Breadth-First Search :

Breadth-First Search

1. On choisit un sommet u_0 aléatoirement. Sa file de recherche est vide
2. On ajoute les voisins non marqués de u_0 à la file de recherche.
3. Tant que la file de recherche n'est pas vide, on enlève le premier sommet de la file de recherche (et on le marque comme faisant partie de la composante de u_0) et on ajoute ses voisins non marqués à la file de recherche

Si des sommets ne sont pas marqués, on peut relancer BFS sur ces sommets pour déterminer la composante connexe à laquelle ils appartiennent.

Nous allons utiliser BFS pour déterminer le diamètre d'un graphe :

1. On choisit un sommet u_0 aléatoirement
2. BFS est lancé sur u_0. Les sommets de la composante connexe sont marqués par leur distance à u_0
3. On choisit aléatoirement un des sommets les plus éloignés de u_0 et on relance BFS dessus

	email	amazon	LiveJournal	Orkut
Proportion de sommets dans la principale composante connexe	98.11%	61.04%	99.04%	99.99%
composante principale (nbre de noeuds)	986	334863	3997962	3072441
noeuds isolés (nbre de noeuds)	19	213689	38576	186
Borne inférieure du diamètre du graphe (itérations de BFS)	7 (13 itérations)	47 (13 itérations)	21 (12 itérations)	10 (5 itérations)
Temps total d'exécution (en s)	0	1	22	547

Table 4 – BFS

2.5 Exercice 6 : Triangles

Pour lister les triangles du graphe, nous commençons par ré-indexer les sommets par ordre de degré décroissant. Nous ne considérons que les arcs (u, v) tels que $u < v$ (la liste des voisins est donc tronquée).

On note pour tout sommet u du graphe $\text{tsl}(u)$ la liste ordonnée et tronquée des voisins de u.

— Pour tout arc (u, v) du graphe
  — Soit W l'intersection de $\text{tsl}(u)$ et $\text{tsl}(v)$, déterminé en $O(|\text{tsl}(u)| + |\text{tsl}(v)|)$ opérations car les voisinages sont ordonnés
  — Pour tout w dans W, u,v,w est un triangle

Les résultats que nous obtenons sont :

	email	amazon	LiveJournal	Orkut
nombre de triangles	105461	667129	177820130	627584181
temps (s)	0	1	188	896

Table 5 – Triangles

2.6 Tableau récapitulatif du TME 1

	email	amazon	LiveJournal	Orkut
size	n = 1005 m = 25571 (0.03s)	n = 548552 m = 925872 (1.6s)	n = 4036538 m = 34681189 (62.2s)	n = 3072627 m = 117185083 (210.8s)
cleaning	642 self-loops 8865 duplicate edges (0.07s)	0 self-loops 0 duplicate edges (4.2s)	0 self-loops 0 duplicate edges (163.7s)	0 self-loops 0 duplicate edges (546.7s)
connected components	main component : 986 nodes 19 isolated nodes (0.04s)	main component : 334863 nodes 213689 isolated nodes (2.3s)	main component : 3997962 nodes 38576 isolated nodes (82s)	main component : 3072441 nodes 186 isolated nodes (275s)
diameter lower bound	7 (0.04s)	47 (2.4s)	21 (87s)	10 (272s)
number of triangles	105461 (0.06s)	667129 (3.4s)	177820130 (149s)	627584181 (896s)

3 TME 2A : PageRank

3.1 Exercice 1 : PageRank (graphe orienté)

3.1.1 PageRank

PageRank détermine la probabilité stationnaire d'un marcheur aléatoire sur un graphe orienté.
— Un marcheur commence sur un noeud u choisi aléatoirement
— Il se téléporte avec probabilité α vers un autre noeud quelconque
— Si il ne s'est pas téléporté, il choisit un noeud sortant de manière équiprobable.
— Si il n'existe pas de noeud sortant, il se téléporte sur un noeud du graphe aléatoirement.

