École nationale Supérieure d'Informatique (ESI ex. INI)

Rapport de TP n°1

Etude comparative entre les algorithmes de Kosaraju et de Tarjan pour déterminer les composantes fortement connectées d'un graphe orienté

Réalisé par :
CHENNI Nidhal Eddine

Table des matières

1. Introduction

Trouver les composants fortement connectés (CFCs) dans un graphe est un problème de graphe très naturel avec de nombreuses applications en informatique. Une composante fortement connectée dans un graphe est un ensemble maximal de sommets où il existe un chemin entre chaque sommet de l'ensemble et chaque autre sommet de l'ensemble.

Le but de ce travail est d'établir une étude comparative entre deux célèbres algorithmes de calcul de composantes fortement connectées, pour cela nous avons réalisé des expérimentations inspirées de recherches antérieures [1], [2], [3] , nous avons implémenté les deux algorithmes en C++ et ensuite , nous avons comparé et analysé les algorithmes en utilisant la représentation dynamique du graphe (liste d'adjacence). Pendant l'expérimentation, nous avons trouvé des résultats intéressants, en particulier l'efficacité de l'algorithme de Tarjan par rapport à l'algorithme de Kosaraju en termes de temps et de consommation de mémoire pour le calcul des composants fortement connectés.

2. Présentation des algorithmes de Kosaraju et Tarjan

Comme dans [2], les algorithmes séquentiels pour trouver les composants fortement liés peuvent être considérés comme un problème résolu, et une grande partie de la recherche est acceptée comme optimale. Bien qu'il existe de nombreux algorithmes séquentiels qui résolvent le problème en une complexité de temps O(|V| + |E|), nous nous concentrerons sur deux des algorithmes les plus populaires, l'algorithme de Kosaraju et l'algorithme de Tarjan.

L'algorithme de Kosaraju a une implémentation plus simple que celui de Tarjan, mais nécessite deux passages du graphe alors que celui de Tarjan ne nécessite qu'un seul passage.

2.1 Algorithme de Kosaraju

L'algorithme de Kosaraju est l'une des méthodes les plus simples pour trouver les CFCs d'un graphe en temps linéaire. Il utilise pour cela l'observation que les CFCs de G et G^T sont identiques. L'algorithme nécessite deux passages du graphe, le premier pour effectuer un tri topologique et le second pour trouver les CFCs. Dans chaque DFS, nous marquons les nœuds que nous avons précédemment visités et nous les marquons non-visités après avoir recherché toutes leurs arcs sortants.
Considérons le psuedo-code suivant : (entièrement inspiré de [2])

Algorithme 1 Kosaraju

```
1  Proc dure Kosaraju(G, b)
2   calculer GT
3   Pour tout v dans V en diminuant vf: #calculer vd, vf
4       C <- DFS(G^T, v) #DFS a v, mettre les noeuds visites dans C
5       ajouter C en tant que CFC et le retirer du GT
6   fin Pour
7   retourner tous les CFC trouves
8  fin Proc dure
```

2.2 Algorithme de Tarjan

L'algorithme de Tarjan améliore le principal inconvénient de l'algorithme de Kosaraju, qui nécessite de multiples passages du graphe. Au lieu de cela, l'algorithme de Tarjan peut calculer les CFCs avec un seul passage DFS tout en maintenant l'état sur une pile séparée. L'algorithme de Tarjan est l'algorithme séquentiel le plus populaire pour trouver les CFCs en raison de sa vitesse accrue et de sa mise en œuvre relativement simple. son implémentation relativement simple.

L'algorithme de Tarjan est l'un des algorithmes optimaux pour trouver les CFCs de manière séquentielle dans un graphe orienté.

Nous allons maintenant examiner le pseudo-code de l'algorithme de Tarjan : (entièrement inspiré de [2])

Algorithme 2 Tarjan

```
Proc dure Tarjan(G, b)
 i <- 0
 creer une pile
 Proc dure DFS(v)
     vlow <- i
     vd <- i
     i <- i + 1
     placer v sur la pile
     pour tous w voisins de v en sortie
         si wd est nil alors # w pas encore visite
         DFS(w)
         vlow = min(vlow, wlow)
         sinon si wd < vd alors
               vlow=wlow
         fin si
     fin pour
         si vlow = vd alors . v est la racine du composant
         creer CFC
         TQ u = sommet(pile) s.t ud >= vd faire
             retirer u de la pile
             ajouter u dans nouveau CFC
         fin TQ
     fin si
 fin Proc dure
 TQ v dans V et vd est nil faire
   DFS(v)  Appliquer sur les sommets non visit s
 fin TQ
fin Proc dure
```

3. Expérimentations

Les expérimentations se sont inspirées de [1], trois valeurs de densité de graphe ont été utilisées (graphe clairsemé (non dense) : 1k arcs, moyennement dense : 5k arcs, dense : 10k arcs), et nous avons changé le nombre de nœuds entre 1k et 10k, puis un benchmark a été conçu pour générer six graphes de la même taille (même nombre d'arcs et de sommets) avec des connexions aléatoires et ensuite mesurer la moyenne des performances avec un algorithme spécifique sur ces six graphes.

