L3-MI / PROG
TP#8 : Les filtres de Bloom : une structure de données probabiliste

2022-W47

Instructions de rendu

Ce TP fait l'objet d'un rendu *en binôme*. Vous devez envoyer votre travail sous la forme d'une archive à philip.scales@univ-grenoble-alpes.fr, au plus tard le 2022-12-02 à 18 :00. Celle-ci doit contenir :

— Votre programme répondant aux questions ci-dessous. Il doit être possible de facilement exécuter toutes les expériences permettant de répondre aux questions.
— Un Makefile et les instructions permettant de l'utiliser.
— Un compte-rendu détaillé en format PDF qui décrit vos réponses aux questions.

Le but de cet exercice est d'implémenter une version simple d'un *filtre de Bloom* (ci-après «F.B.»), qui est une structure de données probabiliste offrant les opérations suivantes :

1. Soit $\mathcal{S}$ un F.B. et x une chaîne binaire, il est possible d'*ajouter* x à $\mathcal{S}$.
2. Soit $\mathcal{S}$ un F.B. et x une chaîne binaire, il est possible de *tester l'appartenance* de x dans $\mathcal{S}$. Si x a préalablement été ajouté à $\mathcal{S}$, ce test renverra toujours « vrai ». Si x n'a *pas* été ajouté, ce test pourra renvoyer « vrai » ou « faux ».

Autrement dit, un F.B. est une structure de donnée d'ensemble où seuls des ajouts sont possibles (les suppressions sont impossibles) et où les tests d'appartenance peuvent donner des *faux positifs* (mais pas de *faux négatifs*).

Le fonctionnement d'un F.B. est le suivant : l'ensemble $\mathcal{S}$ est représenté par une chaîne s de m bits (où m est un paramètre) qui est initialisée à la chaîne de m bits nuls « 0^m », qui représente l'ensemble vide $\varnothing$. On se dote également de $\mathcal{H} := \{H_i : i \in [\![1, k]\!]\}$ un ensemble de k fonctions de hachage indépendantes, de signature $\{0,1\}^* \to [\![0, m-1]\!]$, où k est un autre paramètre. Pour ajouter un élément x dans $\mathcal{S}$, on calcule les k empreintes $h_{x,i} := H_i(x)$ à valeur dans $[\![0, m-1]\!]$ (qui ne sont pas nécessairement distinctes), et pour chacune d'entre elle on affecte 1 au bit de s d'indice correspondant. Pour tester la présence d'un élément x' dans $\mathcal{S}$, il suffit alors de calculer les k empreintes $h_{x',i}$ et de vérifier si tous les bits de s correspondant valent 1.

Exemple : Soit $m = 8$, s vaut initialement `00000000`. On ajoute un élément x d'empreintes 2, 6, 6 (pour $k = 3$) en affectant 1 aux bits correspondants ; cela donne $s =$ `01000100` (en numérotant les bits depuis zéro, et de droite à gauche). On ajoute ensuite un élément x' d'empreintes 2, 5, 7 ; la nouvelle valeur de s est alors `11100100`. Les deux éléments x et x' seront tous deux détectés à raison comme appartenant au F.B., de même (à tord) qu'un élément d'empreintes par exemple égales à 5, 6, 7.

Consignes

1. Pour répondre aux questions suivantes, vous devez utiliser la famille de fonction de hachage du TP#4. Dans le cas où vous ne l'auriez pas (correctement) implémentée, vous pouvez par défaut utiliser la famille de fonction donnée par le code ci-dessous :

```
uint64_t hash_tp8l3mi(uint32_t k, size_t buflen, uint8_t buf[buflen])
{
    uint64_t acc = 0;
    for (size_t i = 0; i < buflen; i++)
    {
        acc ^= buf[i];
        acc += (acc << 28) ^ (acc << 34);
        acc *= (k | 1);
    }
    return acc;
}
```

2. La structure de données utilisée pour représenter un F.B. est imposée, et consiste en :

```
struct bf
{
   uint32_t key; // main hf key
   size_t   m;   // bitsize
   uint32_t k;   // #hf
   uint64_t bits[]; // "s"
};
```

