Etienne ALEXANDRE
Léo BUZELIN

Compte-rendu

Nous soussignés ALEXANDRE Etienne et BUZELIN Léo déclarons sur l'honneur que ce projet est le résultat de notre travail personnel et que nous n'avons pas copié tout ou partie du code source d'autrui afin de le faire passer pour le nôtre.

Guide Utilisateur

Après compilation du projet, il est possible de lancer le programme principal : main.

Pour cela, il faut utiliser la commande : « ./main *options* ».

Les différentes options de lancement sont les suivantes :

-h ou --help permet d'afficher un court message permettant de rappeler les commandes disponibles.

-f ou --file suivi d'un nom de fichier permet d'indiquer au programme le fichier à utiliser pour remplir le Bloom Filter. Cette option est obligatoire pour le fonctionnement du programme.

-k permet de préciser le nombre de fonctions de hachage utilisées (k = 3 par défaut).

-m permet de donner le nombre de bit dans le bit array (m = 8000 par défaut).

Après exécution du programme, 2 lignes sont affichées dans le terminal :

La première indique le ratio de faux positifs sur l'ensemble des mots testés.

La seconde indique le ratio théorique calculé à partir des différents paramètres (k et m).

L'ensemble des fichiers utilisés pour les tests se trouve dans le répertoire « words » du projet.

Collaboration au sein du binôme

La collaboration s'est un peu fait naturellement, le travail était réparti en fonction de ce que l'autre avait déjà produit afin de suivre le fil conducteur du projet.

Pour avoir une idée un peu plus concrète de ce sur quoi chaque membre a travaillé :

Etienne s'est occupé de la structure bitarray (avec l'amélioration des bits), une partie du filter, et des comparaisons en temps et en espaces.

Léo s'est chargé du filter, du programme principal et de l'étude des faux positifs.

Etienne ALEXANDRE
Léo BUZELIN

Problématique et résultats

Etat actuel du projet

Des erreurs trouvées via Valgrind indiquant des variables utilisées non initialisées sont présentes dans le programme, néanmoins, nous n'avons pas réussi à trouver la réelle source du problème.

Toutefois, cela n'empêche en rien le programme principal de fonctionner.

Étude du faux positif

Pour effectuer cette étude, nous avons tout d'abord importé la structure Hashtable.

Ensuite, nous avons rajouté la possibilité de donner un m et un k en entrée via les deux options expliquées dans le paramétrage du programme. Avec cela, nous pouvons donc modifier les différentes valeurs de k et de m sans avoir besoin de recompiler le programme pour chacun des tests.

En testant plusieurs valeurs de K et plusieurs valeurs de M, on peut voir une tendance, si on augmente K sans augmenter M, alors on remarque que le nombre de faux positifs explose. Cela s'explique en affichant l'arrayBit, on peut voir que l'ensemble des bits sont à 1. Si on le descend, alors peu de bits sont remplis dans l'arrayBit mais la plupart des mots vont donner des faux positifs.

Le deuxième cas est de ne pas bouger K mais bouger M. Premièrement, si l'on diminue M, on remarque que le nombre de faux positifs augmente fortement. De même si l'on monte trop M, le nombre de faux positifs augmente aussi.

On peut donc en déduire que K et M sont liés et que l'on ne peut pas bouger l'un sans l'autre. On retrouve alors l'équation donné dans le projet, avec la relation entre K et M.

Dans notre cas, en testant avec plusieurs valeurs, on arrive à tomber sur 2 % de chance de faux positifs dans un fichier dictionnaire de 10000 mots en testant sur celui-ci un fichier contenant l'ensemble des mots de 3 lettres (de a à Z) avec un K = 10 et un M = 16000.

Comparaison en temps et espaces

Pour effectuer ces comparaisons, nous avons tout d'abord importé la structure d 'ABR (Arbre Binaire de Recherche) ainsi que réutiliser la structure Hashtable importé pour l'étude des faux positifs.

Etienne ALEXANDRE
Léo BUZELIN

Ensuite, nous avons modifié le programme principal pour qu'il vienne utiliser à tour de rôle les différentes structures.

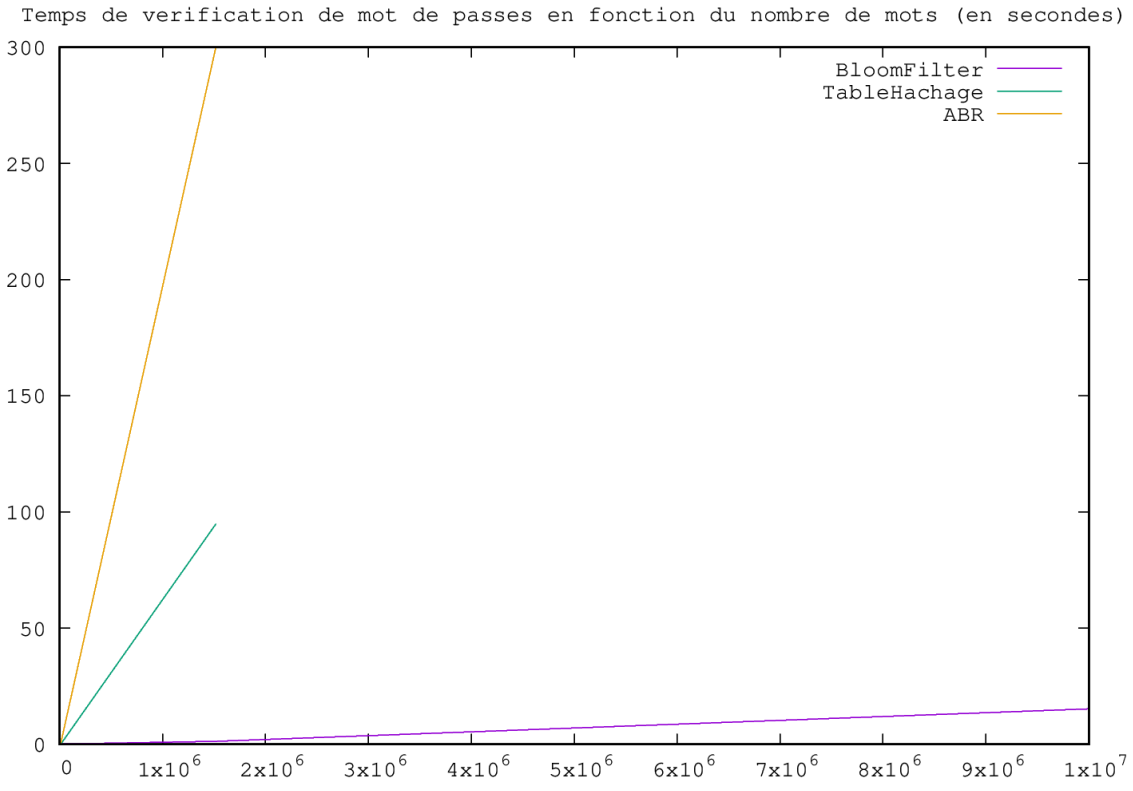
Sur chaque structure, nous avons lancé des appels sur 2 à 3 fichiers words (en fonction du temps maximal qu'elle mettait).

