Social network ordering to reduce cache misses

Thomas Messi Nguélé, Maurice Tchuente, Jean-François Méhaut

Numérotation des graphes sociaux pour la réduction des défauts de cache

Thomas Messi Nguélé[1,2,3] — Maurice Tchuente[1,2] — Jean-Francois Méhaut[2,3]

[1] IRD, UMI 209 UMMISCO, Université de Yaoundé I, BP 337 Yaoundé Cameroun

[2] LIRIMA, Laboratoire d'Informatique et Applications, Fac. des Sciences, Dépt. d'informatique, BP: 812 Yaoundé, Cameroun

[3] Université de Grenoble Alpes, INRIA-LIG, Corse, BP: 38400 Grenoble, France

RÉSUMÉ. L'une des propriétés des graphes sociaux est leur structure en communautés, c'est-à-dire en sous-ensembles où les nœuds ont une forte densité de liens entre eux et une faible densité de liens avec l'extérieur. Par ailleurs, la plupart des algorithmes de fouille des réseaux sociaux comportent une exploration locale du graphe sous-jacent, ce qui amène à partir d'un nœud, à faire référence aux nœuds situés dans son voisinage. L'idée de cet article est d'exploiter la structure en communautés pour optimiser le stockage des grands graphes qui surviennent dans la fouille des réseaux sociaux. L'objectif étant de réduire le nombre de défauts de cache avec pour conséquence la réduction du temps d'exécution. Dans cet article, après avoir formalisé le problème de numérotation des nœuds des réseaux sociaux comme un problème d'arrangement linéaire optimal, nous présentons pour le score de Katz, des simulations comparant les structures de données existantes (*bloc_ds* et *yale_ds*) à leurs versions exploitant la structure en communautés produite par l'algorithme de Louvain. Les résultats obtenus sur une machine NUMA (de 32 cœurs) à partir des jeux de données amazon et dblp montrent que la prise en compte de la structure en communautés contribue à diminuer les défauts de caches et par conséquent à réduire le temps d'exécution. Par exemple, sur amazon nous avons une diminution du temps d'exécution pouvant aller jusqu'à 10.5% (comparé à *bloc_ds*) et 9.5% (comparé à *yale_ds*); cette diminution du temps d'exécution correspond à une réduction des défauts de cache de 42%.

ABSTRACT. One of social graph's properties is the community structure, that is, subsets where nodes belonging to the same subset have a higher link density between themselves and a low link density with nodes belonging to external subsets. Futhermore, most social network mining algorithms comprise a local exploration of the underlying graph, which consists in referencing nodes in the neighborhood of a particular node. The idea of this paper is to use the community structure to optimise the storage of large graphs that arise in social network mining. The goal is to reduce cache misses and consequently, execution time. In this paper, after formalizing the problem of social network ordering as a problem of optimal linear arrangement, we present for Katz score, simulations that compare existing data structures (bloc_ds and yale_ds) to their corresponding versions that use the community structure produced by the Louvain algorithm. Results on a NUMA (32 cores) machine using amazon and dblp datasets show that, taking into account the community structure allows to reduce cache misses and consequently execution time. For example with the amazon dataset, we reduced execution time by up to 10.5% (compared to bloc_ds) and 9.5% (compared to yale_ds) ; this reduction of the execution time matches with the reduction by 42% of cache misses.

MOTS-CLÉS : Fouille de réseaux sociaux, Communauté, Défaut de cache

KEYWORDS : Social network mining, Community, Cache miss

1. Introduction

Lorsqu'un algorithme de fouille des réseaux sociaux (comme par exemple le calcul du score de Katz [9]) s'execute, il opère sur chaque nœud x du graphe en faisant le plus souvent référence aux nœuds situés dans le voisinage de x. Les structures de données utilisées dans les langages spécialisés de graphes ou les plates-formes d'analyse des graphes (Galois [12], Green-Marl [7] ou Stinger [1, 4]) ne considèrent que le voisinage direct de x.

Dans ce rapport, nous nous intéressons à la prise en compte dans la structure de données utilisée, d'un voisinage allant au delà du voisinage direct de x, ici la communauté de x. En d'autres termes, peut-on augmenter les performances des programmes d'analyse des graphes sociaux si l'on tient compte, dans la structure de données utilisée, de l'organisation en communautés des nœuds du graphe ?

1.1. Exemple introductif

Considérons le graphe (G) non orienté de la figure 1 ayant 16 nœuds répartis en quatre communautés.

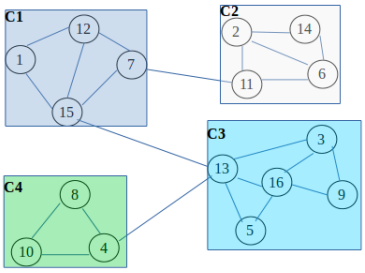


Figure 1 – Graphe (G). Les Ci représentent les communautés

Une façon simple de représenter ce graphe est d'utiliser la représentation matricielle (voir Table 1). Mais celle-ci n'est pas très appropriée pour les graphes sociaux car les matrices résultantes sont très creuses :

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1												1			1	
2						1					1			1		
3									1				1			1
4								1		1			1			
5													1			1
6		1									1			1		
7											1	1			1	
8				1						1						
9			1													1
10				1				1								
11		1				1	1									
12	1						1								1	
13			1	1	1										1	1
14		1				1										
15	1						1					1	1			
16			1		1				1				1			

Tableau 1 – Matrice représentant le graphe (G)

La représentation souvent adoptée par certains langages spécialisés de graphe (à l'instard de Galois [12] et Green-Marl [7]) est celle de Yale [5]. Cette représentation utilise

trois vecteurs :

– un vecteur A représentant les arêtes, chaque arête étant représentée par une de ses extrémités,

– un autre vecteur JA donnant l'autre extrémité de chacune des arêtes de A,

– et un dernier vecteur IA qui donne l'indice dans le vecteur initial du premier élément non nul de chaque ligne de la matrice simulée.

