Devoir 2 : les filtres de Bloom.

La Police Nationale possède une liste de personnes suspectes de corruption. Nous souhaitons installer un contrôle à la douane pour repérer parmi les individus sortants du territoire lesquels font parties des personnes suspectées. Nous voulons donc construire un dictionnaire de toutes les personnes suspectes, par exemple une table de dispersion ou un arbre de recherche équilibré. Chaque personne souhaitant alors quitter le territoire est testé avec ce dictionnaire.

Dans ce genre d'application, on peut vouloir effectuer un premier test pour détecter rapidement et avec peu d'informations si une personne n'est pas suspectée ou bien si elle risque d'être sur la liste. Typiquement, on peut vouloir utiliser juste son prénom et son nom dans un premier temps. Si son prénom et son nom ne sont pas dans la liste des suspects, elle est innocentée. Sinon, il peut s'agir d'un homonyme, on voudra alors tester de façon plus approfondie en comparant d'autres informations d'état civil, biométriques, ou autres, pouvant par exemple être dans un fichier centralisé au ministère de l'intérieur.

On distingue donc plusieurs scénarios possibles :

— La personne est immédiatement innocentée par le premier test (*négatif*).
— La personne n'est pas innocenté par le premier test, mais par le second. On parle alors de *faux positif* : le système a cru trouver un suspect, mais s'est trompé.
— La personne est bien detectée par le premier test et confirmée par le second (*positif*).

Notons que nous ne voulons pas de *faux négatif* qui correspondrait à des suspects qui resteraient indétectés.

Nous voulons donc un dictionnaire qui peut se tromper en affirmant qu'un élément serait contenu alors qu'il ne l'est pas, mais avec un faible taux d'erreurs. En introduisant un peu d'erreurs, on peut espérer avoir un algorithme plus efficace en temps et en espace. L'algorithme que nous allons coder s'appelle filtre de Bloom.

1 Rendu

Ce devoir est relativement court, nous y consacrerons donc deux séances de TP seulement. Il est à réaliser en monôme ou en binôme. À l'issue des séances de TP qui lui sont dédiées, et à la date prescrite (selon votre campus), vous devrez rendre sur Ametice, sous forme d'un unique fichier archive d'extension `gz` ou `tar.gz` **exclusivement**, contenant l'ensemble de vos sources et un bref rapport (dans un simple fichier texte) décrivant comment utiliser votre projet, vos résultats, et le cas échéant une description des fonctionnalités non-implémentées.

Si votre projet contient du code source que vous n'avez pas écrit vous-même, **vous indiquerez précisément sa nature et sa provenance** (internet, les amis, ...). Tout projet contenant du code non-correctement attribué à ses véritables auteurs s'expose à recevoir une note nulle. Vous êtes priés de ne pas fournir votre code à d'autres étudiants, sous peine de vous exposer aussi à recevoir une note nulle.

La date de rendu dépend de votre campus, demandez-la à votre chargé de TP.

2 Principe d'un filtre de Bloom

Notons n le nombre de clés devant être stockées dans le filtre de Bloom. Remarquons d'abord que dans un filtre de Bloom, on ne peut pas associer une valeur à chaque clé, comme on le ferait dans un dictionnaire normal : la seule requête possible sur une clé est de demander si elle est présente ou pas. Une autre différence est qu'on ne peut pas supprimer une clé insérée précédemment.

Le principe est d'utiliser un *champ de bit* : un tableau de bit codant chacun un booléen, et ceux en utilisant un minimum d'espace. Notons m la longueur du champ de bit, initialement ne contenant que des 0. Pour chaque clé contenue par le filtre, on marque k (avec k un entier bien choisi) de ces bits à 1, choisis en fonction de la clé. Ainsi, si une clé est stockée dans le filtre,

les k bits correspondant doivent être positionnés sur 1. Par contraposition, si l'un des bits est positionné à 0, c'est que la clé est absente du filtre.

Comment choisir les k bits associés à une clé ? Idéalement on voudrait les prendre de façon pûrement aléatoire. Mais on veut aussi que leurs positions puissent être trouvées à partir de la clé, ce qui exclut un choix aléatoire. Nous nous rappelons alors que les fonction de dispersions simulent un comportement aléatoire bien qu'elles soient déterministes. La méthode est alors de choisir k fonctions de dispersions $h_1, \ldots, h_k$, avec $h_i : \{\text{clés}\} \to [\![0, m-1]\!]$. Chaque fonction donne la position d'un des bits.

C'est le moment de faire un peu de mathématiques. En supposant que les fonctions de dispersions distribuent les clés uniformément et indépendamment, construire le filtre équivaut à choisir aléatoirement nk fois une case et à la mettre à 1. La probabilité qu'une case ne soit jamais choisie est donc :

$$\left(1 - \frac{1}{m}\right)^{nk} \approx e^{-nk/m}$$

En prenant $k = \ln 2m/n$, on obtient une probabilité $p = 1/2$ qu'une case soit vide. Dans ce cas, la probabilité qu'une clé soit un faux positif est, en supposant encore l'uniformité des fonctions de dispersions, $(1-p)^k$ (la probabilité que toutes les cases soient des 1). Pour une probabilité de l'ordre du pourcent, il faut alors prendre $k \approx 7$, et donc $m \approx 10n$.