Dans nos expériences, nous avons utilisé une structure de données utilisant une liste liée pour la liste d'adjacence. Nous utilisons un processeur intel® Core™ i7-7500U @ 2.70GHz

avec 32GB de RAM.

3.1 Expérimentation avec l'algorithme de kosaraju

Dans ces expériences, un ensemble de graphes aléatoires pour chaque graphe (dense, moyen, clairsemé) est généré. La figure 1 montre la différence de temps d'exécution entre chaque graphe sur un nombre N de nœuds.

L'algorithme de Kosaraju calcule les composantes fortement connectées de manière efficace avec l'augmentation du nombre de nœuds ou du nombre d'arcs. Ainsi, les arcs ont un impact direct sur son temps d'exécution.

3.1.1 Temps de calcul moyen

La figure 1 présente les résultats générés sur différents graphes. Il ressort clairement des figures qu'avec l'augmentation du nombre de nœuds et d'arcs, l'algorithme de Kosaraju prend plus de temps à s'exécuter.

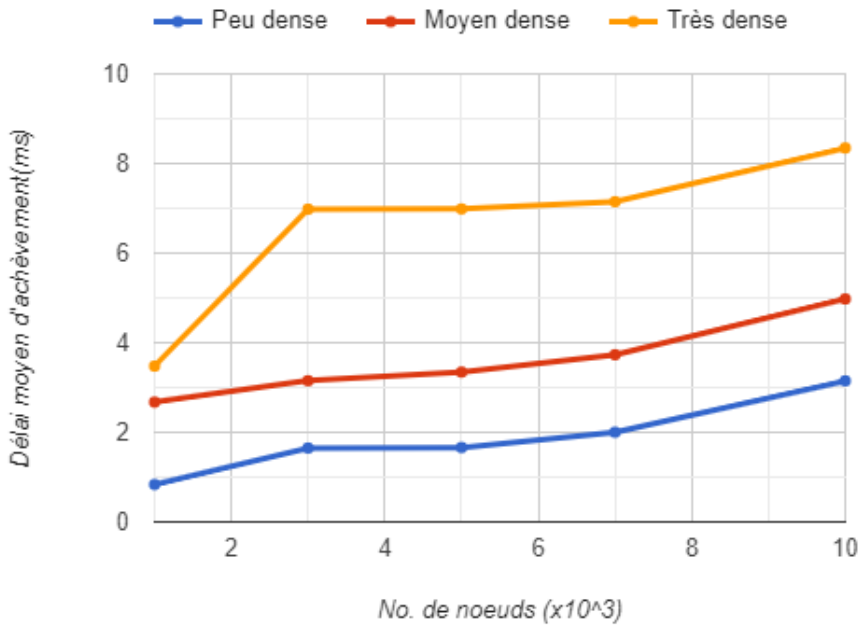

FIGURE 1 – Temps d'exécution sur les graphes denses et épars avec kosaraju

3.1.2 Mémoire utilisée

En passant encore une fois par l'algorithme de Kosaraju, nous remarquerons que pendant le calcul des composants fortement liés, il utilise une table pour marquer les nœuds visités et aussi une pile qui contient les nœuds traversés et enfin un graphe transposé. Si nous voulons approximer une formule pour la consommation de la taille de la mémoire et en ignorant toutes les variables intermédiaires telles que celles utilisées comme index pour les boucles ou les tables, nous arrivons à cette formule :

```
1  taille = taille tableau "visited" + taille pile + taille du graphe
```

plus précisément dans notre cas :

```
taille = nbV x [sBool + 2 x sInt + sPoint] + nbE x [sInt + sPOint]
```

Avec :
nbV, nbE : nombre de nœuds, d'arcs respectivement
sInt, sBool, sPoint : taille d'un entier, boolean, pointeur respectivement

Il est clair que la taille de la mémoire changera proportionnellement au changement de la taille et de la densité du graphe.