D'autres représentations peuvent être utilisées : le graphe est alors représenté par un vecteur de nœuds, chaque nœud pouvant être relié

1) à une liste d'adjacence de ses voisins

2) à un bloc de ses voisins, la taille du bloc étant variable (utilisé dans [14])

3) à une liste chainée de blocs (de taille fixe) de ses voisins, adaptée aux graphes dynamiques (utilisée par la plateforme Stinger [1, 4]).

La représentation de Yale de l'exemple précédent est la suivante :

A	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
–	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

JA	12	15	6	11	14	9	13	16	8	10	13	13	16	2	11	14	11	12	15	4	10	3	16
–	4	8	2	6	7	1	7	15	3	4	5	15	16	2	6	1	7	12	13	3	5	9	13

IA	1	3	6	9	12	14	17	20	22	24	26	29	32	37	39	43	47

Aucune de ces représentations ne tire profit du "regroupement en communautés" des nœuds du graphe pour réduire le temps d'exécution des algorithmes des réseaux sociaux, car ce n'était pas leur but.

2. Comment exploiter la structure en communautés des graphes sociaux pour diminuer les défauts de cache ?

2.1. Idée

L'idée est de faire en sorte que, chaque fois qu'un nœud est en cours de traitement, les autres nœuds membres de sa communauté se retrouvent dans le même cache mémoire que ce nœud. Ainsi, les données correspondant à une communauté doivent être consécutives en mémoire.

2.2. Gestion du cache

Deux cas de figure sont envisagés.

2.2.1. Cas des processeurs laissant la gestion du cache au programmeur

C'est le cas par exemple du processeur MPPA de Kalray où la mémoire locale d'un cluster peut-être perçue comme un cache. Nous ne nous interessons pas à ce cas dans ce rapport, mais nous donnons une idée qui pourra être implémentée dans la suite de ce travail.

Lorsque survient un défaut de cache, il y a un chargement de la communauté (de la donnée correspondant au nœud ayant créé le défaut de cache). Cette communauté est alors chargée dans le cache en suivant l'un des algorithmes classiques :

– algorithme optimal (la ligne de cache qui ne sera pas utilisée pour la plus grande période de temps est remplacée),

– algorithme aléatoire,

– LRU *Least Recently Used*, - FIFO *First In First Out*,

– LFU *Least Frequently Used*, ...

2.2.2. Cas des processeurs généralistes

Les processeurs généralistes tels que Intel, AMD, ARM implémentent dans le matériel leur propre algorithme de gestion de cache. Ainsi, pour bénéficier du regroupement des nœuds en communautés, le calcul des communautés est effectué au moment du chargement du graphe en mémoire.

Représentation du graphe en communauté : Le graphe est représenté par un tableau de communautés (voir figure 2). Une communauté est représentée par un tableau de ses nœuds membres. Chaque nœud est relié à un tableau de ses voisins (les voisins qui ne sont pas membres de la communauté sont classés au fond du tableau, ceci aura pour effet de donner la priorité aux membres de la communauté lors d'un traitement). Par exemple, à la figure 2, la communauté C1 est constituée d'un tabeau de 4 éléments (1,7,12,15) ; les voisins 11 du nœud 7 et 13 du nœud 15 sont classés au fond des tableaux des voisins.

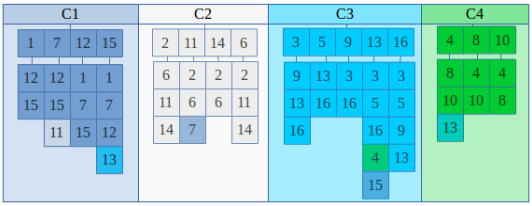


Figure 2 – Représentation de (G) tenant compte des communautés

La différence fondamentale entre cette représentation [11] et les autres représentations est la priorité accordée aux noeuds de même communauté dans le stockage des voisins.

Renumérotation du graphe initial : La nouvelle structure de données est obtenue en renumérotant le graphe initial de sorte que les noeuds appartenant à la même communauté aient des numéros consécutifs. Un exemple de graphe avant et après numérotation est donné à la figure 3.

Dans la suite, nous étudions de façon détaillée le problème de numérotation des graphes sociaux pour la réduction des défauts de cache.

