Projet d'Algorithmique 2
Vérification de mots de passe avec un filtre de Bloom

1 Introduction

A la sélection d'un mot de passe, c'est important de vérifier s'il figure dans une liste de mots de passe non autorisés (par exemple, un dictionnaire de mots français ou anglais ou bien une liste de mots de passe "leakés"). Il devient rapidement très lourd de garder tous ces mots en mémoire.

L'objectif du projet est d'implanter une solution à ce problème basée sur le **filtre de Bloom**.

https://en.wikipedia.org/wiki/Bloom_filter.

2 Le filtre de Bloom

Un **filtre de Bloom** est une structure de données pour représenter un **ensemble** d'objets (clés). Cette structure est très efficace en termes de mémoire. En particulier, sa taille en mémoire ne dépend pas de la taille des clés !

L'inconvénient est qu'on ne peut pas être complètement sûr qu'une clé est dans l'ensemble. Plus précisément, une recherche peut dire **non** ou **peut-être** :

- si une recherche d'une clé dit **non**, alors la clé n'est sûrement pas dans l'ensemble ;
- si une recherche d'une clé dit **peut-être**, alors la clé peut être dans l'ensemble ou pas.

Une réponse **peut-être** pour une clé qui n'est pas dans l'ensemble s'appelle un **faux positif**.

2.1 Principe de base

Un filtre de Bloom est basé sur un **Bit Array**, un tableau de 0 et de 1. Pour ajouter une clé dans l'ensemble, on y applique plusieurs fonctions de hachage. Chaque fonction de hachage donne une position dans le tableau qui est mise à 1. Pour ensuite faire une recherche, on répond **peut-être** si toutes les positions correspondant aux valeurs des fonctions de hachage sont à 1 et **non** sinon.

2.2 Exemple

Exemple d'un Bit Array de longueur 16 initialisé à 0 (l'ensemble est vide). L'indice 0 se trouve le premier à gauche.

```
0000 0000 0000 0000
```

Dans cet exemple, on utilisera des caractères comme clés avec les 3 fonctions de hachage suivantes :

- $h_1(ch) = 3 \times \text{ASCII}(ch)$
- $h_2(ch) = 11 \times \text{ASCII}(ch)$
- $h_3(ch) = 17 \times \text{ASCII}(ch)$

Pour ajouter le caractère `'A'` (code ASCII 65) à l'ensemble, on applique les 3 fonctions de hachage à `'A'` et met les bits correspondants à 1. On calcule : $h_1(\texttt{A}) = 3 \times 65 = 195$, $h_2(\texttt{A}) = 11 \times 65 = 715$, $h_3(\texttt{A}) = 17 \times 65 = 1105$. Ensuite, on met les bits correspondants (les valeurs des fonctions de hachage modulo 16, la longueur du tableau) à 1. Ici, $195\%16 = 3$, $715\%16 = 11$, $1105\%16 = 1$, donc après l'ajout, on obtient le tableau suivant :

```
0101 0000 0001 0000
```

Ensuite, on ajoute le caractère `'B'` : $h_1(\texttt{B})\%16 = 6$, $h_2(\texttt{B})\%16 = 6$, $h_3(\texttt{B})\%16 = 2$.

```
0111 0010 0001 0000
```

Finalement, on ajoute le caractère `'C'` : $h_1(\texttt{C})\%16 = 9$, $h_2(\texttt{C})\%16 = 1$, $h_3(\texttt{C})\%16 = 3$.

```
0111 0010 0101 0000
```

Pour vérifier si le caractère `'A'` est dans l'ensemble, on applique le même principe : on calcule les trois fonctions de hachage et on vérifie si tous les 3 bits correspondants (3, 11 et 1) sont mis à 1, ce qui est bien le cas ici. Donc le caractère `'A'` est **peut-être** dans l'ensemble (nous, on sait qu'il y est mais il pourrait être un faux positif).

