

Rapport de Tests – Projet C/C++

*Projet C/C++*

Sujet : « Réseaux sociaux : Recherche de composantes fortement connexes »

DREYER Quentin

JAMBET Pierre

NGUYEN Michael

Introduction :

Dans le cadre du projet de Langage C, nous avons effectué de nombreux « tests ». Nous détaillerons dans ce document, la nature de ces tests ainsi que leur intérêt pour le projet.

Voici la liste des tests que nous avons réalisés :

* Tests du programme sur une grande batterie de fichiers générés

Ces tests avaient pour but d’analyser les temps d’exécution de notre programme en fonction du fichier analysé.

* Analyse des fuites mémoires et profilage avec VALGRIND
* Tests Unitaires réalisés avec le framework Google TEST

# Statistiques sur de nombreux tests

A l’aide d’un simple script shell (voir ci-dessous) nous avons récolté des informations qui nous ont permis de réaliser des statistiques sur les performances du programme.

|  |
| --- |
| #!/bin/bash |
| # Script pour tester le programme avec pleins de fichiers generes. |
| # Premiere boucle : i => nombre de sommets |
| # Deuxieme boucle : j => nombre de relations |
| # Troisieme boucle : k => nombre de questions |
| date >> ../test/log |
| for (( i = 200; i <= 2000; i+=200 )); do |
| ibis=$(($(($i\*$i))-$i)) |
| echo -n $i " "; |
| for j in 5 7 10 100 200 500 750 1000; do |
| nb\_rel=$(($(($j\*$ibis))/1000)) |
| for k in 5 7 10 100 200 500 750 1000; do |
| nb\_quest=$(($(($k\*$ibis))/1000)) |
| for (( l = 1; l < 5; l++ )); do |
| ./Lager --log $i $nb\_rel $nb\_quest >> ../test/log |
| done |
| done |
| done |
| done |
| echo ""; |

Ce script consiste à générer des graphes aléatoirement des graphes de taille croissante. Pour un même triplet (nombre de personnes, nombre de relations, nombre de questions), on génèrera 4 graphes, ceci dans le but de pouvoir calculer une moyenne de temps d’exécution.

Cependant, nous n’avons malheureusement pas pu achever l’exécution de ce script par manque de temps. Le temps maximal ne devrait donc cependant pas excéder 20 minutes, temps d’exécution pour un graphe de 2000 personnes, avec environ 4 000 000 de relations et environ 4 000 000 de questions possibles.

Nous avons exploité les résultats fournis par l’exécution de ce script pour dresser des graphiques montrant clairement le temps d’exécution en fonction des paramètres.

**Graphique concernant le temps de calcul des CFC**

**Graphique concernant le temps de calcul des calculs des plus courts chemins**

# Utilisation de VALGRIND

L’utilisation de Valgrind nous a permis de tester la fiabilité et la robustesse de notre code. En utilisant le module memcheck nous avons pu vérifier le bon fonctionnement de notre programme au niveau des allocations mémoire et donc de la restitution de la mémoire. Pour effectuer de nombreux tests nous avons utilisé un script se basant sur la figure 1, mais dont la ligne principale est modifiée.

valgrind ./Lager --log $i $j $k >> log 2>&1

Ceci permet de rediriger les messages de Valgrind dans le fichier log, nous avons ainsi pu voir en parcourant ce fichier qu’à chaque exécution du programme, chaque zone mémoire allouée est libérée.

# Tests Unitaires avec Google Test

Les tests unitaires rentrent dans la catégorie des tests fonctionnels. Ces tests permettent de valider le bon fonctionnement du programme. En écrivant ces tests au fur et à mesure du développement nous avons pu conserver un code solide et facilement « *débuggable* ». En effet après une série de modifications sur le code, un lancement de test unitaire permet d’isoler immédiatement la source du problème, car chaque fonctionnalité importante est testée indépendamment des autres.

Afin de créer un code robuste et facile à modifier, nous avons choisi d’utiliser un framework permettant de mettre en place des tests unitaires. Nous avons choisi le framework Google Test, pour sa simplicité d’utilisation. Ainsi à chaque modification du code, nous pouvions utiliser les tests pour vérifier que chaque fonction se comportait correctement, indépendamment du reste du programme.

Un test s’écrit simplement de la manière suivante :

*Sur cet exemple, on teste premièrement que la condition cond est vraie et on teste ensuite l’égalité des valeurs de val1 et val2.*

TEST(nom\_categorie\_test, nom\_test) {  
 ASSERT\_TRUE(cond);  
 EXPECT\_EQ(val1, val2);  
}

On a donc pu écrire une succession d’assertions vérifiant les propriétés de nos fonctions.

Chaque fichier possède son fichier associé contenant ses tests unitaires, nous avons nommé chaque fichier de la manière suivante « <nom\_fichier>\_unittest ».

Voici un exemple concret de tests unitaires.

|  |
| --- |
| TEST(AdjList, sortDescEnd) { |
| AdjList Al; |
| Graph Gr; |
| Gr.setStructGraph('l'); |
| string s = "../test/gene-ut"; |
| Generator Ge; |
| Ge.changeOptionAutomatic( "../test/noms.dat", "../test/gene-ut",100, 100, 50); |
| Ge.generateFile(); |
| vector< s\_summit > tabSummit; |
| vector< vector< int > > listFriends; |
| vector< vector< int > > listDualFriends; |
|  |
| Gr.initGraph(s); |
| tabSummit = Gr.getTabSummit(); |
| listFriends = Gr.getListFriends(); |
| listDualFriends = Gr.getListDualFriends(); |
|  |
| Al.initData(tabSummit, listFriends, listDualFriends); |
| Al.DFS(); |
| Al.sortDescEnd(); |
| for(int i = 0; i < (int)Gr.getSizeGraph() - 1; i++ ) { |
| EXPECT\_GT(Al.getTabSummit(i).end, Al.getTabSummit(i + 1).end); |
| } |
| } |

La partie important de ce test est situé à la fin, la partie soulignée. On parcourt le tableau tout en vérifiant que chaque temps de fin soit supérieur à celui qui le suit (clause *EXPECT\_GT* pour *Greater Than*).