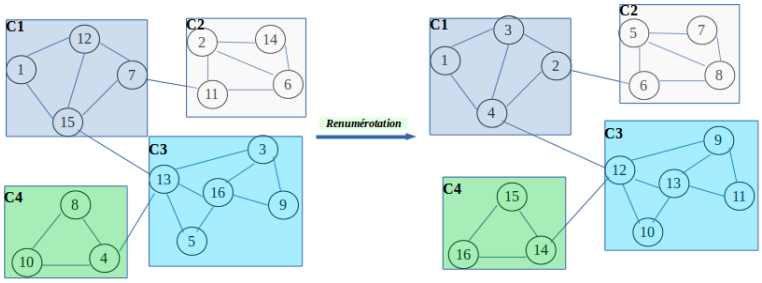


Figure 3 – Le graphe (G) avant et après numérotation. Les Ci sont les communautés

2.3. Formalisation du problème

2.3.1. Modélisation des défauts de cache

Soit

– D_{cache} : la taille du cache mémoire,

– $pos(x)$: la position occupée par le nœud x en mémoire,

La fonction définissant un défaut de cache est donnée par :

$$\sigma : \quad N \times N \longrightarrow \{0,1\}$$

$$(x,y) \longmapsto \sigma(x,y) = \begin{cases} 0 & si\ |pos(x) - pos(y)| \leq D_{cache} \\ 1 & sinon \end{cases}$$

Où $|pos(x) - pos(y)|$ donne la distance (le nombre d'octets) qui sépare x et y. Intuitivement, le défaut de cache porte sur le nœud y : le processeur est sur le nœud x et il essaye d'accéder au nœud y ; si le nombre d'octets qui séparent les deux nœuds est plus petit que la capacité du cache de données, alors il n'y a pas de défaut de cache ($\sigma(x,y) = 0$), sinon il y aura un défaut de cache ($\sigma(x,y) = 1$).

On considère qu'un programme P dans son exécution fait reference à une séquence ordonnée de nœuds $N_k = (n_1, n_2, n_3, ..., n_k)$. Ainsi :

– $CM(N_k) = 1 + \sum_{i=2}^{k} \sigma(n_i, n_{i-1})$ est le nombre de défauts de caches provoqué par une séquence N_k.

Pour des raisons de convénience, on notera $\sigma(x,y)$ par $\sigma(|pos(x) - pos(y)|)$ (pos étant une permutation).

2.3.2. Problème de numérotation des graphes sociaux

Soit $G = (N, E)$ un graphe social (N est l'ensemble des nœuds et E l'ensemble des arêtes). On aimerait trouver une numérotation (permutation) π des nœuds de ce graphe pour qu'un programme s'exécutant sur une séquence ordonnée N_k produise le minimum de défauts de cache (min_{dc}). Autrement dit,

$$Soit\ G = (N,E), min_{dc} \in \mathbb{N}, \begin{cases} - & on\ cherche \quad \pi : \ N \longrightarrow N \\ & \qquad\qquad\qquad\quad n \longmapsto \pi(n) \\ - & pour\ que \quad \sum_i^k \sigma(|\pi(n_i) - \pi(n_{i-1})|) \leq min_{dc}, \\ & avec\ \ N_k = (n_1, ..., n_k)\ et\ n_i \in N. \end{cases}$$

Théorème 2.1 (relatif à la complexité du problème) *Le problème de numérotation des graphes sociaux (PNGS) tel que défini plus haut est NP-complet.*

Démonstration :*Nous nous servons du problème d'arrangement linéaire optimal (PALO) [6]. En rappel, ce problème est défini comme suit :*

$$Soit\ G=(N,A), min \in \mathbb{N} \begin{cases} - & on\ cherche & \pi : & N & \longrightarrow & N \\ & & & n & \longmapsto & \pi(n) \\ - & pour\ que & & \sum_{\{n_i,n_j\}\in A} |\pi(n_i)-\pi(n_j)| \leq min \end{cases}$$

À partir de cette définition, on voit aisement que le PNGS est une instance du PALO où l'on recherche un π permettant d'obtenir le nombre minimum de défauts de cache dans un parcours des nœuds du graphe social.

2.3.3. Numérotation basée sur les communautés

L'algorithme 1 présente une solution basée sur le regroupement en communautés des nœuds des graphes sociaux.

Algorithm 1 Numérotation basée sur les communautés

Entrée : $G = (N, E)$, un graphe social
Sortie : π, un ordonnancement (permutation) des nœuds de G

1: $Com \leftarrow calulDesCommunautes(G)$
2: Calculer π de sorte que les nœuds de la même communauté aient des numéros consécutifs

3: **return** π

La première étape calcule les communautés du graphe social. Nous considérons ici l'algorithme de Louvain [2]. Ce dernier s'exécute en $O(nlogn)$ (avec $n = |N|$). La deuxième étape attribue les numéros des nœuds en faisant en sorte que les numéros des nœuds situés dans la même communauté soient consécutifs. Ce calcul s'effectue en $O(n)$. Ainsi la complexité totale de l'algorithme est de $O(nlogn + n)$.

Une amélioration future de cet algorithme consisterait à classer d'abord les communautés issues de l'algorithme de Louvain par affinité (ce qui n'est pas fait ici).

2.3.4. Quelques propriétés : gain dû à la numérotation

Étant donné un programme P d'analyse des réseaux sociaux, le gain désigne ici la réduction du nombre de défauts de cache provoquée par l'exécution de P lorsque les nœuds ont été stockés en mémoire en suivant l'algorithme de numérotation présenté plus haut.

Propriéte 2.1 (Exploration des nœuds) *Le gain obtenu par cette numérotation depend de l'exploration des noeuds du graphe pendant l'exécution du programme :*

1) Il est grand pour une exploration locale du graphe (à partir d'un nœud, on fait réference aux nœuds situés dans son voisinage).

2) Il est petit pour une exploration non locale du graphe (à partir d'un nœud, on fait réference aux nœuds situés en dehors de son voisinage).
Indications :*Dans le premier cas, comme les nœuds successifs sont plus rapprochés en mémoire, on a moins de défauts de cache ; le gain est donc grand.*

Dans le deuxième cas, les nœuds successifs étant éloignés en mémoire, il y a plus de défauts de cache ; le gain est donc plus petit.

