Rapport du TME3 CPA

Ivan Delgado

April 8, 2019

Introduction

Le code du TME

La code a été réalisé en Rust.

Le repo se trouve a : https://github.com/Vonstrab/CPA

Pour faire tourner le programme on utilise le builder de Rust "cargo" avec l'option "- -release"

Dans un premier temps il faut "nettoyer" les fichiers de graphe , c'est a dire ré-indexer les noeuds qui sont vides

avec la commande : $ cargo run --release -- --clean <fichier-ungraph.txt> <fichier_output_clean.txt>

la structure du code:

le code propre a chaque tme se trouve dans src/tmex

Part I
TME3

Pour lancer tous les calculs de ce TME sur un fichier il faut faire : cargo run --release -- --tme3 <fichier_clean.txt>

A special quantity :

Table 1: Qvalues et temps de calculs pour chaque graphe

Nom	Q value	Temps
Eu mail	88 109 182	0 s 10 ms
Amazon	103 415 531	3 s 37 ms
LJ	789 000 450 609	33 s 778 ms
Orkut	22 292 678 512 329	1 m 13 s 220
Friendster	379 856 554 324 947	31 m 58 s 112ms

Densite des degres : voici les graphes des densités des nodes pour chaques graphes:

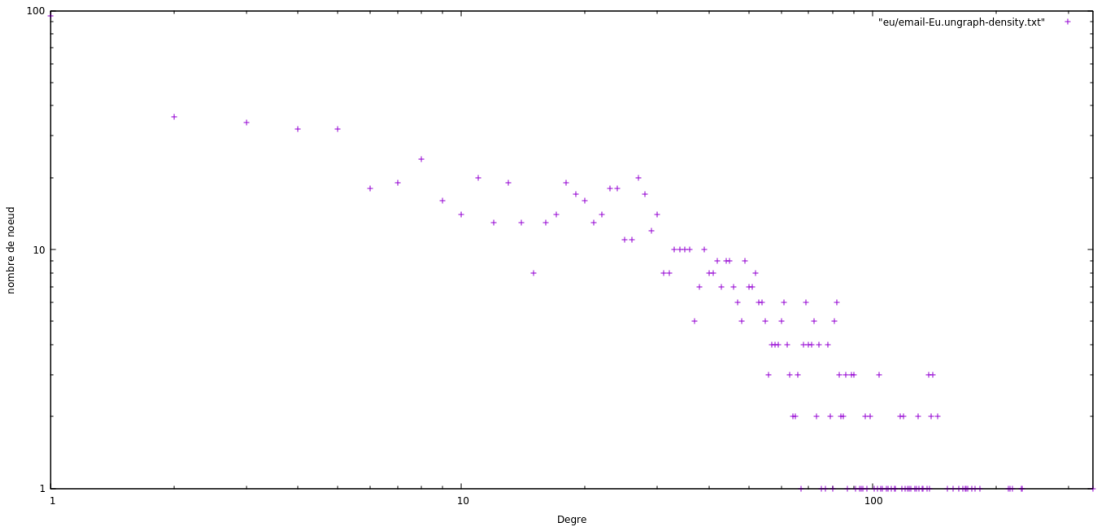

Figure 1: eu-mail

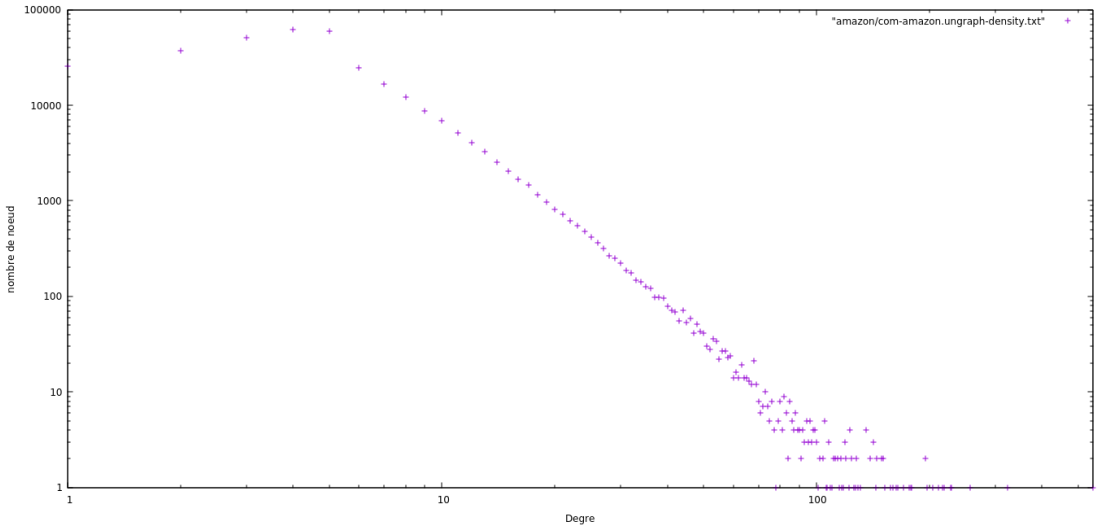

Figure 2: amazon

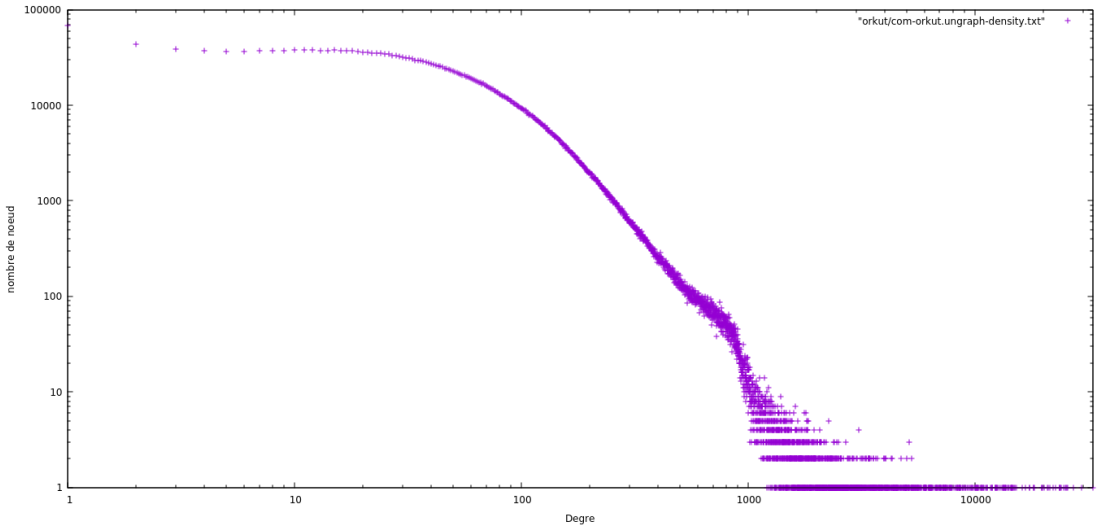

Figure 3: Orkut

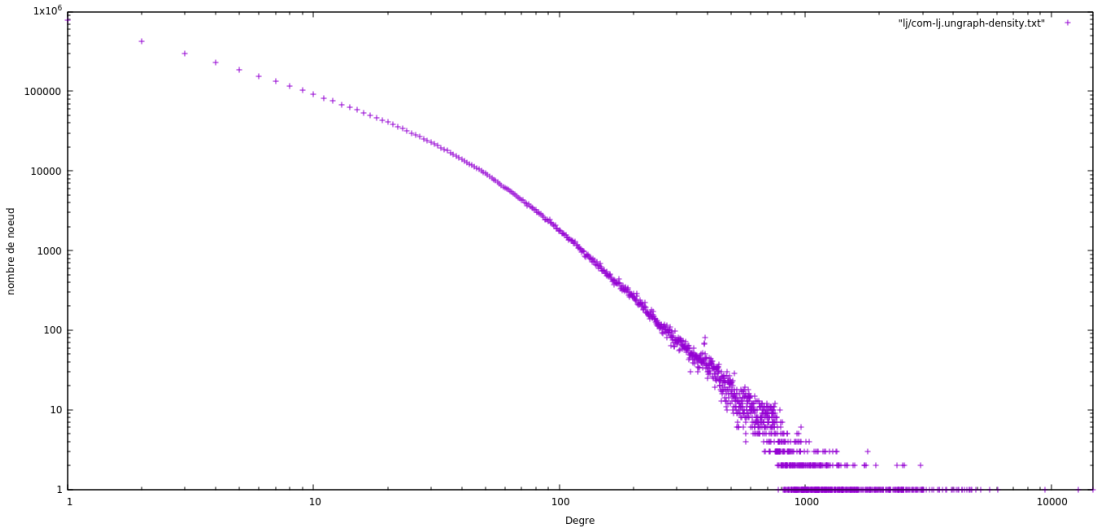

Figure 4: Lj

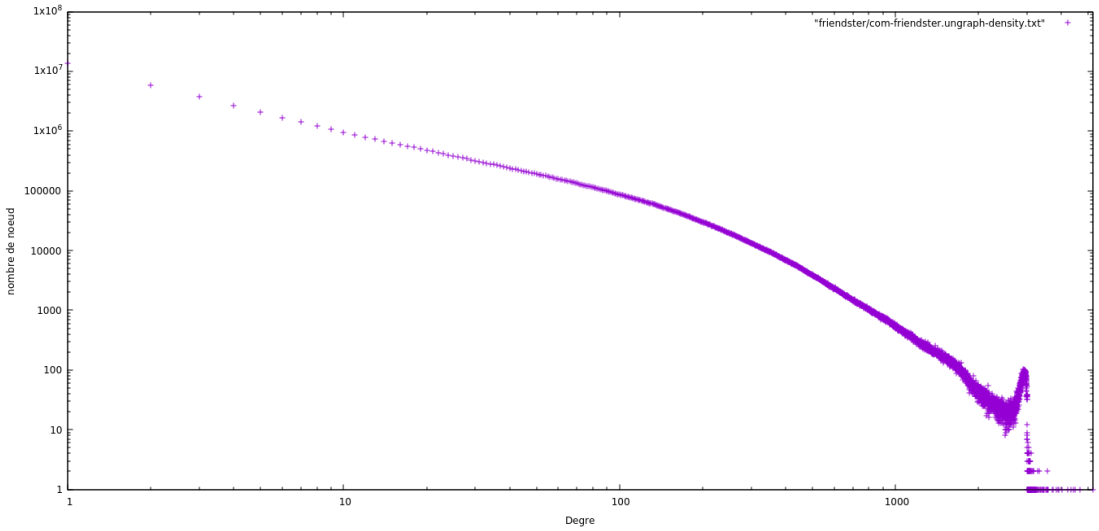

Figure 5: Friendster

On voit des similaritées entre les graphes, les noeuds ont tendance a avoir tres peu de voisins comparé au nombre de noeuds total. La majorité ont moins de 10% des noeuds en vosins pour eu-mail et moins d'un milliéme voire un millioniéme pour les autres.

1 Load in Memory

On a implémenté les trois structure pour les graphes , voici les performances

- Eu-mail:
  - Matrice d'adjacence : 1 mo en 3 ms
  - Liste des Arêtes : 257ko en 6 ms
  - Liste d'adjacence : 132ko en 11 ms
- Amazon:
  - Matrice d'adjacence : 280 Go (Théoriquement)
  - Liste des Arêtes : 1.4Mo en 118 ms
  - Liste d'adjacence : 0.740ko en 379 ms
- Lj:
  - Matrice d'adjacence : 15To (Théoriquement)

  - Liste des Arêtes : 554Mo en 4 s 327 ms
  - Liste d'adjacence : 293Mo en 10 s 897 ms

- Orkut:
  - Matrice d'adjacence : 8 To (Théoriquement)
  - Liste des Arêtes : 1.8 Go en 33s 398 ms
  - Liste d'adjacence : 0.9 Go en 62 s 43 ms

- Friendster : Pas assez de Ram
  - Matrice d'adjacence : 13 880 To (Théoriquement)
  - Liste des Arêtes : 13.45 Go (Théoriquement)

On voit que le nombre de degré faible des noeuds, comme vu précedement rend la matrice d'adjacence complétement inutilisable dés le deuxiéme graphe. La liste d'adjacence a de meilleures performances spaciales par rapport a la liste des arêtes ,mais est plus lente.

