Compte-Rendu Bloom Filter

Utilisation :

Pour **utiliser le programme** : Compilez le programme avec la commande « make » puis lancez le programme en tapant « make execute » suivis de 2 arguments.
Le premier argument correspond au fichier que l'on utilisera pour remplir le filtre bloom ainsi que la table de hachage.
Le second est le fichier de vérification qu'on utilisera pour vérifier le taux de faux positif : si le mot est ans le filtre bloom, on vérifie sa présence dans la table de hachage. Si le programme est lancé avec 1 argument ou 2 manquant, une erreur s'affichera indiquant qu'il manque des arguments.

Une fois cela fait, vous devrez rentrer 3 paramètres : la taille en bit du bitarray, le nombre de fonction de hachage et la taille de la table de hachage. A noté que aucune des 3 valeurs ne peux être inférieure à 1, dans le cas où vous choisissez une valeur inférieure ou égale à 0, vous restez dans la boucle et devez entrer une nouvelle valeur.

En **sortie de programme**, 3 lignes seront affichées :
- La première contient 2 valeurs : le nombre de mots ajoutés dans le filtre bloom (nombre de mots du premier fichier) et le nombre de mots testés (nombre de mots du second fichier).
- La seconde contient 2 valeurs : le nombre de « peut-être » (présent dans le filtre bloom) et le nombre de « oui » (présent dans le filtre bloom **et** la table de hachage).
- La troisième contient 1 valeur : le taux de faux positif correspondant au nombre de « oui » divisé par le nombre de « peut-être ».

Répartition du travail :

Pour ce qui est de la **collaboration dans le binôme et de la répartition**, nous avons travaillé un maximum ensemble en vocal pour pouvoir discuter de nos choix et prendre en compte l'avis de l'autre. Nous avons répartie le travail de façon équitable avec pour objectif d'apprendre et de comprendre. Nous avons donc tous deux travaillé sur le bitarray, le filtre bloom, la table de hachage et le main, divisant seulement le nombre de fonction par 2 pour être un maximum équitable.
Par exemple, Guillaume avait des lacunes concernant l'allocation et la libération, il s'est donc chargé des fonction d'allocation et de libérations du bitarray et du filtre bloom. Cynthia, quant à elle voulais retravailler le bit à bit, c'est donc elle qui s'est chargés des fonctions bit à bit du bitarray.

Lors de ce projet, nous avons fait toutes sortes de **tests**. Pour commencer nous avons tester plusieurs façon de représenter le bitarray en bit à bit pour en venir à la conclusion que faire un tableau de int était la meilleure solution : pour chaque int on concidère 32 bits donc pour un bitarray de 16 bits, un tableau de 1 élément suffit, nous n'agissons ensuite que sur les 16 bits nécessaires.
Par conséquent, notre partie filtre bloom fonctionne, nous pouvons créer le filtre avec les paramètres nécessaire (taille du bitarray et nombre de fonctions de hachage), libérer la mémoire allouée au filtre, ajouter un mot au filtre et vérifier la présence d'un mot (avec comme réponse « non » ou « peut-être »).
Nous avons également eu un bug que nous pensons avoir résolut. Dans le main, nous avions une variable word de type char[50] mais lors de l'exécution du programme, l'erreur « smashin stack detected » s'affichait constamment. Nous avons simplement augmenté la taille de word de 50 à 1000 et le problème a disparu, malgré tout nous ne sommes pas sûr que c'est la meilleure solution.

Taux de faux positifs :

Nous avons effectué l'**étude du taux de faux positif** en utilisant des compteur tout au long du programme afin d'avoir en plus quelques statistiques.
Nous avons donc utilisés 4 compteur : un pour le nombre de mot ajouté, un pour le nombre de mot testés, un pour le nombre de « peut-être » et un pour le nombre de « oui ».
Pour obtenir le taux de faux positif, il nous a suffit de diviser le nombre de « peut-être » moins le nombre de « oui » fois 100 par le nombre de « peut-être » : ((maybe – find) * 100) / maybe.
A noter, avec les fichiers textes par défaut, 1000 mots sont ajoutés et plus de 1,5 million sont vérifiés. Pour obtenir des résultats représentatifs, il est nécessaire d'utiliser un bitarray avec un nombre bit élevé (nous avons utilisés entre 10000 et 100000 dans nos tests).
Le résultat qui est affiché correspond au pourcentage de faux-positif autrement dit, sur 100 « peut-être » si 5 sont des « oui », alors le taux de faux-positif et 95 car 95 % des « peut-être » sont au final des « non ».

Comparaison temps / espace :

Nous allons maintenant vous détailler notre **étude comparative** pour les différentes structures vues en cours. Nous avons testé, sur les fichiers 1000word et morewords, notre filtre de bloom, ainsi que la table de hachage et l'arbre binaire de recherche équilibré travaillé en TD.
Pour le filtre de bloom, nous avons donner un bitarray de 1000 bits, 10 hash différents et la table de hachage qui sert de structure secondaire a 1000 entrées.
La table de hachage possède elle aussi 100 entrées.
Voici nos résultats :

Fichier de mots de passe	Fichiers de mots tests	Filtre de bloom (1000, 10, 100)		Table de hachage (1000)		Arbre binaire de recherche équilibré	
1000words.txt	morewords.txt	0m8,988s	63,264 bits	0m0,237s	117,272 bits	0m0,284s	68,064 bits

Comme vous pouvez le constater et comme le sujet le suggérais, notre filtre de bloom est la structure la moins rapide, mais la et la plus légère. Elle a une complexité au pire des cas de O(n) en recherche et, de par sa nature probabiliste, elle nous oblige à avoir uns structure secondaire pour confirmer les résultats.
En cas de besoin d'une application plus rapide, la table de hachage est une bonne alternative, avec une complexité O(1). Cependant, elle prend beaucoup plus d'espace est n'est peut-être pas la meilleur solution pour des projets avec beaucoup de mots de passe.
Enfin, l'arbre binaire de recherche est une bonne alternative au problème. Il est plus rapide qu'un filtre de bloom et moins lourd qu'une table de hachage, mais ses rééquilibrages constants ajoutent beaucoup d'opérations pour l'ordinateur.

Améliorations :

Pour ce qui est des **améliorations**, nous avons décidé d'utiliser la manipulation de bits. Cela nous paraissait plus logique et plus simple d'utiliser le bit à bit pour ce genre de projet par soucis de mémoire, efficacité et surtout cela nous à permis de retravailler cette notion.

Nous soussignés DOMART Guillaume et DULYMBOIS-LOUISON Cynthia 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.