Rapport VLG

Chady DIMACHKIE

Table des matières

1 Introduction

Le projet choisi est le benchmarking de la méthode de calcul de communautés de Louvain sur des graphes de type caveman.

2 Graphes caveman

Le graphe caveman choisi est celui décrit par Watts dans [Watts, D. J. Small Worlds : The Dynamics of Networks between Order and Randomness. Princeton, NJ : Princeton University Press, 1999.], entre autres.

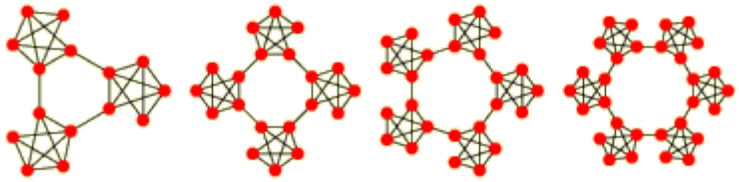

FIGURE 1 – Caveman

Le graph contient K cliques de k noeuds. Chaque clique a une arête retirée qui va servir à se connecter à la clique suivante.

De plus, sachant qu'un caveman est un graphe complet, on peux en déduire le nombre d'arêtes comme suit :

$$nb\_edges = \frac{k(k-1)}{2} * K$$

3 Brève description de l'implémentation

Un programme en C++ a été écrit pour générer des graphes caveman. Le C++ a été choisi pour sa capacité à produire des programmes très rapides.

Le programme a été écrit de tellement manière à ne rien stocker en RAM. Ceci fait que tout dépend de la taille et de la vitesse du disque sur lequel le graphe est généré, car ce sera le facteur limitant.

De plus, le programme est multi-threadé, de telle sorte à faire plusieurs graphes en même temps.

4 Résultats

Un notebook Jupyter écrit en Python a été réalisé pour cette partie.

Il faut noter que le benchmark a été fait avec l'algorithme de Louvain original et non optimisé.

Le benchmark a été réalisé avec $k, K \in [3...1000]$.

Les tests effectués sont des tests de durée de Louvain. De plus, on rajoutera des graphiques par rapport au nombre d'arrêtes par graphe et à la taille du fichier généré sur disque.

4.1 Durée

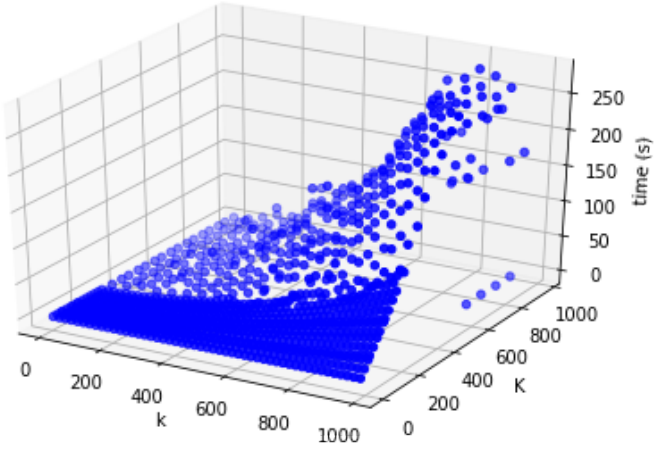

FIGURE 2 – Benchmark de Louvain en temps (s)

On voit que la première partie (avec une grande concentration de bleu), avec $k \in [3...1000]$ et $K \in [3...400]$ a été faites avec une granularité de 1, c'est-à-dire que l'on calcule tout les combinaisons des k et K, sans sauter aucune itération.

Par contre, de $k \in [3...1000]$ et $K \in [401...1000]$, on met un offset de 20, donc on saute 20 itérations à chaque fois. Sans cela, le temps de calcul aurait été encore bien plus long.

Le calcul a été répartit sur 2 machines, ayant une taille de disque suffisamment grande.

Le temps de calcul est d'environ une nuit et demi, pour les 2 machines. Les puissances des deux machines étaient équivalente,

donc aucun biais n'est à noter dans les résultats.

4.2 Espace disque

Le benchmark n'a pas pu être lancé correctement sur les machines de l'école, sachant que la taille de disque ($7GB - RAM\_utilisee$ avec un tmpfs) n'était pas clairement pas suffisante.

Le programme étant multi-threadé, avec généralement 8 cores par machine avec 1 graphe par core, il faut donc prévoir au moins $30/40GB$ d'espace disque. Pour visualiser l'espace pris par ces graphes :

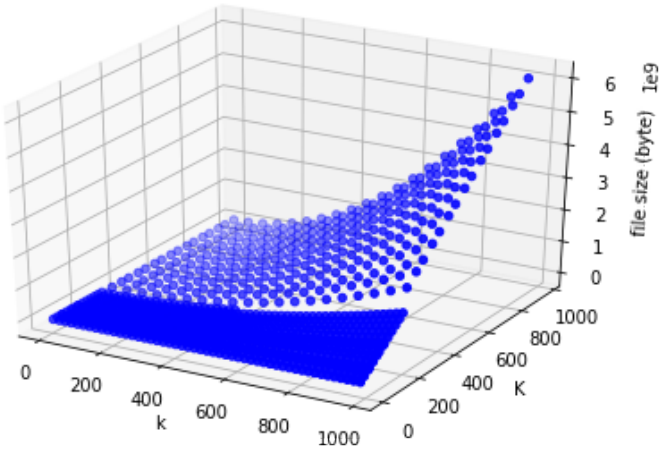

FIGURE 3 – Taille des graphes en $1e9$ Byte

4.3 Nombre d'arrêtes

On peux visualiser sur le graphe suivant le nombre d'arrêtes en fonction des k et K :