2 BFS and Diameter

Implémentation d'un algorithme de BFS
On implémente les algitithmes pour la structure Liste d'Adjacence.

La methode bfs_from(Node Index) rend une liste de tous les noeud du composant dans l'ordre exploré en largeur.

La methode composants() rend la liste des composants du graphe, on remarque que pour tous les graphes on a un grand composants et un nuage de noeuds isolés.

La taille du plus gros composant est :

- Eu-mail : 986 sur 1 005 noeuds soit 98%
- Amazon : 334 863 sur 548 552 noeuds soit 61%
- Lj : 3 997 962 sur 4 036 538 noeuds soit 99%
- Orkut : 3 072 441sur 3 072 627noeuds soit 99%
- Friendster : -

La methode pour minorer le diametre est de faire, pour le gros composant une BFS depuis un noeud au hasard, soit U le dernier noeud retourné par la BFS , on refait une BFS a partir de U et on prend encore une fois le dernier noeud W, la minoration du diamétre est la distance UW

- Eu-mail : 7
- Amazon : 47

- Lj : 21
- Orkut : 9
- Friendster : -

3 Triangles

- Eu-mail : 105461 en 0.012 s
- Amazon : 666767 en 173s
- Lj : -
- Orkut : -
- Friendster : -

Part II
TME4

4 Simple Benchmark

Pour génerer les graphes aléatoires on utilise la commande : cargo run --release -- --tme4rand <p> <q>

on obtien les figures suivantes

Figure 6 : avec la même probabilité on a un graphe homogéne

Figure 7 : avec un grande disproportion entre p et q on commence a voir les communautés se singulariser

Figure 8 : avec une moindre probabilité q on dénote mieux les différentes communautés , on a encore certains noeud qui sont ambigus , a cheval entre deux communautés

Figure 9 : avec q = 0.0001 les communautés sont clairement différenciables

5 Label Propagation

Pour génerer le fichier d'histogramme des labels on utilise la commande : cargo run --release -- --tme4hist <fichier_source> <fichier_dest>

Pour youtube , on trace avec gnuplot et on obtient :

L'algorithme de propagation est non-déterminisme, le nombre de round nécessaire est variable (entre 15 et 30), le nombre de communautées calculées s'en ressent.