3.1.2 PowerIteration

La matrice de transition du graphe est définie de la manière suivante $T_{u,v} = \frac{1}{d^{out}(u)}$ si $d^{out}(u) \neq 0$ et 0 sinon

P vaut initialement $\frac{1}{n} * \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$

Tant que le critère d'arrêt n'est pas atteint :
— $P \leftarrow MatVectProd(T, P)$
— $P \leftarrow (1 - \alpha)P + \alpha I$
— $P \leftarrow Normalize2(P)I$

3.1.3 Implémentation

Pour l'implémentation de l'algorithme de PageRank avec la méthode Power Iteration, nous avons choisi de mesurer, pour chaque itération, la norme du vecteur différence entre 2 valeurs successives du vecteur P (vecteur qui est censé converger vers les valeurs de PageRank).

Critère d'arrêt : Ainsi, au fur et à mesure de la convergence de l'algorithme, cette norme doit converger vers 0. Nous avons alors choisi comme critère d'arrêt $d_t < 10^{-9} d_{max}$, où d_t **est la norme 2** du vecteur différence à l'itération t et $d_{max} = \max_{i \leq t} d_i$ la plus **grande norme observée** (depuis le début de l'algorithme).

3.1.4 Résultats

top 5 pages (page rank)	United States	(1.95e-3)
	United Kingdom	(8.55e-4)
	Germany	(7.33e-4)
	2007	(7.31e-4)
	2006	(7.29e-4)
last 5 pages (page rank)	Hotel Babylon episode list	(3.93e-8)
	Los Time	(3.93e-8)
	Jon Cole	(3.93e-8)
	Salt of aspartame-acesulfame	(3.93e-8)
	Stefan Gaulin	(3.93e-8)
nombre d'itérations	33	
temps (s)	47	

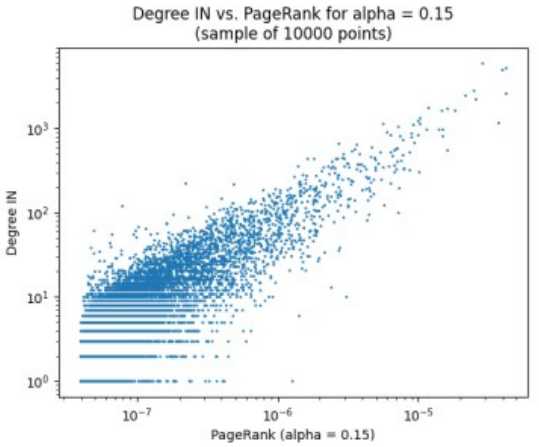
TABLE 6 – Power Iteration

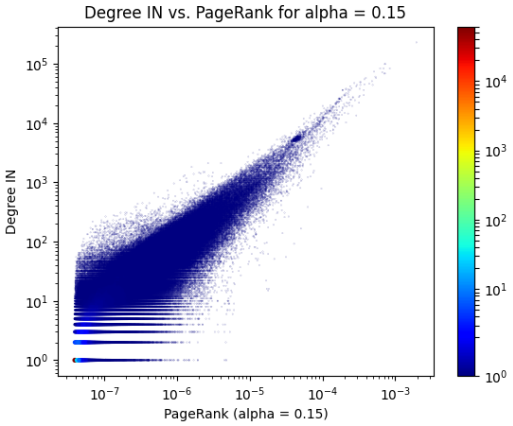
3.2 Exercice 2 : Corrélations

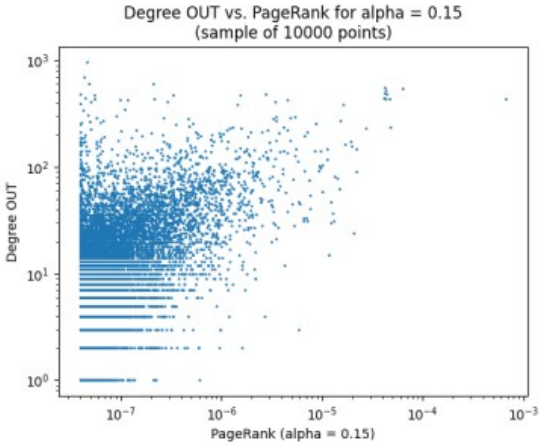
Pour cet exercice, nous avons choisi de calculer les données (faire tourner l'algorithme de PageRank) en C, puis d'exporter les résultats dans un fichier texte afin de les exploiter avec Python pour tracer les graphiques. Comme le graphe Wikipedia entier comporte quelques 13 millions de sommets, cela faisait beaucoup trop de points (scatter plot) sur une seule figure.