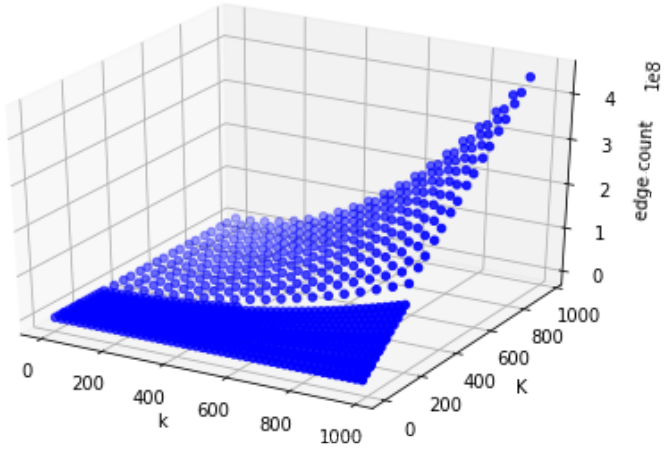

FIGURE 4 – Taille des graphes en 1e8 arrêtes

5 Commentaires

5.1 Algorithme de Louvain

On aperçois à première vue que les temps de calcul ont une croissance exponentielle, on peux en déduire qpue Louvain a une complexité en temps de $\mathcal{O}(n \log n)$.

Une brève description de l'algorithme de Louvain est nécessaire pour expliquer la remarque qui sera faites sur le cas des graphes

caveman.

5.1.1 Étape 1

En effet, pour chaque noeud, Louvain essaie de simuler le fait de l'attacher à un voisin de l'une de ses "communautés" voisines, et pour ce faire et choisir le meilleur voisin, on mesure les variations de la fonction de modularité, et on essaie de la maximiser.

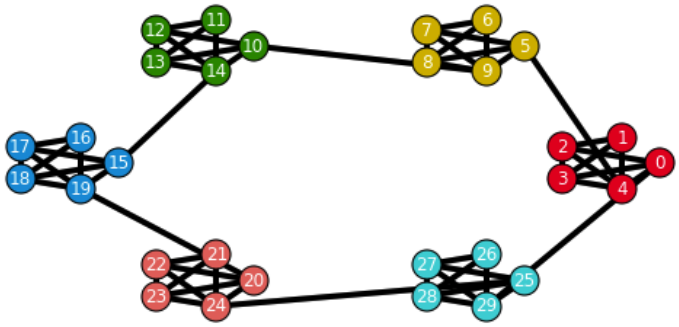

FIGURE 5 – Louvain, étape 1

5.1.2 Étape 2

Une fois tous les voisins parcourus, certaines communautés se seront déjà formées. Mais ce n'est pas fini, car Louvain, traite chacune des nouvelles communautés comme un nouveau "noeud" dans sa prochaine itération de l'algorithme. Louvain va donc essayer de créer de plus grosses communautés à partir de ces noeuds.

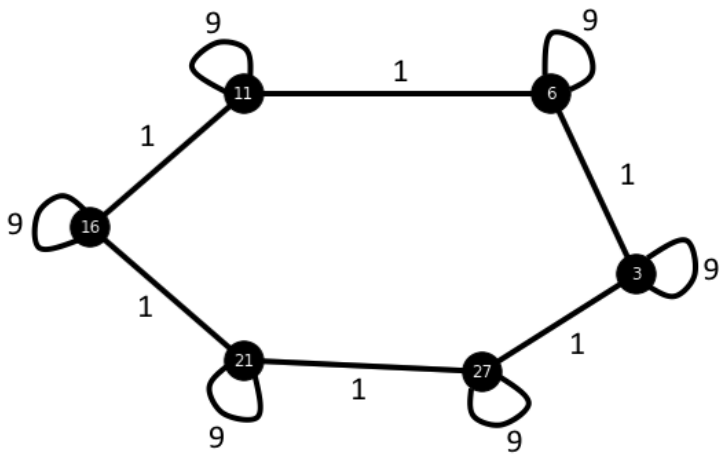

FIGURE 6 – Louvain, étape 2

On répète ces itérations jusqu'à ne plus pouvoir optimiser cette fonction de modularité (car l'étape 1, ne fait que donner les même communautés).

5.2 Remarque sur le cas des graphes caveman

On remarque que pour le cas des graphes caveman de Watts, on peux tout simplement s'arrêter à l'étape 1 et ne même pas avoir à faire toutes ces itérations.

De ce fait, on peut éviter de perdre du temps de calcul pour rien, en ayant une idée sur la distribution de nos communautés. Dans notre cas, on connais parfaitement la distribution de nos communautés.

6 Git Repository

Le projet pourra être trouvé à l'adresse suivante : https ://github.com/cpcdoy/Very-Large-Graph.git.

Il suffit de lancer :

```
sh run.sh # Lance les benchmark
jupyter notebook # Pour visualiser les résultats
```