3.2 Expérimentation avec l'algorithme de Tarjan

Nous avons eu la même série d'expériences pour l'algorithme de Tarjan e.i. un ensemble de graphes aléatoires pour chaque graphe (dense, moyen, clairsemé) est généré, Nous avons calculé leur temps d'exécution moyen et leur stockage en mémoire comme le montre la figure 2, qui montre la différence entre les graphes denses, moyen et clairsemé sur un nombre N de nœuds. L'algorithme de Tarjan calcule les composantes fortement connectées de manière efficace lorsque le nombre d'arcs est plus faible. Les arcs ont donc un impact direct sur son temps d'exécution.

3.2.1 Temps de calcul moyen

Dans la figure 2, un graphique linéaire est utilisé pour présenter les résultats générés qui montrent que avec l'augmentation du nombre de nœuds et d'arcs l'algorithme composant fortement connectées de Tarjan prend plus de temps à s'exécuter.

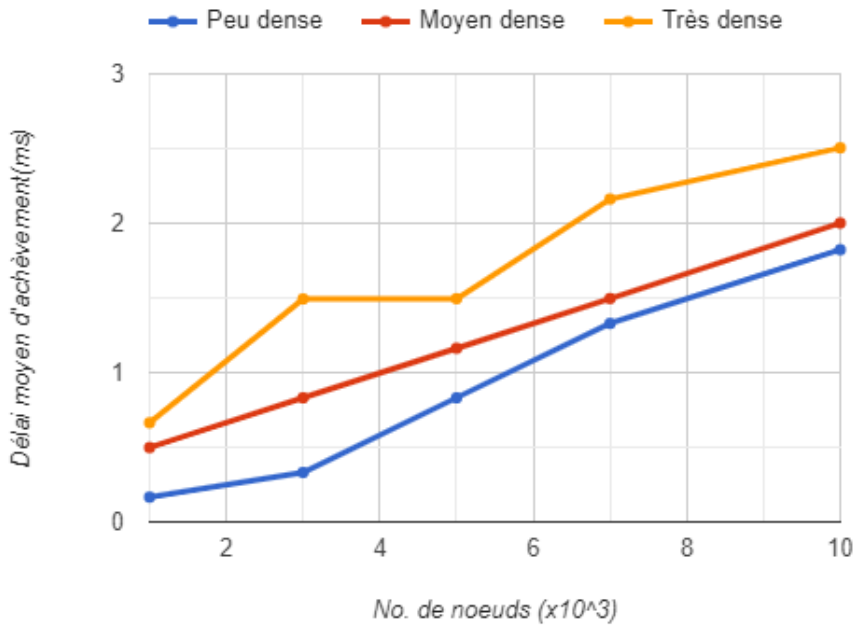

FIGURE 2 – Temps d'exécution sur les graphes denses et épars avec Tarjan

3.2.2 Mémoire utilisée

Comme nous l'avons fait pour kosaraju, nous allons essayer de faire la même chose et d'estimer une fonction pour calculer l'utilisation de la mémoire par l'algorithme tarjan , comme nous jetons un coup d'oeil sur l'algorithme, nous avons constaté qu'il utilise trois tables de la taille des nœuds et une pile , nous allons donc arriver à la formule ci-dessous :

```
taille = 2 x nbV x sInt + nbV x sBool + nbV x sInt + nbV x sPoint
```

Avec :
nbV : nombre de nœuds
sInt, sBool, sPoint : taille d'un entier, boolean, pointeur respectivement

Il est clair que la taille de la mémoire utilisée changera proportionnellement au changement de nombre de noeuds.

3.3 Comparaison sur le temps de calcul moyen

Les mêmes données sont utilisées pour calculer le temps d'exécution moyen pour chaque graphe. Les données sont également combinées pour obtenir des données uniques qui sont utilisées pour comparer les algorithmes de Kosaraju et de Tarjan. Le temps d'exécution moyen est calculé sur un graphe clairsemé (arcs=1000) et un graphe dense (arcs=10k). Les figures 3 et 4 montrent les statistiques obtenues pendant les expériences sur les deux algorithmes. Les performances des deux algorithmes sont remarquables ; l'algorithme de Tarjan prend moins de temps de calcul et de variation que l'algorithme de Kosaraju.