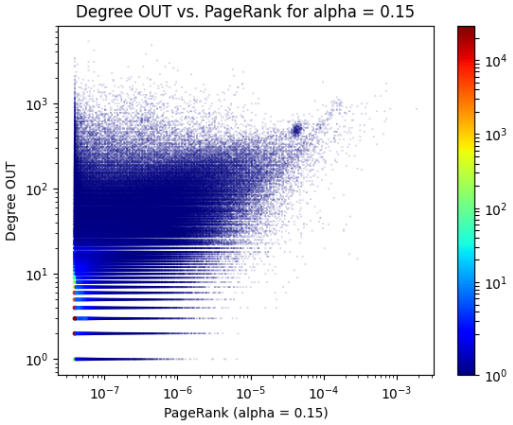
Nous avons donc dans un premier temps opté pour une approche consistant à ne représenter sur le graphique qu'un sous-ensemble de points tirés au hasard (partie gauche). Puis, dans un second temps, comme cette première approche ne permettait pas de distinguer les zones plus ou moins denses en points, nous avons représenté les scatter plots avec des couleurs correspondant à la densité de points (partie droite). Ainsi, on voit que la zone la plus dense est de loin celle avec un très faible PageRank.

Remarque : Etant donné la distribution des données, il était absolument nécessaire de tracer ces figures avec des échelles logarithmiques.



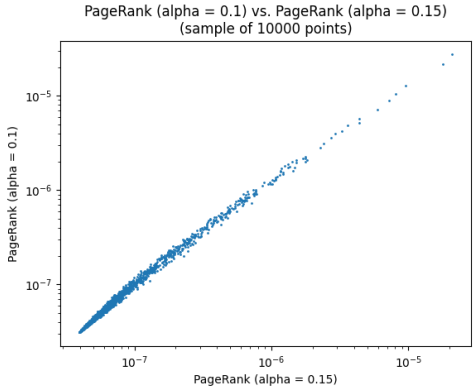


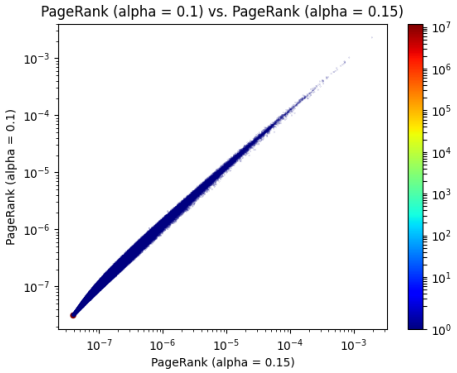


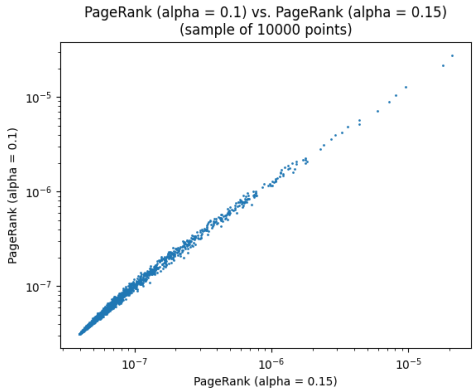


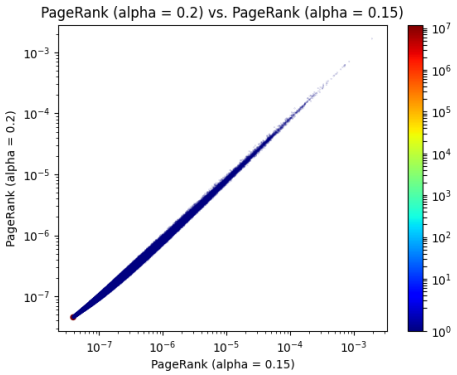
Analyse : Nous pouvons remarquer que le PageRank d'une page est beaucoup plus corrélé au degrés entrants qu'aux degrés sortants, ce qui est ce que l'on attendrait intuitivement.