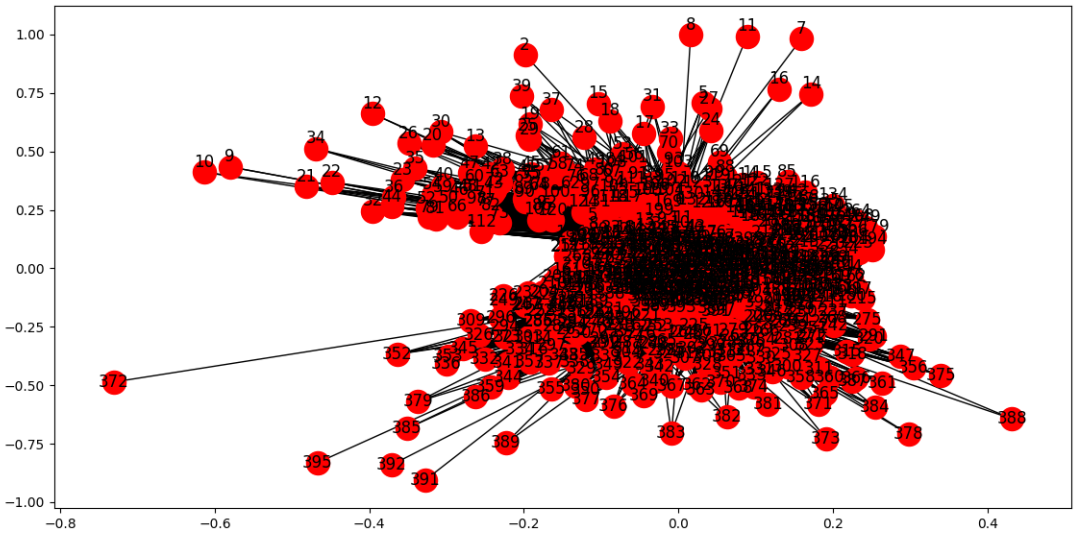

Figure 6: pc= 0.2 pp=0.2

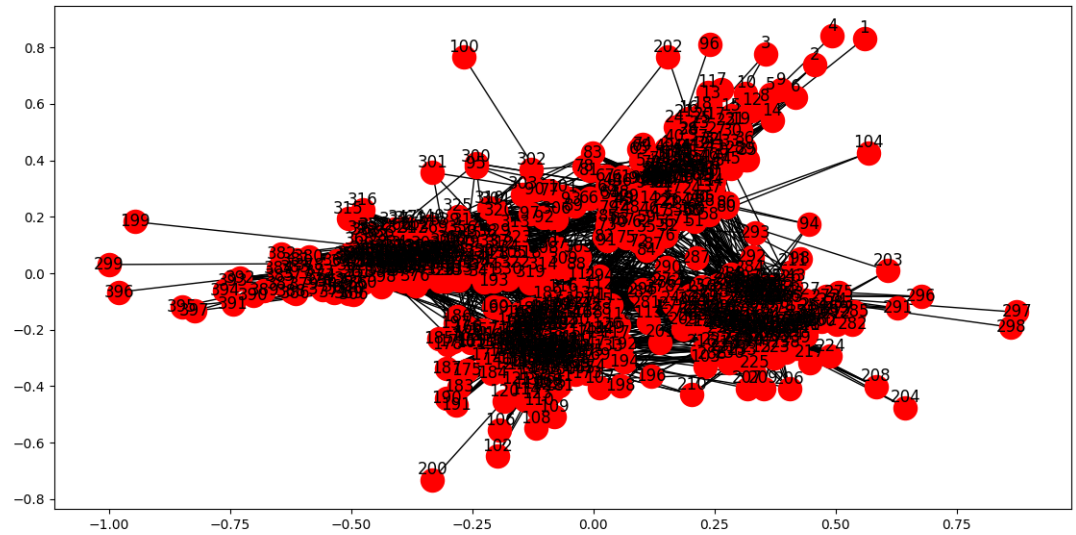

Figure 7: p= 0.7 q=0.01

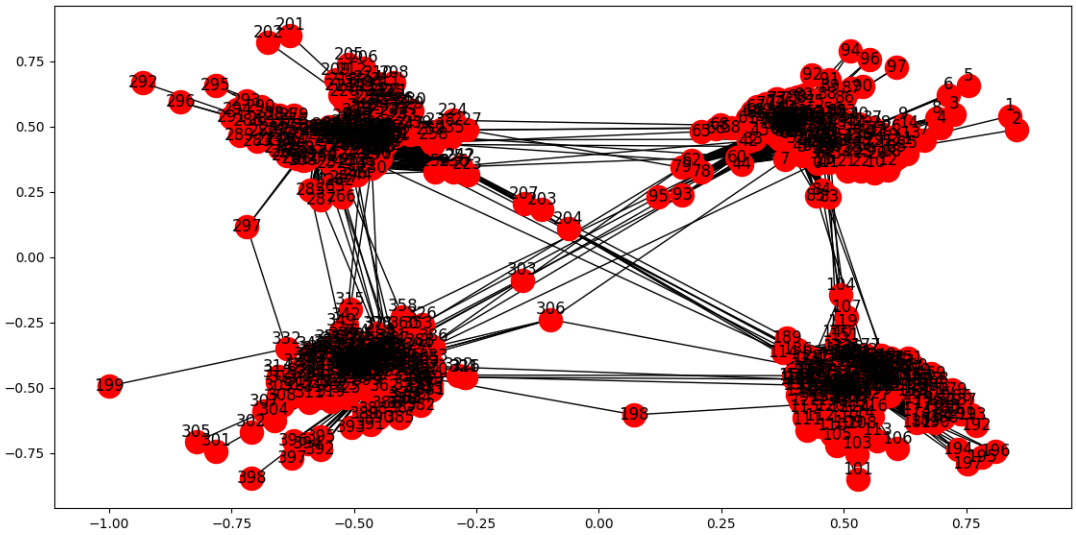

Figure 8: p = 0.7 q =0.001

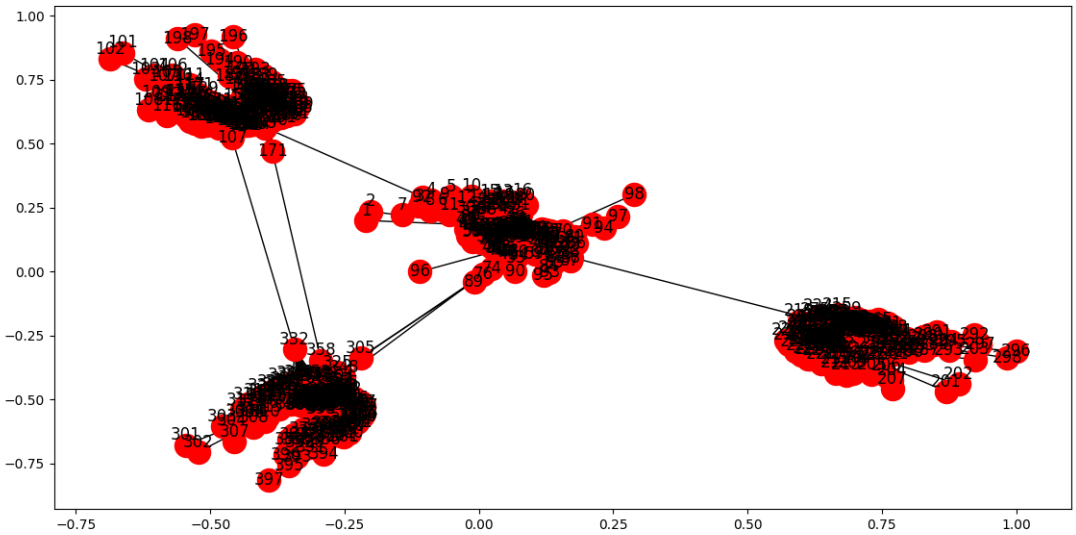

Figure 9: p= 0.7 q= 0.0001

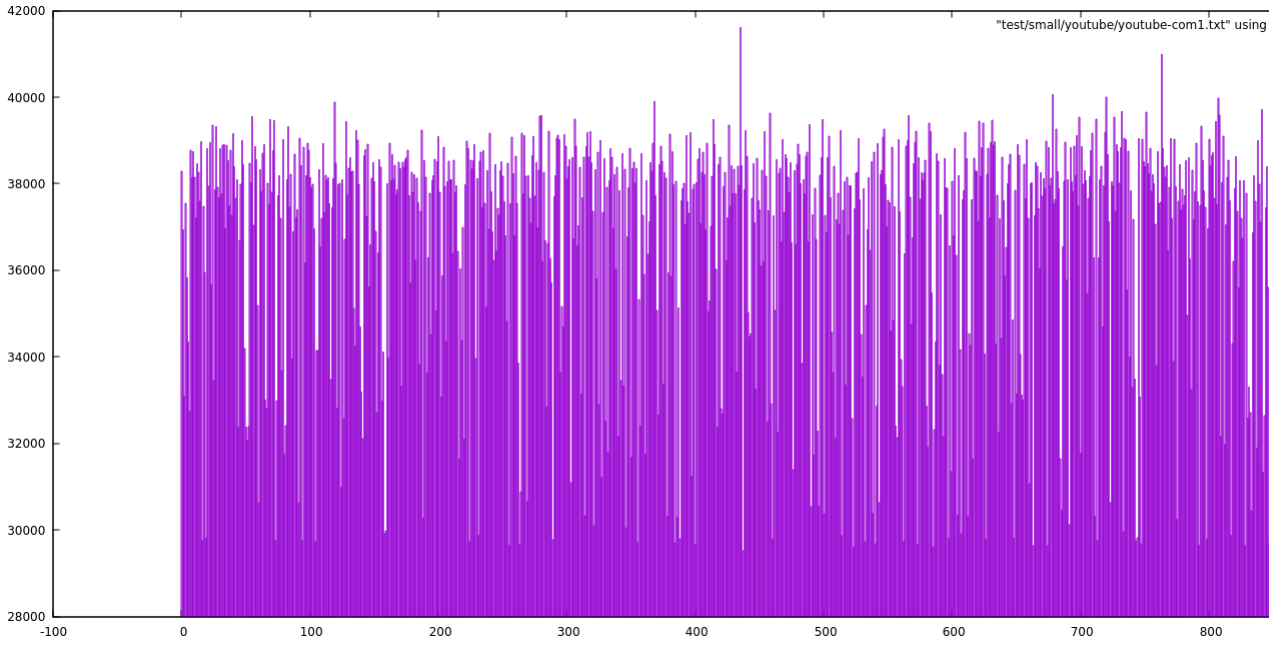

Figure 10: histogramme pour 1000 calculs de label

6 Validation

Pour compare les resultat de louvain on lance pour un fichier on lance : cargo run --release -- --tme4hcomp <fichier_source>

Cependant mon implémentation de l'algorithme laisse a désirer, les valeurs de deltaQ semblent absurdement hautes, le code est dans src/tme4.

Part III
TME5

le code du page rank se trouve dans src/tme5 et se lance avec : cargo run --release -- --tme5 <intput.txt> <fichier_pagerank.txt>

7 PageRank

l'implémentation actuelle se sert d'une matrice, ce qui ne permet pas de passer a l'echelle, de plus le résultat donne la même valeur a chaque noeud