Propriéte 2.2 (Rangement initial des nœuds) *Le gain obtenu par cette numérotation depend du rangement initial (avant l'usage de l'algorithme 1) des noeuds dans le graphe :*

1) Il est grand lorsque les noeuds inter-communautaires (de chacune des communautés) ont des numéros non consécutifs et très éloignés.

2) Il est petit lorsque les noeuds inter-communautaires (de chacune des communautés) ont des numéros consécutifs (gain nul) ou proche (gain petit).

Indications :*Dans le premier cas, le rapprochement des nœuds voisins est effectif. Ainsi dans une exploration locale du graphe, il y aura moins de défauts de cache avec la numérotation générée par l'algorithme 1.*

Dans le deuxième cas, la numérotation générée par l'algorithme 1 sera presqu'identique à la numérotation initiale. Ainsi le gain sera faible.

3. Travaux connexes

Duong et co-auteurs [3] formalisent le problème de répartition des réseaux sociaux sur un système distribué de machines. Ils proposent ensuite un algorithme de répartition qui tire profit de la structure en communautés des réseaux sociaux. Dans notre cas, nous recherchons un algorithme de numérotation des nœuds dans une mémoire (partagée) permettant de réduire les défauts de cache.

Hoque et co-auteurs [8] ont conçu une technique d'organisation du disque dur en se basant sur le regroupement en communautés des données issues des graphes sociaux. Cette technique leur a permis de diminuer le nombre de déplacements de la tête de lecture et ainsi d'améliorer l'accès au disque (48% plus rapide). Nous agissons plutôt sur la mémoire vive (tandis qu'ils agissent sur le disque dur).

Pour améliorer le *prefetching*, Li et co-auteurs [10] utilisent l'algorithme de recherche des itemsets (fréquents fermés) pour fabriquer leurs propres algorithmes (c-miner et c-miner*). Ces algorithmes sont ensuite utilisés pour trouver la corrélation entre les blocs d'une mémoire : les blocs sont perçus comme des items, les règles d'association issues de ces items permettent de faire du *prefetching*. Dans notre cas, nous nous servons de la détection des communautés.

Plusieurs travaux visent l'usage d'une meilleure représentation des matrices creuses pour accroitre les performances de certaines applications (dans la résolution des systèmes d'équations linéaires) :

- Lukas Polok et co-auteurs [13] proposent une structure de données basée sur la représentation en sous-blocs d'une matrice creuse. Cette représentation permet de réduire les défauts de cache lors des opérations arithmétiques effectuées sur la matrice pendant l'exécution.

- Rukhsana S. et Anila U. [15] proposent une représentation en sous-blocs (d'éléments tous non nuls) d'une matrice creuse. Les auteurs montrent que, non seulement cette représentation permet d'économiser plus d'espace, mais aussi permet d'obtenir une meilleur perfomance (lors de la multiplication matrice-vecteur).

Dans notre cas, nous recherchons la représentation mémoire des graphes sociaux la plus appropriée pour réduire les défauts de cache des programmes d'analyse des réseaux sociaux.

4. Évaluation

Les expérimentations ont été menées sur la machine NUMA32, 4 nœuds (*numa node*) de 8 cœurs chacun, soit au total 32 cœurs pour 64 Go. Chaque noeud est de type Intel Xeon avec les caractéristiques 2.27GHz, L1 32KB, L2 256KB, L3 24MB, pas d'*Hyper-Threading*. La figure 4 présente un *numa node*.

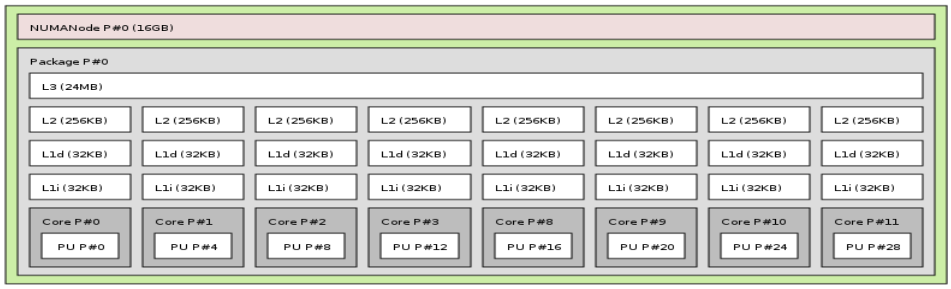


Figure 4 – *Numa node* : 8 cœurs, L1 et L2 privé, L3 partagé

Pour cette évaluation, nous avons implémenté l'algorithme du score de katz [9] avec deux structures de données pour la repésentation des graphes :

– représentation du graphe avec un tableau de nœuds, chacun étant relié à un bloc de ses voisins, la taille du bloc étant variable *(bloc_ds)*

– représentation du graphe sous forme de Yale *(yale_ds)*
Chacune de ces deux structures a ensuite été réorganisée en utilisant l'algorithme d'ordonnancement 1 présentée à la section 2.3.3. Nous notons chacune des structures résultantes par *b_comm_ds* et *y_comm_ds*.

4.1. Algorithme du score de katz

Le score de Katz [9] est utilisé comme une mesure de similarité basée sur les distances entre les nœuds. Le score de Katz entre deux nœuds x et y est donné par la formule :

$$katz\_score(x,y) = \sum_{l=1}^{L} (\beta^l . |paths_{x,y}^{<l>}|) \tag{1}$$

Où :

– L représente la taille maximale d'un chemin.