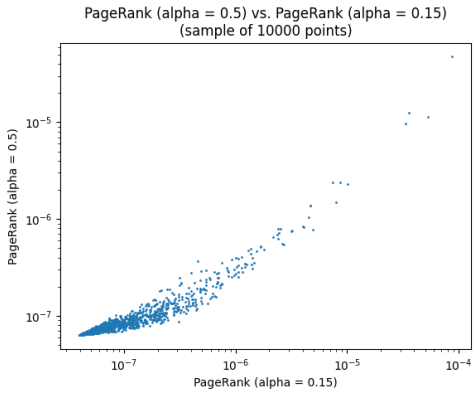
Sur les graphiques de droite, nous avons coloré les points selon la densité de présence de pages pour un PageRank et un degré (sortant ou entrant) donné. Nous observons que la grande majorité des pages ont des PageRank faibles et que zones de PageRank/degré élevés sont peu denses en points.

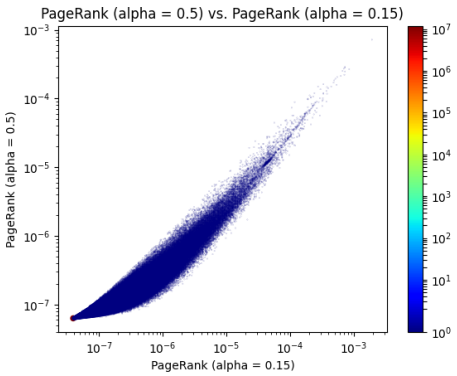

PageRank (alpha = 0.1) vs. PageRank (alpha = 0.15)
(sample of 10000 points)
PageRank (alpha = 0.1)
PageRank (alpha = 0.15)

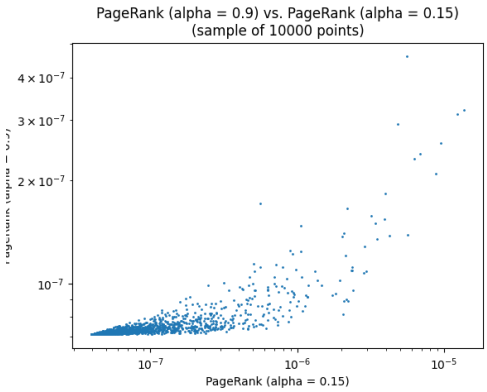

PageRank (alpha = 0.1) vs. PageRank (alpha = 0.15)
PageRank (alpha = 0.1)
PageRank (alpha = 0.15)

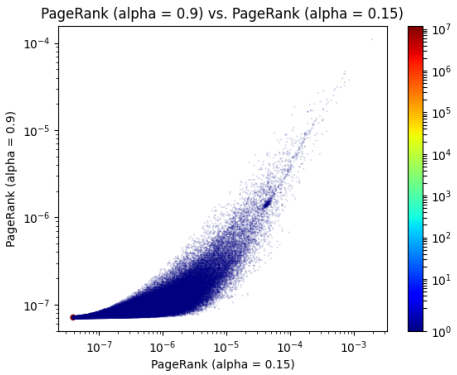

PageRank (alpha = 0.1) vs. PageRank (alpha = 0.15)
(sample of 10000 points)
PageRank (alpha = 0.1)
PageRank (alpha = 0.15)


PageRank (alpha = 0.2) vs. PageRank (alpha = 0.15)
PageRank (alpha = 0.2)
PageRank (alpha = 0.15)


PageRank (alpha = 0.5) vs. PageRank (alpha = 0.15)
(sample of 10000 points)
PageRank (alpha = 0.5)
PageRank (alpha = 0.15)


PageRank (alpha = 0.5) vs. PageRank (alpha = 0.15)
PageRank (alpha = 0.5)
PageRank (alpha = 0.15)


PageRank (alpha = 0.9) vs. PageRank (alpha = 0.15)
(sample of 10000 points)
PageRank (alpha = 0.15)


PageRank (alpha = 0.9) vs. PageRank (alpha = 0.15)
PageRank (alpha = 0.9)
PageRank (alpha = 0.15)

Analyse : Il apparaît que des α proches induisent des distributions fortement corrélées.

De plus, les α élevés (typiquement 0.9) séparent moins les pages les moins importantes : puisque la probabilité de saut aléatoire est très importante, avoir peu liens qui pointent vers une page ou pas de lien du tout n'a (presque) aucune incidence sur le PageRank de la page. Ce qui se traduit par le fait qu'un α élevé a tendance à lisser les différences entre les pages les moins bien classées.