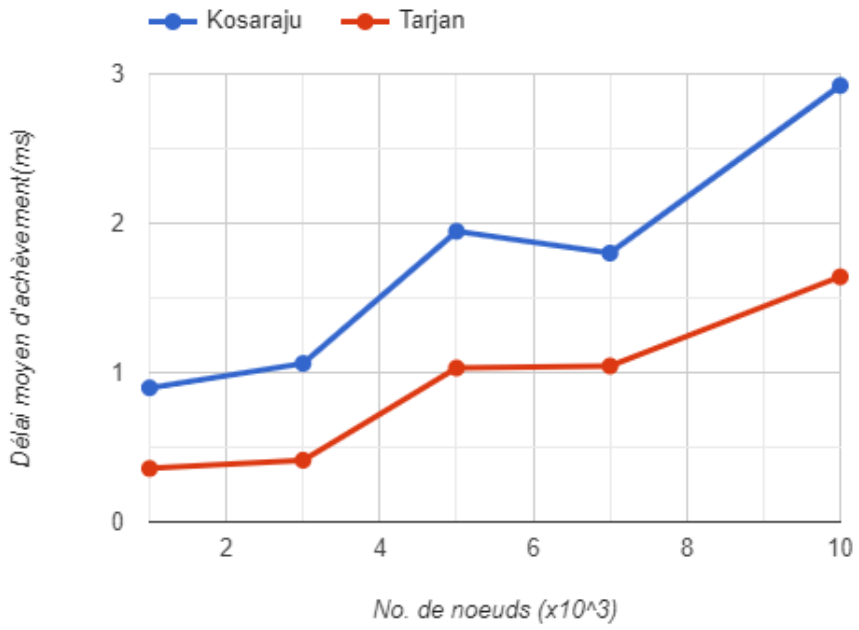

FIGURE 3 – Temps d'exécution sur les graphes clairsemés avec Kosarju et Tarjan

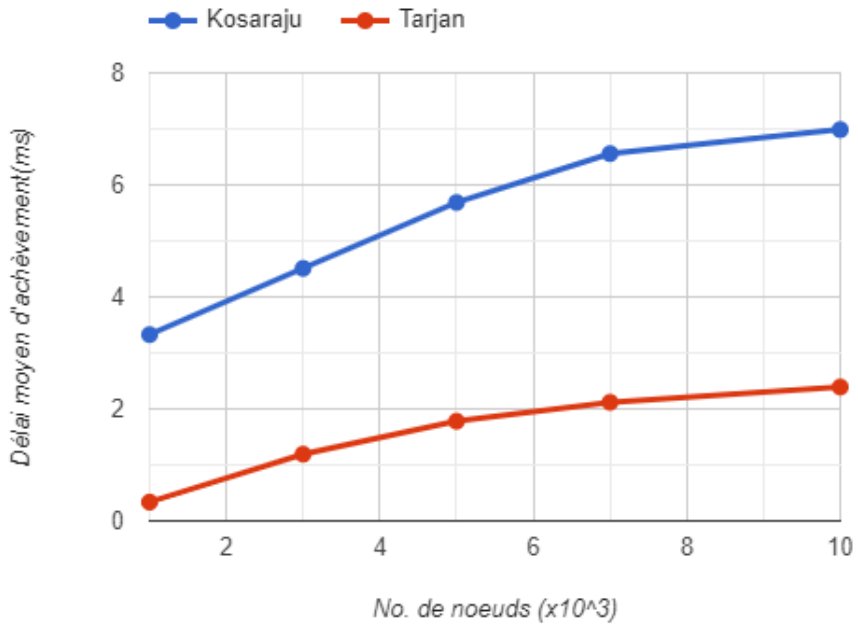

FIGURE 4 – Temps d'exécution sur les graphes denses avec Kosarju et Tarjan

3.4 Comparaison sur la mémoire utilisée

Les mêmes expériences que pour le calcul du temps ont été faites pour la consommation de la taille de la mémoire en utilisant les deux formules développées précédemment, les deux graphiques ci-dessous montrent la différence entre les deux algorithmes dans les cas de graphes clairsemés et denses. Pour la consommation de la mémoire, nous observons sur les