Pour vérifier si le caractère `'D'` est dans l'ensemble, on calcule : $h_1(\texttt{D})\%16 = 12$, $h_2(\texttt{D})\%16 = 12$, $h_3(\texttt{D})\%16 = 4$. Les bits 12 et 4 sont tous les deux 0 dans le tableau, donc `'D'` n'est sûrement pas dans l'ensemble.

Pour vérifier si le caractère `'I'` est dans l'ensemble, on calcule : $h_1(\texttt{I})\%16 = 11$, $h_2(\texttt{I})\%16 = 3$, $h_3(\texttt{I})\%16 = 9$. Les bits 11, 3 et 9 sont tous les trois 1 dans le tableau, donc `'I'` est **peut-être** dans l'ensemble, c'est un **faux positif**.

Parmi les caractères `A`, ..., `Z`, il y a 8 faux positifs. Le taux de faux positifs est une fonction de la longueur du tableau, le nombre de clés stockées dans l'ensemble et le nombre de fonctions de hachage. La longueur du tableau et le nombre de fonctions de hachage sont deux paramètres fixés à la création de la structure. On paye pour un taux de faux positifs réduit en augmentant la taille de la structure et/ou le nombre de fonctions de hachage.

2.3 Remarques sur le filtre de Bloom

Comment peut-on savoir si une clé est un faux positif ou pas ? Sans autres informations, on ne peut pas car les clés ne sont pas explicitement stockées dans la structure. Dans l'exemple

ci-dessus, si on ajoute également le caractère `I`, alors le tableau ne change pas (car les positions 11, 3 et 9 sont déjà à 1) mais la réponse **peut-être** d'une recherche de `I` est désormais correcte. Donc, le filtre ne contient pas suffisamment d'informations pour dire si `I` est un faux positif ou pas.

Un filtre de Bloom est parfois utilisé pour éviter de faire une recherche plus coûteuse dans une structure secondaire telle qu'une table de hachage ou un arbre binaire de recherche qui stocke toutes les clés explicitement. Dans ce cas, si le filtre répond **non**, alors, on n'a pas besoin de consulter la structure secondaire. Par contre, si la réponse est **peut-être** et on veut être sûr, alors il faut à ce moment faire une deuxième recherche dans la structure secondaire.

Une autre utilisation d'un filtre de Bloom est dans des applications où ce n'est pas strictement nécessaire de vérifier les faux positifs. C'est le cas dans l'application de ce projet : si, dans 1% ou 0,1% des cas, on tombe sur un faux positif et que l'application n'autorise pas un mot de passe même s'il ne figure pas réellement dans l'ensemble, alors ce n'est pas la fin du monde.

Ce n'est pas possible de retirer des clés de l'ensemble, seulement de les ajouter (mais il y a des variations de la structure qui permettent également des suppressions). Alors, le nombre de clés dans l'ensemble ne peut qu'augmenter.

3 Implantation d'un filtre de Bloom

Pour l'implantation d'un filtre de Bloom, vous devez fournir les structures et les fonctions suivantes dans des fichiers `.h` et `.c` séparé.

3.1 Bit Array

Dans les fichiers `bitarray.h` et `bitarray.c` vous devez fournir au moins la structure et les fonctions suivantes :

```
typedef struct _bitarray {
    /* TODO */
} bitarray;

/* Return a pointer to an empty bitarray that can store m bits */
bitarray *create_bitarray(int m);

/* Free the memory associated with the bitarray */
void free_bitarray(bitarray *a);

/* Set position pos in the bitarray to 1 */
void set_bitarray(bitarray *a, int pos);
```

```
/* Set position pos in the bitarray to 0 */
void reset_bitarray(bitarray *a, int pos);

/* Get the value at position pos in the bitarray */
int get_bitarray(bitarray* a, int pos);

/* Set all positions in the bitarray to 0 */
void clear_bitarray(bitarray *a);
```

On vous conseille dans un premier temps de faire une implantation basée sur un tableau d'entiers (un unsigned char par exemple) qui ne prennent que des valeurs 0 et 1. Une telle implantation est rapide et facile à faire mais utilise plus de mémoire que nécessaire (dans le cas d'un tableau de unsigned char, il y a 7 bits de chaque octet ne sont pas utilisés).