4 TME 2B : k-core

4.1 Exercice 3 : Décomposition en k-core

4.1.1 Algorithme rapide de décomposition en k-core

Algorithme :

$c \leftarrow 0$

Tant que tous les sommets n'ont pas été marqués :

— Sélectionner aléatoirement un sommet v parmi les sommets de degré minimum

— $c \leftarrow max(c, d_G(v))$

— On assigne la valeur c au noeud courant v

— On retire v du graphe

— On retire les arêtes de v du graphe

MinHeap : Pour simplifier l'implémentation de l'algorithme de décomposition en k-cores, une structure de MinHeap a été codée. L'utilité principale est d'avoir les sommets déjà triés par ordre de degré, et donc de pouvoir sélectionner facilement un des sommets de degré minimum, dans une structure qui évolue.

4.1.2 Résultats

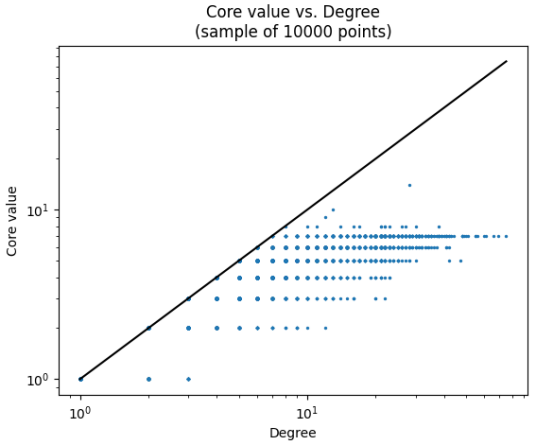
Les résultats que nous obtenons sur les datasets du premier TME sont :

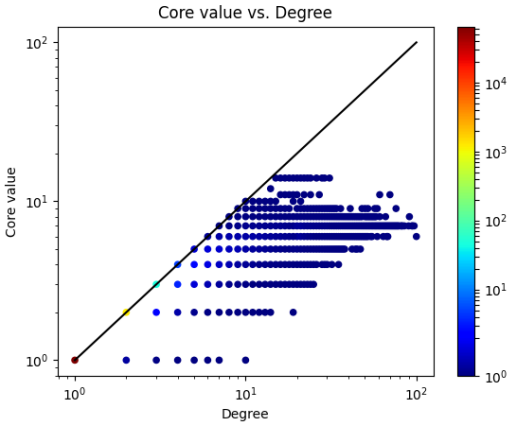
	email	amazon	LiveJournal	Orkut
core value	34	6	360	253
temps (s)	0	2	26	64

Table 7 – Core value

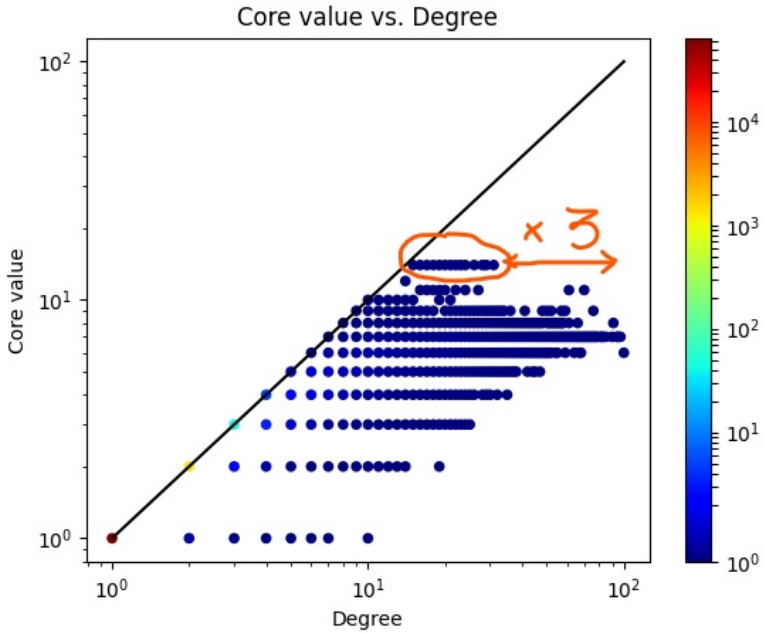
4.2 Exercice 4 : Utilisation des k-cores pour extraire des informations

En calculant la décomposition en k-cores sur le graphe de Google Scholar, on obtient les graphiques suivants :





On s'aperçoit que le k-core de valeur la plus élevée correspond à un groupe dont le degré maximum de ses membres est "anormalement" bas. Cela semble indiquer que ce groupe est fortement interconnecté, mais qu'il a anormalement peu de connections avec les autres membres du réseau.