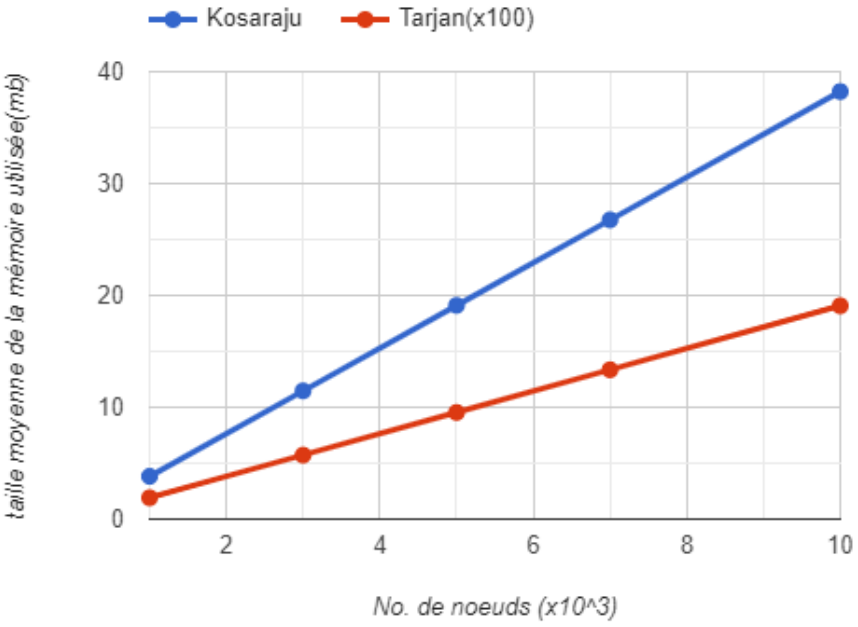

FIGURE 5 – Mémoire utilisée pour les graphes clairsemés avec Kosarju et Tarjan

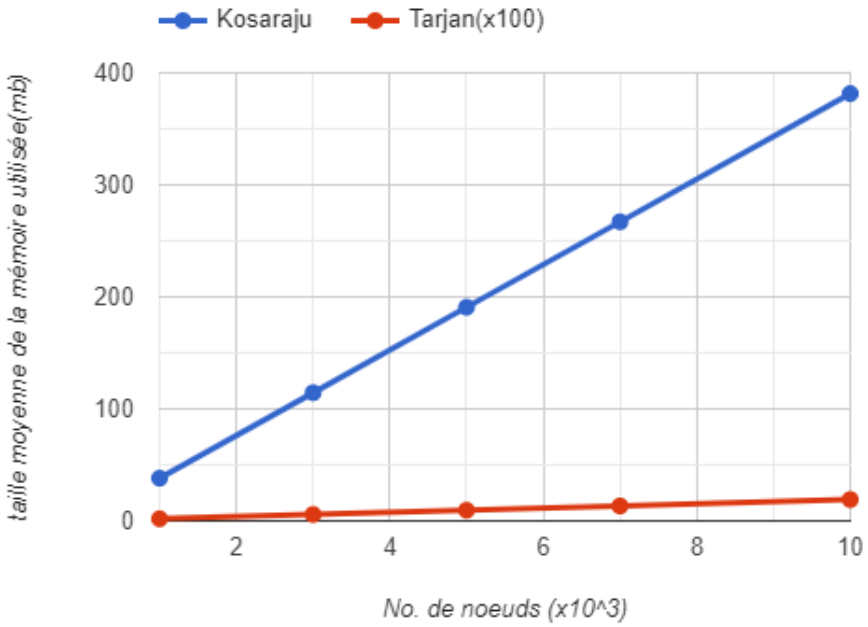

FIGURE 6 – Mémoire utilisée pour les graphes denses avec Kosarju et Tarjan

courbes que l'algorithme de Tarjan est bien meilleur que celui de Kosaraju. Comme nous l'avons noté précédemment, l'algorithme de Kosaraju crée à chaque fois un graphe transposé du graphe original, ce qui prend beaucoup d'espace, surtout lorsque le graphe a beaucoup d'arcs et de nœuds.

4. Conclusion

Au cours du travail précédent, nous avons analysé et comparé les algorithmes de composants fortement connectés de Kosaraju et de Tarjan afin de déterminer s'ils conviennent à diverses applications. Nous avons produit des graphes denses et clairsemés de manière aléatoire pour calculer la mémoire et la différence de temps des deux algorithmes. Nous avons constaté que l'algorithme de Kosaraju prend plus de temps et de mémoire que l'algorithme de Tarjan sur les graphes denses et clairsemés.

Bibliographie

[1] Saleh Alshomrani and Gulraiz Iqbal. An extended experimental evaluation of scc (gabows vs kosarajus) based on adjacency list. *Global Journal of Computer Science and Technology*, 2013.

[2] Andrew Gamble. Survey of strongly connected components algorithms. 2018.

[3] D. Frank Hsu, Xiaojie Lan, Gabriel Miller, and David Baird. A comparative study of algorithm for computing strongly connected components. pages 431–437, 2017.