Et c'est avec la commande time du terminal qu'on a pu récupérer les valeurs de temps.

Nous avons fait de même pour l'utilisation de la mémoire. Toutefois, plutôt que de prendre la taille totale accumulé allouée. Nous avons pris la taille maximale allouée simultanément dans toute la durée du programme.

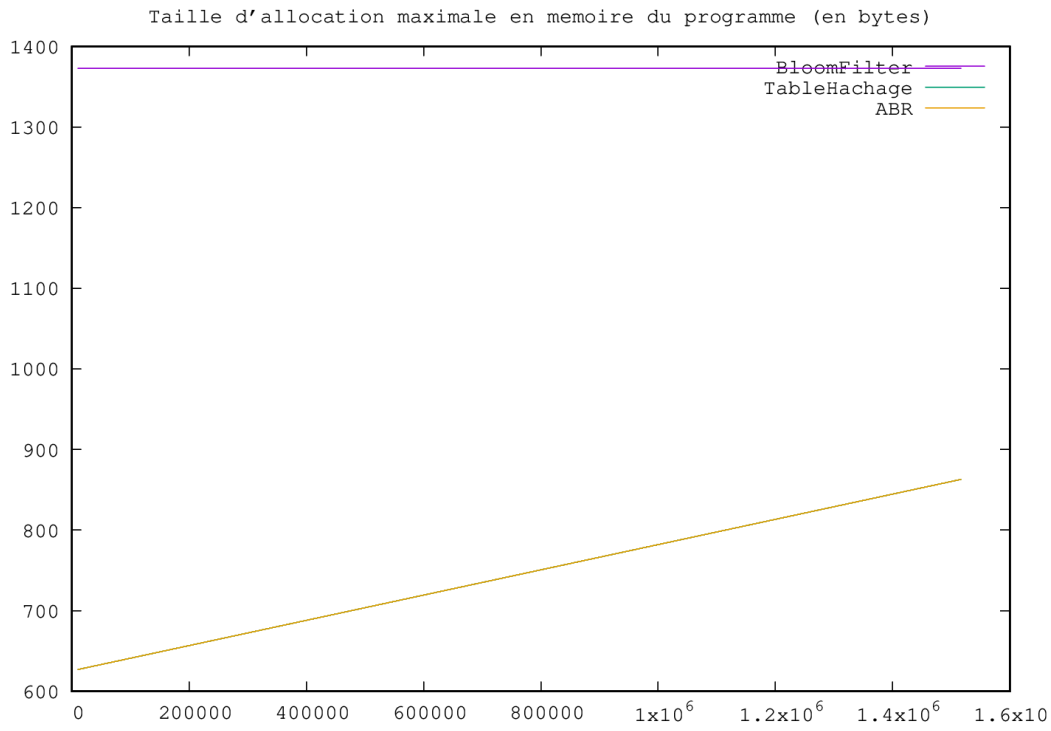
Pour cela, l'option « --xtree-memory=full » de Valgrind, nous a permis de récupérer ces valeurs.

Les différents graphiques ont ensuite pu être créés (ils peuvent être retrouvés dans le répertoire « graphics » :



Concrètement, même si le BloomFilter ne permet de savoir exactement si un mot est présent. Il reste visible que l'utilisation d'ABR est une très mauvaise idée face à notre problématique.

Etienne ALEXANDRE
Léo BUZELIN



L'utilisation de la structure Table de Hachage et ABR donnent les mêmes résultats (les 2 droites sont superposées).
Ce qu'on peut conclure de ces données est que le BloomFilter utilise un maximum en taille d'allocation qui ne bouge même pas avec le nombre de mots passés. Néanmoins, ce n'est pas le cas pour les 2 autres structures qui allouent moins lorsqu'il y a peu de mots, mais qui allouent de plus en plus avec un nombre de mots élevés.

Améliorations implantées

La seule amélioration implantée est celle de l'utilisation de bits dans la structure bitarray afin de gagner un facteur 8 en terme d'utilisation de mémoire.

Ce que nous avons appris

Tout d'abord, nous avons appris ce qu'est un filtre de Bloom. Nous avons appris à travailler en binôme sur un projet. Et même si nous n'avons pas implanté l'amélioration sur le double hachage amélioré dans notre projet, nous nous sommes renseignés dessus.