On trouve que ce groupes possédant la plus grande core value (ie 14) est composé de 27 auteurs, dont les noms sont les suivants :

— Sa-kwang Song
— Sung-Pil Choi
— Chang-Hoo Jeong
— Yun-soo Choi
— Hong-Woo Chun
— Jinhyung Kim
— Hanmin Jung
— Do-Heon Jeong
— Myunggwon Hwang

— Won-Kyung Sung
— Hwamook Yoon
— Minho Lee
— Won-Goo Lee
— Jung Ho Um
— Dongmin Seo
— Mi-Nyeong Hwang
— Sung J. Jung
— Minhee Cho
— Sungho Shin
— Seungwoo Lee
— Heekwan Koo
— Jinhee Lee
— Taehong Kim
— Mikyoung Lee
— Ha-neul Yeom
— Seungkyun Hong
— Yun-ji Jang

5 TME 3 : Détection de communautés

5.1 Exercice 1

On génère des graphes aléatoires ayant quatre communautés de cent noeuds chacun, avec p la probabilité que deux noeuds dans la même communauté soient reliés, et q la probabilité qu'un lien extra-communautaire existe. Plus le rapport $\frac{p}{q}$ est élevé, plus les communautés sont soudées par de nombreux liens intra-communautaires et peu de liens extra-communautaires, et plus la détection de communautés est aisée (voir figures 1 et 1.

FIGURE 1 – Graphe facile : $p = 0.1$ et $q = 0.005$.

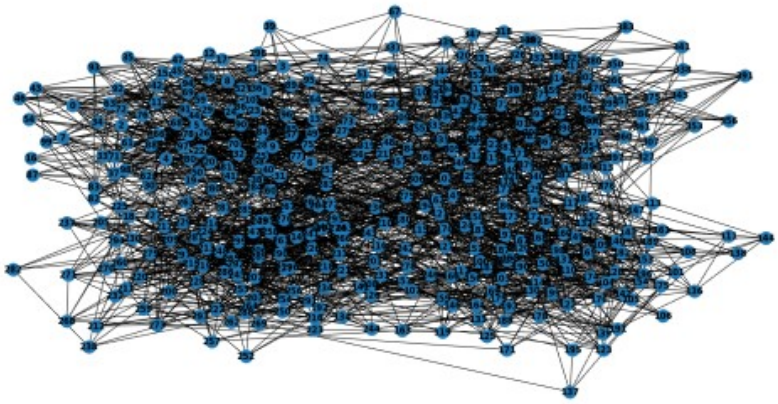


FIGURE 2 – Graphe plus ardu : $p = 0.05$ et $q = 0.005$.

5.2 Exercice 2

L'algorithme a été codé en privilégiant une faible occupation mémoire, quitte à être un peu lent au moment de la lecture des données. L'implémentation permet de faire chaque itération avec une complexité temporelle de $O(m)$. Des résultats de l'algorithme sont représentés sur les figures 3 et 4. Sur ces figures, la position des points ne révèle a priori pas la communauté "ground truth", et il n'y a en fait pas eu de comparaison effectuée de manière systématique entre la coloration proposée et la ground truth.