Part IV
TME6

le code du tme se trouve dans src/tme6

8 k-core decomposition

la decomposition se lance avec : cargo run --release -- --tme6decomp <input.txt>

- eu
  - average degre density : 1.0135
  - edge density : 74
  - size : 1.0
- amzon
  - average degre density : 1.000189
  - edge density : 0.9887323
  - size : 5264

les tas de rust ne supportent pas le fait de modifier des noeuds en place, dans mon implémentation on construit et déconstruit le tas a chaque fois , ce qui rend le temps pour des grands graphes tels que lj ou orkut prohibitif.

Part V
Conclusions

Premiere conclusion de ces TME, implémenter des alogorithmes efficaces qui passent a l'échelle ça prend beaucoup de temps, d'autant plus quand un calcul prend 5-10 minutes. C'est d'autant plus compliqué quand on est en monôme, penser a insister pour que le projet soit fait en binôme.

Deuxiéme remarque quand a l'utilisation de Rust; c'est un language qui se veut efficace et qui est tout trouvé pour le calcul de grands graphes passant trés bien l'échelle, outre le temps suplémentaire demandé par l'apprentissage d'un nouveau language, Rust demande un plus grande rigeur dans la gestion des pointeurs et des allocations (devoir aussi un peu se battre avec le borrow checker), mais quand j'ai vu certains autres groupes passer un partie de leurs scéances a chercher des segfaults, la sureté memoire du language a du me faire gagner au change.

Mention notable a l'outil cargo qui est un builder complétement intégré au compliteur et qui associe une simplicité d'utilisation avec de nombreux outils

tels qu'un profiler et Critérion, qui permet de faire des benchmarks du code avec un soucis de la rigeur statistique, permettant d'analyser avec facilité du code et ainsi de l'optimiser.