– $paths_{x,y}^{<l>}$ est l'ensemble des chemins de longueur l entre x et y, et $|paths_{x,y}^{<l>}|$ représente leur nombre.

– $0 < \beta < 1$. β est choisi tel que les chemins avec un l grand contribuent moins à la somme que les chemins avec un l petit.

Nous avons réalisé une implémentation multithreadée de l'algorihme de katz. Les nœuds sont rangés dans une liste chainée qui est ensuite parcourue en parallèle. Pour chaque nœud, la fonction **computeKatzNode** est invoquée (voir algorithme 2).

Algorithm 2 Katz multi-threadé

```
1: Global nodeList, G, β, L, K_sc
2: do_work()
3: while nodeList ≠ ∅ do
4:     x ← atomic_dequeue(nodeList)
5:     K_sc[x] ← computeKatzNode(x, G, β, L)
6: end while
7:
8: main()
9: nodeList ← generate_nodeList(G)
10: for i = 1 to n_threads do
11:     spawn_thread(do_work())
12: end for
13: wait_every_child_thread()
14: output(K_sc)
```

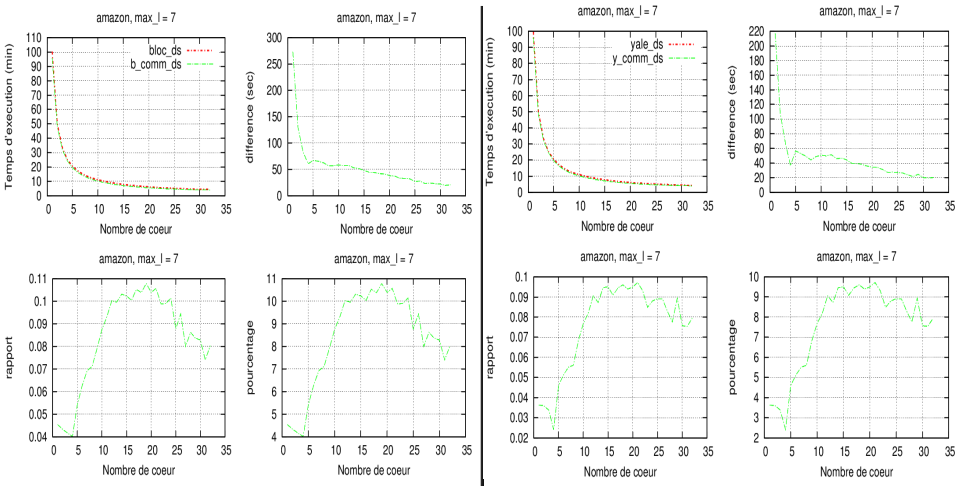
Pour tout nœud x de G, on peut démontrer que les nombres de chemins à partir de ce nœud vers les autres nœuds se calculent ainsi qu'il suit ($N_i,\ et\ L_i$ représentent respectivement les voisins et les nombres de chemins d'ordre i) :

$$\begin{cases} i = 1 & N_i \quad = G.neighbors(x) \\ & L_i[y] = 1, \forall y \in N_i \\ \\ 2 \leq i \leq L & N_i \quad = \{z / z \in G.neighbors(y) \wedge y \in N_{i-1}\} \\ & L_i[z] = \sum_y L_{i-1}[y]_{/\{y \in N_{i-1} \wedge z \in G.neighbors(y)\}} \end{cases} \tag{2}$$

Ceci nous permet d'établir l'algorithme 3. La clé réside dans le calcul du vecteur de nombre de chemins $cLenPath$ à la ligne 6 avec la fonction **updateLenPath**() développée entre les lignes 15 et 26). À chaque valeur de l, avant de passer à la valeur suivante, le score de katz est mis à jour (lignes 7 à 10) ; l'ensemble des nœuds courants et le tableau des nombres de chemins sont également mis à jour (lignes 11 et 12).

Algorithm 3 Score de Katz entre x et tout nœud atteignable avec L

```
1: computeKatzNode(x, G, β, L)
2: dNeig ← G.neighbors(x)
3: pNeig ← dNeig
4: pLenPath[{dNeig}] ← 1
5: for l = 2 → L do
6:     [cNeig, cLenPath] ← updateLenPath(pNeig[ ], pLenPath[ ])
7:     for all (t ∈ cNeig) and (t ∉ dNeig) do
8:         katz[t] ← katz[t] + β^l cLenPath[t]
9:         accessibleNeig.add(t)
10:    end for
11:    [pNeig, cNeigh] ← [cNeig, empty()]
12:    [pLenPath, cLenPath] ← [cLenPath, empty()]
13: end for
14: return buildLign(x, katz[ ], accessibleNeig[ ])
15: updateLenPath(pNeig[ ], pLenPath[ ])
16: for all y ∈ pNeig do
17:    for all z ∈ G.neighbors(y) do
18:        if z ∈ cNeig then
19:            cLenPath[z] ← cLenPath[z] + pLenPath[z]
20:        else
21:            cLenPath[z] ← pLenPath[z]
22:            cNeig.add(z)
23:        end if
24:    end for
25: end for
26: return [cNeig, cLenPath]
```



a-comparaison avec bloc_ds *b-comparaison avec yale_ds*

Figure 5 – Diminution du temps d'exécution – amazon

4.2. Résultats

Pour cette évaluation, nous nous sommes servis des graphes amazon et dblp [16].