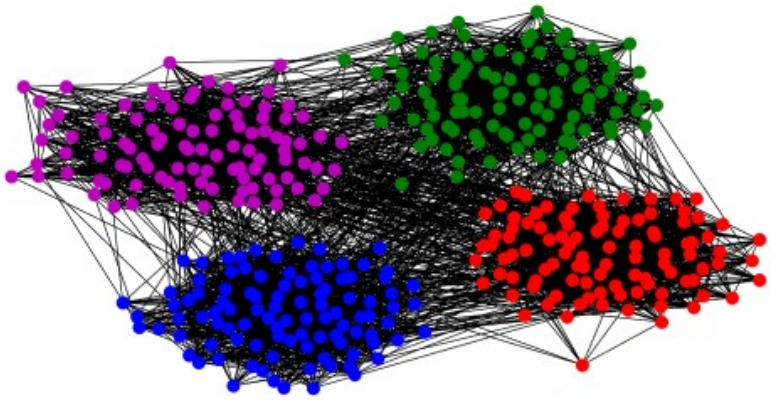


FIGURE 3 – Communautés détectées sur le graphe facile : $p = 0.1$ et $q = 0.005$.

5.3 Exercice 3

Les algorithmes de propagation de label et de Louvain permettent tous les deux de traiter des grandes graphes, comme le prouvent les résultats du tableau 8 où les instances sont celles du premier TP. L'algorithme de Louvain tel qu'il a été implémenté est particulièrement rapide.

Quatre graphes ont été générés avec le LFR benchmark :

— ./lfrbench_udwov -N 1000 -k 15 -maxk 30 -muw 0.1 -on 0 -name LFR1

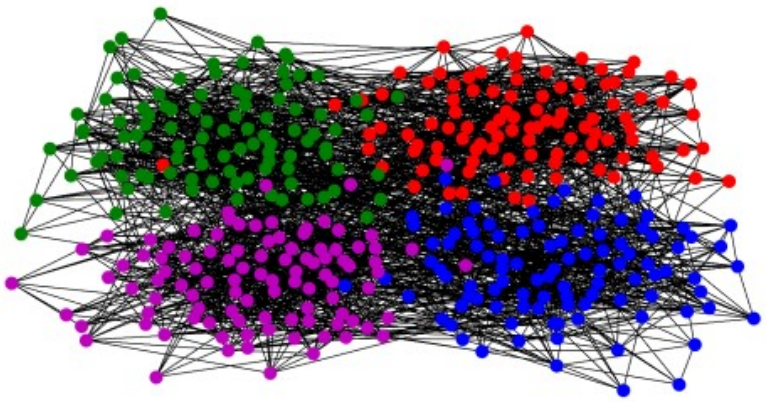


FIGURE 4 – Communautés détectées sur le graphe plus ardu : $p = 0.05$ et $q = 0.005$.

	email	amazon	LiveJournal	Orkut
label propagation	7 itérations (0.13s)	15 itérations (8.3s)	82 itérations (519s)	36 itérations (1143s)
Louvain algorithm	0.08s	1.8s	24s	126s

TABLE 8 – Temps de calcul des algorithmes sur des grands graphes

— ./lfrbench_udwov -N 1000 -k 10 -maxk 30 -muw 0.1 -on 0 -name LFR2
— ./lfrbench_udwov -N 1000 -k 5 -maxk 30 -muw 0.1 -on 0 -name LFR3
— ./lfrbench_udwov -N 1000 -k 10 -maxk 20 -C 1 -muw 0.1 -on 0 -name LFR4

Les comparaisons de quelques scores sont présentés dans le tableau 9. On remarque que la moyenne des distances d'édition des clusters est généralement en opposition avec les autres scores pour comparer les résultats. Le même problème a été observé avec la somme, donc il faudrait trouver une autre manière de généraliser ce score pour évaluer une partition en communautés (peut-être regarder le max).

	l2/(l1+l2) (avg)	edit distances (avg)	modularity
p=0.1, q=0.0005	0.929 0.929	3989 3989	0.734 0.734
p=0.1, q=0.005	0.7594 0.7594	4265.5 4265.5	0.613 0.613
p=0.05, q=0.005	0.609 0.598	4773 4789	0.51 0.50
p=0.03, q=0.005	0.51 0.41	10361 5106	0.31 0.33
LFR1	0.014 0.017	435 540	0.012 0.014
LFR2	0.0094 0.012	272 420	0.013 0.021
LFR3	0.0036 0.01	116 431	-0.014 -0.006
LFR4	0.0085 0.018	182 469	0.012 0.015

TABLE 9 – Scores des algorithmes de propagation de label (résultats du haut des cases) et de Louvain (résultats du bas des cases). Le meilleur score est souligné.