3. La structure de données utilisée pour représenter les éléments à insérer dans le F.B. est imposée, et consiste en :

```
struct bd
{
    size_t sz;
    void *dat;
};
```

4. Valgrind ne doit reporter aucune fuite mémoire pour votre programme.

Q.1 : Écrivez une fonction de signature :

```
struct bf *create_bf(uint32_t key, size_t m, uint64_t n);
```

qui fait les choses suivantes :
— Allouer sur le tas l'espace mémoire nécessaire pour représenter un F.B. avec m bits dans une structure `struct bf`.
— Tirer uniformément une clef de 32 bits pour une fonction de hachage.
— Calculer le nombre de fonctions de hachage à utiliser, via la formule $k := \lceil \ln(2) \times m/n \rceil$ [1]
— Initialiser la structure représentant le F.B. dans l'espace précédemment alloué, et la retourner.

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

Q.2 : Écrivez une fonction `delete_bf` qui libère l'espace mémoire alloué pour la représentation d'un F.B., et affecte zéro au pointeur qui le référençait.

Q.3 : Écrivez une fonction de signature :

```
void insert(struct bf *s, struct bd elem);
```

qui insère l'élément représenté par `elem` dans le F.B. représenté par `s`. Les k fonctions de hachage à utiliser sont celles de clef $(\texttt{key} + i)$ `%` 2^{32} pour $i \in [\![0, k-1]\!]$ [2].

Q.4 : Écrivez une fonction de signature :

```
bool prbly_in_set(struct bf *s, struct bd elem);
```

qui indique si l'élément représenté par `elem` est (probablement) présent dans le F.B. représenté par `s`.

Q.5 : Écrivez une fonction de signature :

```
bool test_insert(size_t m, uint64_t n);
```

qui génère aléatoirement (uniformément et indépendamment) `n` éléments d'au moins 64 bits, les insère dans un F.B. de `m` bits, et vérifie ensuite que tous les éléments sont bien détectés comme présents dans le F.B.. Utilisez cette fonction pour tester votre implémentation de F.B..

Q.6 : Écrivez une fonction de signature :

```
uint64_t test_fp(size_t m, uint64_t n, uint64_t nfpt);
```

qui génère aléatoirement (uniformément et indépendamment) `n` éléments d'au moins 64 bits et les insère dans un F.B. de `m` bits, et ensuite génère de la même façon `nfpt` nouveaux éléments distincts des `n` premiers [3] et compte combien parmi ces éléments sont détectés comme appartenant au F.B. (autrement dit, combien sont des faux positifs).

Q.7 : Lancez votre fonction `test_fp` avec les paramètres suivants, et commentez brièvement les résultats (aucune analyse statistique n'est requise, mais vous pouvez néanmoins en proposer une si vous le souhaitez) :

1. 64, 64, 100 000
2. 1280, 1280, 100 000
3. 64, 64×6, 100 000
4. $64 \times i$, 64, 64 000 pour $i \in [\![1, 40]\!]$
5. $100 \times i$, 100, 100 000 pour $i \in [\![1, 40]\!]$
6. $256 \times i$, 256, 256 000 pour $i \in [\![1, 40]\!]$
7. $1024 \times i$, 1024, 1 024 000 pour $i \in [\![1, 40]\!]$

2. On peut remarquer que ces fonctions ne sont pas indépendantes, et donc ne respectent pas strictement la spécification d'un F.B.

3. Par le paradoxe des anniversaires, tant que `n` $\times$ `nfpt` $\ll 2^{64}$, il est très peu probable qu'un de ces nouveaux éléments soit égal à l'un des `n` premiers, et vous pouvez donc vous dispenser de tester explicitement cette propriété.

Q.8 : Même question que ci-dessus pour une valeur k du nombre de fonctions fixée à 1, puis fixée à 10.

Q.9 : On suppose un contexte où les éléments à insérer dans le F.B. sont « grands » (par exemple de taille supérieure à 1 MB) et où $k > 1$. Proposez une façon de calculer les k indices requis par un F.B. qui soit plus efficace que l'appel à k fonctions de hachage indépendantes sur les éléments. (Aucune implémentation n'est demandée pour cette question.)