4.2.1. Résultats sur amazon

Les nœuds représentent les produits vendus en ligne par le site amazon. Il existe un lien entre deux produits si ceux-ci sont fréquemment achetés ensemble. Les communautés correspondent aux produits appartenant à la même catégorie. Le graphe considéré ici (après suppression des nœuds isolés) a 269906 nœuds et 1851744 arêtes.

Diminution du temps d'exécution : La figure 5 présente les temps d'exécution obtenus sur la machine décrite plus haut. À gauche nous avons la comparaison avec la structure *block_ds* et à droite la comparaison avec la structure *yale_ds*. Nous lisons chaque partie de la gauche vers la droite et de haut en bas. Considérons la première partie (*a-comparaison avec bloc_ds*). Au premier cadrant, la courbe *b_comm_ds* reste en dessous de la courbe *bloc_ds*. Ceci traduit bien le fait que la prise en compte du regroupement en communautés des noeuds dans les structures de données contribue à réduire le temps d'exécution. Dans le deuxième cadrant, on peut voir les différences de temps (en seconde) entre les deux structures. Ces différences se réduisent avec le nombre de cœurs (même variation que les temps d'exécution). Elles sont plus significatives en observant les rapports (dans le troisième cadrant) ou les pourcentages (quatrième cadrant).

La courbe des pourcentages (multiplication par 100 des rapports) croit et atteint une valeur optimale (environ 10.5%) entre 18 et 20 cœurs. Au dela de ce nombre de cœur, les pourcentages commencent à diminuer. Ceci peux s'expliquer par l'architecture utilisée, plus précisement par le cache L3 qui est partagé entre les cœurs : les cœurs profitent de plus en plus des données chargées par les autres. Mais à partir de 20 cœurs, le coup de synchronisation des threads commence à réduire le pourcentage.

Nous faisons des observations similaires dans la deuxième partie (*b-comparaison avec yale_ds*). Cette fois ci, la prise en compte de la structure en communautés permet de réduire le temps d'exécution jusqu'à 9.5%.

Diminution des défauts de cache : Pour vérifier que les temps d'exécution observés étaient liés au nombre de défauts de caches causés par le programme, nous avons lancé le

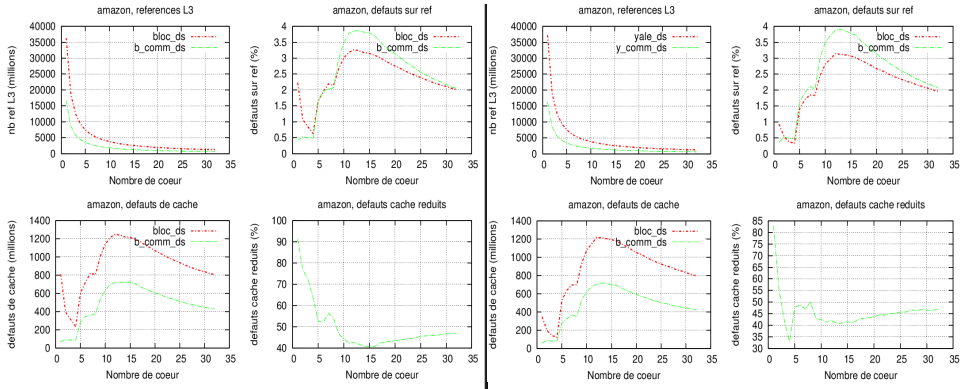


Figure 6 – Diminution des défauts de cache – amazon

programme avec l'outil *perf* [1]. La figure 6 présente les résutats obtenus pour les évenements "cache-references" et "cache-misses".

Dans chaque partie, le premier cadrant correspond aux courbes des évènements "cache-references", le troisième cadrant correspond aux courbes de l'évènements "cache-misses". Le deuxième cadrant correspond au rapport (en %) entre les évènements "cache-misses" et "cache-references". Le quatrième cadrant donne le pourcentage des nombres de défauts de caches réduit grâce à la prise en compte de la structure en communautés.

Considérons la première partie (*a-comparaison avec bloc_ds*). Dans le premier cadrant, la courbe *b_comm_ds* reste en dessous de la courbe *bloc_ds*. Ceci signifie que la prise en compte de la structure en communautés permet de réduire le nombre de références au cache L3. En effet, les données (les nœuds) étant mieux organisées (avec la structure *b_comm_ds*), les caches L2 et L1 se retrouvent plus sollicités. Ceci contribue à moins referencer le cache l3.

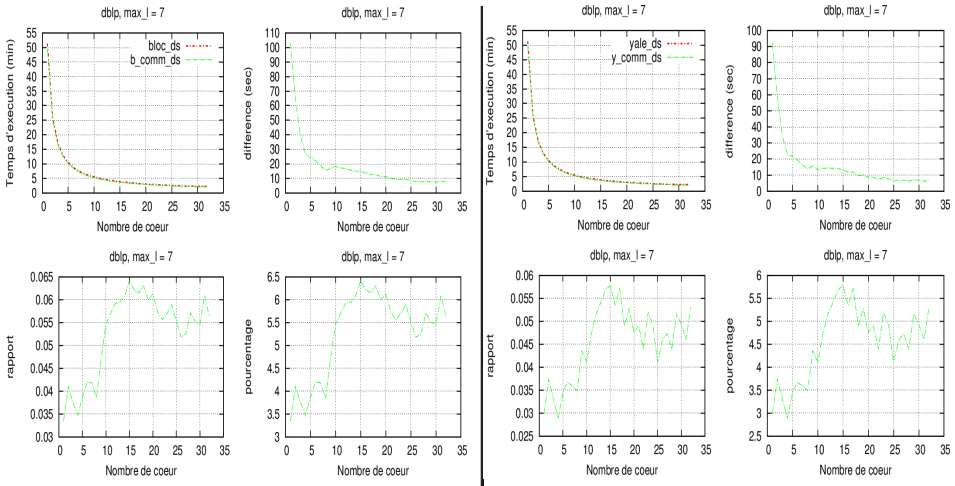
L'effet direct de la réduction du nombre de références au cache L3 est la diminution du nombre de défauts de cache observable au troisième cadrant. Ici, on observe bien que la courbe *bloc_ds* reste au dessus de la courbe *b_comm_ds* ; ce qui montre qu'en prennant en compte la structure en communautés, on réduit le nombre de défauts de cache.

