Théorie des Graphes

Pierre Tassel

Juin 2018

1 Introduction

Ce rapport explique les difficultés et les différents choix effectués lors de la réalisation du projet surs les trois algorithmes de détection des Composants Fortement Connexe.

2 Algorithme Préféré Conceptuellement

Conceptuelle-ment et à faire à la main c'est l'algorithme de Kosaraju que je préférais.
C'est le plus simple à appliquer à la main et c'est celui qui semble le plus logique et donc le plus simple à comprendre.
Paradoxalement c'est celui qui m'a causé le plus de soucis à implémenter, ma DFS non récursive m'a causé bien des soucis et il m'a fallu la débugger plusieurs fois. La suppression dans la pile fût facile, au début j'ai voulu éviter l'opération de dépilement, rempilement sur une nouvelle pile en utilisant un tableau des sommets supprimé mais, cela ne marchant pas, j'ai opté pour la solution basique de dépilement-rempilement de la pile excepté de l'élément souhaité. Inverser et supprimer un élément du graphe n'est pas très compliqué, il faut juste pas se faire piéger à inverser les booléens dans la matrice d'adjacence, mais bien inverser le sens des arcs.
Tarjan est lui aussi agréable à appliquer à la main, mais Gabow est, je trouve, relativement complexe à effectuer à la main.

3 Algorithme Préféré à Implémenter

Tarjan a été celui que j'ai implémenté le plus facilement, il a marché du premier coup.
J'ai eu un léger bug dans Gabow, j'avais oublié d'ajouter le sommet par ou l'on commençait la visite dans la CFC si elle n'avait pas déjà été ajouté dans une CFC.

4 Choix d'Implémentation

J'ai créé une classe Abstraite Graphe dont les deux implémentations Graphe_Adjancence et Graphe_Liste dérive, cela me permet d'éviter d'avoir à implémenter plusieurs fois les algorithmes de CFC pour chaque implémentation des Graphes.
De même chaque algorithme de détection de CFC implémente la classe CFC_Implémentation ce qui permet d'avoir une interface unifiée pour les différents algorithmes au travers de la méthode std::vector¡std::set¡int¿¿ * CFC(Graphe * graphe).
Parmi les choix d'implémentations, j'ai implémenté le graphe de liste d'adjacences, avec un ensemble (set) et non un vector ou une liste. Car cela à deux avantages:

1. Cela évite d'avoir à vérifier que l'arc à été ajouté plusieurs fois
2. Cela permet d'avoir les noeuds voisins dans l'ordre croissant

Je n'ai pas implémenté la matrice d'incidence, car comme nous l'avions vu en cours, elle n'est pas du tout optimale en temps. Cependant, implémenter une nouvelle représentation n'est pas bien complexe, car il suffit d'hériter de la classe abstraite Graphe et d'implémenter les fonctions suivantes:

- void ajouterLien(int a, int b)
- std::stack¡int¿ dfs(int debut, bool * visitee)
- std::vector¡int¿ voisins(int a)
- Graphe * transposer()
- void supprimerSommet(int noeud)

5 Méthode De Test Des Temps d'Exécutions

La classe GrapheGenerator contient deux méthodes permettant de générer soit un graphe de liste d'adjacence, soit un graphe de matrice d'adjacence contenant 500 sommets et un nombre de liens à fournir en paramètre, qui sont ajoutés de manière aléatoire.
Le fichier main contient 6 méthodes, chacune générant un graphe d'un certain type (matrice ou liste d'adjacence) et utilisant un des trois algorithmes de détection des CFC. Elle fait varier le nombre d'arêtes de 500 à 40 000 en augmentant de 10 arrêtes à chaque fois et ont exécute 10 fois d'affiler l'algorithme de détection des CFC sur le même graphe en calculant le temps d'exécutions moyens.
On fait tourner l'algorithme et on re-dirige la sortie standard vers un fichier.
On tracera les courbes au travers d'un script Gnuplot qui tracera le temps médian pour chaque nombre d'arêtes.

6 Comment lancer le programme ?

Il suffit de compiler le programme en allant dans le dossier "cmake-build-release" puis d'éxécuter la commande make qui générera l'éxécutable "graphes".
Quand on lance le programme sans arguments, un manuel expliquant les différents arguments apparaîssent pour lancer les différents tests de durée d'algorithmes.
Si on veut tout lancer d'un coup, on peut éxécuter le script "run.sh" qui se trouve à la racine du dossier, il lance tous les tests de temps et sauvegarde les temps d'exécutions dans le dossier "temps_executions".
Les scripts pour tracer les courbes avec GnuPlot se trouvent dans le dossier "temps_executions" est se termine par l'extension plt. Le dossier "Graphes" contient les différentes implémentations des graphes.
Le dossier CFC contient les différents algorithmes de CFC.

7 Graphes Temps d'Exécutions

7.1 Graphe Liste Adjacence

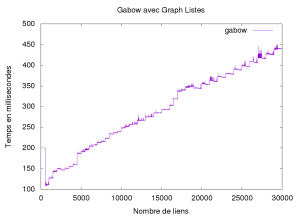

Figure 1: Gabow

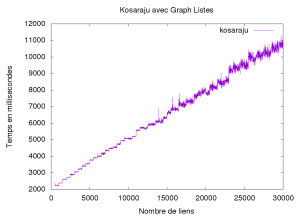

Figure 2: Kosaraju

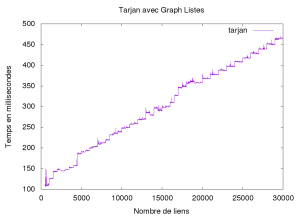

Figure 3: Tarjan

7.2 Graphe Matrice Adjacence

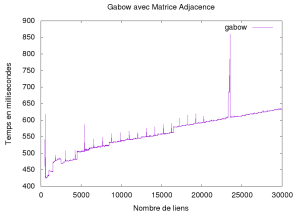

Figure 4: Gabow

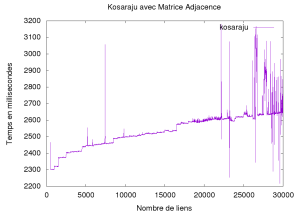

Figure 5: Kosaraju

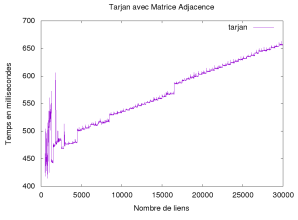

Figure 6: Tarjan

8 Conclusions

L'implémentions la plus stable est celle avec la liste d'adjacence, c'est aussi la plus rapide. Les deux algorithmes les plus rapides sont Gabow et Tarjan.
On remarque que certains cas particuliers apparaissent avec la matrice d'adjacence, surtout avec Kosaraju. C'est notamment du au fait que la complexité pour récupérer tous les voisins d'un sommet u est de O(V) pour la matrice d'adjacence alors qu'elle n'est de O(deg(u)) pour la liste d'adjacence.
Donc si il y a beaucoup de CFC, alors on doit lancer beaucoup de fois la DFS, qui a un coût plus important avec la matrice d'adjacence.