UNIVERSITÉ DE MONTRÉAL
Département IRO

Été 2018
Neil Stewart

IFT 2015 E18

Devoir 3.

10/10, soit 10% de la note finale.

1 Partie Pratique (6 points)

Tel que vu en démonstration, un filtre de Bloom est une structure de données probabiliste qui permet de savoir avec certitude qu'un élément est absent d'un ensemble, et avec une certaine probabilité qu'il est présent.

En d'autres termes, il est possible d'avoir de faux positifs, mais pas de faux négatifs. Sans oublier qu'il est possible de controller la probabilité de faux positifs, la structure a l'avantage d'être particulièrement compacte !

Tout d'abord, vous devez implanter l'ensemble de bits (`BitSet`) utilisé par votre filtre. La façon naïve (i.e. tableau de booléens) ne vaudra que des points partiels. Vous êtes encouragés à ajouter d'autres fonctions, si le besoin se présente.

Par la suite, vous devez implanter votre fonction de hachage. Elle peut résider dans `BloomFilter` ou séparément, dans une autre classe ou même dans un autre fichier. La signature n'a pas été définie. Vous pouvez donc spécifier la vôtre.

Le choix de la fonction de hachage est important. Il faut qu'elle génère peu de collisions et qu'elle soit suffisamment rapide. Vous n'êtes pas obligés d'en inventer une, mais l'implantation doit être effectuée par vous. *Attention au plagiat !*

Finalement, vous devez implanter le filtre de Bloom, en utilisant ce que vous avez développé précédemment. Vous pouvez ajouter un `public static void main` à des fins de tests, mais veuillez le commenter avant la remise. La méthode `getBytes()` de `String` peut s'avérer utile lors de vos tests.

Comme d'habitude, des fichiers squelettes vous sont fournis sur StudiUM, avec plus de détails sur chacune des méthodes à implanter. Il ne faut pas modifier la signature de ces méthodes.

À noter que vous ne pouvez pas importer de classes de la librairie standard de Java pour ce devoir.

Veuillez remettre vos versions complétées, avec vos noms et matricules en début de fichier, sans déclaration de `package`. Assurez-vous que le fichier remis compile avec `javac` et est encodé en *UTF-8*.

2 Partie Théorique (4 points)

1. (1 point) *Hachage pour chaînes*

   Il a été dit dans le cours qu'il est possible de faire le modulo au fur et à mesure dans le calcul de la fonction de hashing (règle de Horner).

   Si nous effectuons l'opération $a \leftarrow b \bmod M$ nous obtenons a tel que $a \equiv b \pmod{M}$ et $a \in [0, M-1]$. Nous nous intéressons au cas où a, b, x et M sont des entiers non-négatifs. Montrez que

   $$(((ax) \bmod M) + b) \bmod M = (ax + b) \bmod M.$$

2. (1/2 point) *Arbres cousus*

   La motivation pour les arbres cousus est de ne pas gaspiller les champs qui contiennent des références nulles dans un arbre binaire (qui est peut-être par exemple un arbre binaire de recherche): utilisons ces champs pour d'autres types d'opération.

   S'il y a N clefs dans l'arbre, il y a combien de références nulles? Expliquez (c'est trivial pour vous maintenant).

3. (1 point) *Prim/Dijkstra*

   Dans l'algorithme de Prim il y a un champs de la table T, soit $T[v].p$, qui contient la valeur de p, et les paires (v, p) donnent les arêtes dans l'arbre sous-tendant minimal.

   Nous avons vu la petite modification de l'algorithme de Prim qui donne l'algorithme de Dijkstra. Donnez en pseudo-code une procédure récursive qui donne, pour l'algorithme de Dijkstra, le chemin du point de départ (v_4 dans l'exemple présenté dans le cours) au noeud spécifié. Par exemple, pour l'exemple présenté dans le cours, un appel avec v_6 donnerait

   $$v_4 \text{ à } v_5 \text{ à } v_7 \text{ à } v_6$$

4. (1/2 point) *Unicité Prim*

   Weiss, page 419, Exercice 9.15, (a) et (b), avec l'algorithme de Prim seulement pour la partie (a).

5. (1 point) *Splay Tree*

   Weiss, page 163, Exercices 4.28 et 4.29.

À réaliser en équipes de 1 ou 2. À remettre le 25 juillet, 2018, avant `9:00`. Les devoirs en retard ne seront pas acceptés.