Le quatrième cadrant nous permet de voir qu'on réduit les defauts de caches de 40% (15 cœurs) à 90% (1 cœur). En particulier entre 18 et 20 cœurs, on réduit les défauts de cache de 42%. La courbe semble inversée lorsqu'on la compare avec celle qui donnait les gains en pourcentage des temps d'exécution (figure 5, quatrième cadrant). On s'attend particulièrement à obtenir la meilleure réduction de défauts de cache entre 18 et 20 cœurs car, à ce niveau, on a observé la meilleure réduction du temps d'exécution.

Pour comprendre les deux courbes (gain en temps d'exécution et gain des défauts de cache), il suffit de regarder à nouveau les courbes des défauts de cache (troisième cadrant). On peut remarquer qu'entre 10 et 20 cœurs, le nombre de défauts de cache est plus élévé. La réduction de ce nombre (grâce à la prise en compte de la structure en communautés), même avec un faible pourcentage (40% comparé à 90% avec un cœur) a un plus grand impact sur le temps d'exécution.

Des observations similaires sont effectuées dans la deuxième partie (*b-comparaison avec yale_ds*).

1. `https://perf.wiki.kernel.org/index.php/Tutorial`



a-comparaison avec bloc_ds *b-comparaison avec yale_ds*

Figure 7 – Diminution du temps d'exécution – dblp

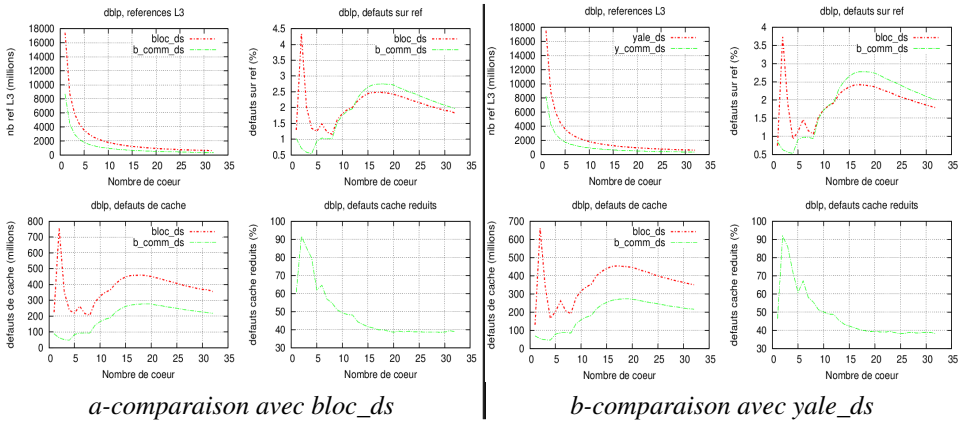
4.3. Résultats sur dblp

Les nœuds correspondent aux auteurs des articles scientifiques en informatique. Il existe un lien entre deux auteurs s'ils ont été co-auteurs d'au moins deux articles. Les communautés correspondent aux auteurs qui ont publié dans un même journal ou dans une même conférence. Le graphe considéré ici (après suppression des nœuds isolés) a 195310 nœuds et 2099732 arêtes.

Diminution du temps d'exécution : La figure 7 présente les résultats les temps d'exécution obtenus.

Les courbes ont les mêmes allures que dans le cas précédent (cas d'amazon). La différence se fait au niveau des performances observées. Avec le jeu de données dblp, la prise en compte de la structure en communauté permet de réduire le temps d'exécution jusqu'à 6.4% par rapport à *bloc_ds* et 5.7% par rapport à *yale_ds*.

Diminution des défauts de cache : Comme dans le cas du jeu de données amazon et comme le montre la figure 8, la diminution des temps d'exécution est aussi liée à la diminution des défauts de cache. Ici, les defauts de cache sont réduits de 39% (entre 18 et 32 cœurs) à 90% (2 cœurs). En particulier entre 18 et 20 cœurs, on réduit les défauts de cache de 39%.



a-comparaison avec bloc_ds *b-comparaison avec yale_ds*

Figure 8 – Diminution des défauts de cache – dblp

La différence de performance avec le jeu de données amazon s'explique par le rangement initial des nœuds (propriété 2.2) : les nœuds inter-communautaires du graphe initial d'amazon ont des numéros plus éloignés que les numéros des nœuds inter-communautaires du graphe initial de dblp. (Pour le vérifier, nous avons calculé la différence de coût $\sum_{(n_i,n_j)} |\pi(n_i) - \pi(n_j)|$ avant et après usage de l'algorithme de numérotation, cette différence était plus élévée avec le graphe amazon ; ce qui traduit un meilleur gain).