Dans un deuxième temps, après avoir terminé les autres parties du projet, vous pouvez revenir sur cette structure et la réimplanter avec des manipulations de bits pour stocker 8 positions du Bit Array dans chaque octet. Cela permet de gagner un facteur 8 en terme d'utilisation de mémoire.

3.2 Filtre de Bloom

Dans les fichiers `filter.h` et `filter.c` vous devez fournir au moins la structure et les fonctions suivantes (voir section 3.3 pour la fonction `hash`) :

```
typedef struct _filter {
    /* TODO */
} filter;

/* Return a pointer to an empty filter with parameters m and k */
filter *create_filter(int m, int k);

/* Free the memory associated with the filter */
void free_filter(filter *f);

/* Compute k hash values for the string str and place them in the
array hashes. */
void hash(filter *f, char *str, unsigned hashes[]);

/* Add the key str to the filter */
void add_filter(filter *f, char *str);

/* Check if the key str is in the filter, 0 means no, 1 means maybe */
int is_member_filter(filter *f, char *str);
```

3.3 Fonctions de hachage

La fonction `hash` de `filter.h` doit appliquer k fonctions de hachage différentes (et idéalement indépendantes). Il faut également que les fonctions de hachage soient rapides à appliquer ce qui exclue certaines fonctions de hachage cryptographiques.

On vous propose ici dans un premier temps d'utiliser une solution simple basée sur la fonction de hachage utilisée dans le tp 6. A la création du filtre, vous pouvez tirer au hasard k poids **différents** entre 2 et 255 et placer ces poids dans un tableau. La fonction de hachage numéro i peut alors être calculée comme

$$str[0] * (poids[i])^t + str[1] * (poids[i])^{t-1} + ... + str[t-2] * (poids[i])^1 + str[t-1]$$

où t est la longueur de la chaîne.

4 Vérification de mots de passe

Vous devez écrire une application qui permet de charger une liste de mots dans un filtre de Bloom et d'effectuer des recherches pour voir si un mot est dans l'ensemble ou pas. Il faut que l'application soit utilisable sur le terminal avec des options : le nom du fichier de mots à charger, des options pour les valeurs de k (le nombre de fonctions de hachage) et m (le nombre de bits dans le bit array) (ou une sélection automatique de ces paramètres), etc. Pour commencer, essayez de fixer m à une petite constante (5-10) fois le nombre de clés n. Augmenter m permet de réduire le taux de faux positifs.

Pour faciliter de faire plusieurs recherches consécutives, vous pouvez prendre un deuxième fichier en entrée et rechercher tous les mots dans celui-ci.

Des exemples de listes de mots de passe ainsi que des tests pour vos structures bitarray et filter seront mis à disposition sur le site `elearning` du cours.

Vous devez également effectuer une étude sur la performance de votre solution, décrite dans les deux sections suivantes.

4.1 Sélection des paramètres

Effectuez une étude des faux positifs comme une fonction des paramètres m et k pour une liste de mots fixée. Pour estimer le taux de faux positifs, vous devez :

- stocker les mots dans votre filtre de Bloom ainsi que dans une structure secondaire (par exemple, une table de hachage) ;
- pour tout mot possible de longueur 3 (ou 4) : rechercher le mot dans le filtre, si la réponse est **peut-être**, alors vérifier dans la structure secondaire si c'est un faux positif ou pas ;

— diviser le nombre de faux positifs trouvés par le nombre de mots recherchés au total ;
— finalement, comparer le résultat avec la valeur théorique (voir la section 5) et vérifier si le meilleure choix du paramètre k coïncide avec la valeur optimale théorique.