4.4. Surcoût induit par l'algorithme de numérotation

L'algorithme de numérotation présenté plus haut a pour vocation d'être exécuté avant un programme d'analyse des réseaux sociaux. Ceci afin d'avoir une organisation des nœuds en mémoire permettant de réduire au maximum les défauts de cache. Les performances du programme sont alors améliorées. Toutefois, pour atteindre cet objectif, il faudra que le programme en question ait une complexité plus élévée que l'algorithme de numérotation. Dans les expérimentations que nous proposons dans ce rapport, l'algorithme de katz a une plus grande complexité ($O(n^3)$ dans le pire des cas, comparé à $O(nlogn)$ de l'algorithme de numérotation).

Plus concrètement, en relevant le temps mis par l'algorithme de numérotation, nous avons obtenu *1462 ms* pour amazon et *1282 ms* pour dblp (voir tableau 2). Ce qui est relativement petit par rapport au gain obtenu. Par exemple avec le jeu de données amazon (voir tableau 3), en comparant avec la structure *bloc_ds*, on a obtenu le gain *273263 ms* sur 1 cœur, *41529 ms* sur 19 cœurs et *20727 ms* sur 32 coeurs (ce qui correspondent à environ *187 fois*, *28 fois* et *14 fois* le temps perdu).

Jeu de données	amazon	dblp
Temps algo. numérotation	1462 ms	1282 ms

Tableau 2 – Temps d'exécution de l'algorithme de numérotation

	amazon			dblp		
nb de cœurs	1 cœur	19 cœurs	32 cœurs	1 cœur	19 cœurs	32 cœurs
sans numérotation (ms)	6020869	384892	258333	3079606	190321	131220
avec numérotation (ms)	5747605	343363	237606	2976453	178911	123838
Gain (ms)	273264	41529	20727	103153	11410	7382
(n fois coût numérot.)	***186.9 fois***	***28.4 fois***	***14.2 fois***	***80.5 fois***	***8.9 fois***	***5.7 fois***

Tableau 3 – Gain observé avec *b_comm_ds*

5. Conclusion

Dans cet article, il était question de voir comment exploiter la structure en communautés pour diminuer les défauts de cache et le temps d'exécution des programmes de fouille des réseaux sociaux. Nous avons proposé une numérotation des nœuds permettant de tenir compte de cette struture en communautés des graphes sociaux. Des expérimentations effectuées sur le score de katz avec les jeux de données amazon et dblp ont montré que les performances sont améliorées qu'on utilise cette numérotation.

En perspective, nous envisageons d'intégrer cet algorithme dans les langages dédiés et les plates-formes d'analyse de graphes afin d'améliorer leurs performances. Une autre perspective est de trouver un moyen de classer les communautés (par affinité) avant la numérotation des nœuds ; ceci pourrait contribuer à diminuer d'avantage les défauts de cache et donc le temps d'exécution.

Références

[1] David A Bader, Jonathan Berry, Adam Amos-Binks, Daniel Chavarría-Miranda, Charles Hastings, Kamesh Madduri, and Steven C Poulos. Stinger : Spatio-temporal interaction networks and graphs (sting) extensible representation. *Georgia Institute of Technology, Tech. Rep*, 2009.

[2] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics : Theory and Experiment*, 2008(10) :P10008, 2008.

[3] Quang Duong, Sharad Goel, Jake Hofman, and Sergei Vassilvitskii. Sharding social networks. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 223–232. ACM, 2013.

[4] David Ediger, Robert McColl, E. Jason Riedy, and David A. Bader. Stinger : High performance data structure for streaming graphs. *HPEC*, pages 1–5, 2012.

[5] Stanley C Eisenstat, MC Gursky, MH Schultz, and AH Sherman. Yale sparse matrix package. i. the symmetric codes. Technical report, DTIC Document, 1977.

[6] Michael R Garey, David S. Johnson, and Larry Stockmeyer. Some simplified np-complete graph problems. *Theoretical computer science*, 1(3) :237–267, 1976.

[7] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl : a dsl for easy and efficient graph analysis. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 349–362. ACM, 2012.

[8] Imranul Hoque and Indranil Gupta. Social network-aware disk management. 2010.

[9] Leo Katz. A new status index derived from sociometric analysis. *Psychometrica-vol.18, No.1*, 18(1), March 1953.

[10] Zhenmin Li, Zhifeng Chen, and Yuanyuan Zhou. Mining block correlations to improve storage performance. *ACM Transactions on Storage (TOS)*, 1(2) :213–245, 2005.

[11] Thomas Messi Nguélé, Maurice Tchuenté, and Jean-François Mehaut. Exploitation de la structure en communautés pour la réduction de défauts de cache dans la fouille des réseaux sociaux. In *Conférence de Recherche en Informatique (CRI)*, Yaoundé, Cameroon, December 2015.

[12] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, 2013.

[13] Lukas Polok, Viorela Ila, and Pavel Smrz. Cache efficient implementation for block matrix operations. In *Proceedings of the High Performance Computing Symposium*, page 4. Society for Computer Simulation International, 2013.

[14] Jason Riedy, David A. Bader, and Henning Meyerhenke. Scalable multi-threaded community detection in social networks. *IEEE Computer Society Washington, DC, USA 2012*, 18(1) :1619–1628, 2012. IPDPSW '12 Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum.

[15] Rukhsana Shahnaz and Anila Usman. Blocked-based sparse matrix-vector multiplication on distributed memory parallel computers. *Int. Arab J. Inf. Technol.*, 8(2) :130–136, 2011.

[16] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1) :181–213, 2015.