4.2 Comparaison de temps et d'espace

Pour le problème de vérification de mot de passe, effectuez une comparaison de temps et d'espace utilisé par des 3 solutions suivantes :

1. un filtre de Bloom (votre application principale)
2. une table de hachage (cf. tp 6 sans stocker les occurrences)
3. un arbre binaire de recherche (cf. tp 8-9)

5 Un peu de théorie

On note la longueur du Bit Array par m et le nombre de fonctions de hachage par k. Dans l'exemple de l'introduction, on a utilisé $m = 16$ et $k = 3$. Ces deux paramètres sont fixés à la création de la structure et ne peuvent pas changer. En général, on réduit le nombre de faux positifs en augmentant m. Par contre, si on augmente k de trop, alors le tableau va rapidement se remplir par des 1 ce qui augmente le taux de faux positifs. (et le temps d'exécution pour appliquer les k fonctions augmente aussi).

Le taux de faux positifs ϵ est approché par la formule

$$\epsilon \approx (1 - e^{-kn/m})^k$$

où n est le nombre de clés stocké dans le filtre.

On peut choisir ses paramètres d'une manière optimale : pour m et n fixé, le choix optimal pour k est donné par la formule

$$k = \frac{m}{n} \ln(2)$$

Voir aussi

https://en.wikipedia.org/wiki/Bloom_filter#Optimal_number_of_hash_functions

6 Améliorations / pour aller plus loin

Les suivantes sont des suggestions pour explorer davantage le filtre de Bloom. Certaines sont plus laborieuses que d'autres. Vous pouvez les faire ou pas dans l'ordre que vous voulez.

6.1 Manipulation de bits

Réimplanter la structure bitarray avec des manipulations de bits pour pouvoir stocker 8 positions dans chaque octet. Cela permet de gagner un facteur 8 en terme d'utilisation de mémoire.

6.2 Fonctions de hachage

Pour accélérer le temps de calcul des k fonctions de hachage, vous pouvez utiliser une technique qui s'appelle le **double hachage amélioré** :

https://en.wikipedia.org/wiki/Double_hashing#Enhanced_double_hashing.

6.3 L'union de deux filtres

Écrire une fonction qui permet de faire l'union de deux filtres passés en argument : si une clé appartient à au moins un des deux filtres, alors elle devrait aussi appartenir à l'union. Comment effectuer cette opération sans avoir accès aux clés ?

6.4 Nettoyage de mots de passe

Afin de détecter davantage de mots de passe faibles, vous pouvez nettoyer les mots de la liste de mots non autorisés avant de les ajouter au filtre. Vous devez de la même manière nettoyer un mot de passe avant de faire une recherche. Le nettoyage pourrait par exemple supprimer les accents et mettre tous les caractères alphabétiques en minuscule.

6.5 Suppression

Une façon de permettre l'opération de suppression d'une clé du filtre est de remplacer chaque bit par un compteur qui enregistre le nombre de fois que ce bit à été mis à 1. La suppression consiste alors en décrémenter les compteurs correspondant aux fonctions de hachage. Voir le lien :

https://en.wikipedia.org/wiki/Counting_Bloom_filter

6.6 Have I been pwned ?

Les codes de hachage des mots de passe dans la base du site `haveibeenpwned.com` sont disponibles pour téléchargement.

https://haveibeenpwned.com/Passwords

Écrivez une application qui les stocke dans un filtre de Bloom et permet de faire des recherches. La fonction de hachage utilisée pour les mots de passe est SHA-1 (https://en.wikipedia.org/wiki/SHA-1), donc pour faire une recherche, il faut d'abord appliquer cette fonction au mot de passe donné par l'utilisateur avant de vérifier s'il figure dans le filtre. La liste complète est de 12,5 Go, quelle est la taille de votre filtre pour un taux de faux positifs de 1% ?