Lorsque vous construirez votre filtre de Bloom, pensez à utiliser ces valeurs, et à vérifier la densité du nombre de cases marquées 1 par rapport au nombre total de cases : elle devrait être proche de 0.5.

Ainsi, si une clé n'a pas été explicitement ajoutée dans le filtre, avec forte probabilité au moins une des cases associées à cette clé vaut 0. Par contre si la clé a été ajoutée lors de la création du filtre, par définition tous les bits associés sont à 1. Pour vérifier si une clé est négative ou probablement positive, il suffit donc de tester si tous ses bits associées sont des 1.

Au final, en quoi les filtres de Bloom sont-ils efficaces dans leur travail ? En terme de complexité en temps, chaque opération (insertion d'une clé, test d'une clé), prend un temps constant puisque k, le nombre de fonction de dispersion, est fixé. C'est donc asymptotiquement optimal, théoriquement mieux que les tables de dispersions (temps constant en espérance seulement). Pour l'utilisation mémoire, notre choix de m conduit à utiliser 10 bits par clés, quel que soit la taille d'une clé. Une table de dispersion doit garder une copie de chaque clé, donc utilisera en général bien plus de place, même si aussi en quantité linéaire par rapport au nombre de clés. Le filtre de Bloom est donc particulièrement recommendable lorsque l'on ne souhaite pas stocker les clés en mémoire (ou bien seulement en mémoire distante), par exemple pour des soucis de préservation d'anonymat.

3 Travail

Nous vous donnons deux fichiers de données :
- une liste des suspects, contenant 577 personnes,
- une liste de personnes à tester, contenant 100.000 personnes (et une autre contenant 10.000.000 personnes pour mettre votre programme, et Java, au supplice).

Chaque personne est codée en trois lignes, contenant respectivement les prénoms, le nom, la ville de naissance de ces personnes.

Vous devez les utiliser en construisant un filtre avec les 577 suspects, puis en filtrant les noms dans l'autre fichier. Comparer alors le nombre de faux positifs et le nombre de positifs. Vous pouvez mettre ça en parallèle au dernier débat sur la loi renseignement, proposant d'adopter des logiciels qui se basent sur des écoutes de données pour détecter des personnes susceptibles d'appartenir à des mouvances terroristes : un reproche fait par les détracteurs concerne justement le problème des faux positifs (voir par exemple `http://fr.scribd.com/doc/265206918/Note-interne-de-l-Inria`). Un autre exemple célèbre de problème des faux positifs : `news.bbc.co.uk/2/hi/americas/2995288.stm`. On peut aussi faire le parallèle avec

```java
public class Bloom<Key> {

  Random gen;
  BitSet bitmap;
  // D'autres champs...

  public void add(Key key) {
    //TODO
  }

  public boolean probablyContains(Key key) {
    //TODO
  }

  // D'autres méthodes...
}
```

FIGURE 1 – Une ébauche pour la classe `Bloom`.

certaines méthodes de détections de maladies rares, qui peuvent aussi détecter bien plus de faux positifs que de cas réels : est-ce pour autant de mauvaises méthodes ?

En utilisant les deux fichiers ainsi, le filtre de Bloom est petit, et le nombre de tests est grand. Pour analyser votre algorithme avec un gros filtre de Bloom, inversez l'utilisation des deux fichiers : créer un filtre avec les 100.000 personnes, puis tester les 577 autres.

Dans les deux cas, comparer votre filtre de Bloom avec les tables de dispersion (`HashSet` en Java) et les arbres binaires de recherche équilibrés (`TreeSet` en Java pour les arbres rouge-noirs).

4 Conseils pour l'implémentation

Créez une classe pour les personnes. Dedans, ajoutez une fonction de dispersion, un test d'égalité, et une fonction de comparaison. Vous pouvez demander à Eclipse de vous générer les deux premiers : lisez alors le résultat et n'hésitez pas à simplifier ou modifier le code ainsi généré. Enfin, vous aurez besoin d'une méthode `toString` pour l'affichage ou le débugage.

La classe pour le filtre de Bloom devrait être générique : n'importe quelle classe peut être choisie comme classe de clés, du moment qu'il y a une fonction de dispersion (la méthode `hashCode()` en Java, que tout objet doit avoir). Votre code doit ressembler à la Figure 1.

Vous pourrez ensuite créer un filtre de Bloom sur les personnes ainsi :

```java
Bloom<Person> filter = new Bloom<>(expectedNbrOfKeys);
```

Un filtre de Bloom contient k fonctions de dispersions. On utilise une famille universelle de fonctions de dispersions, en prenant un nombre premier p et en choisissant aléatoirement $a \in [\![1, p-1]\!]$ et $b \in [\![0, p-1]\!]$ et comme fonction :

$$x \to (ax + b) \bmod p \bmod m$$

avec m la longueur du champ de bit.

Pour générer le nombre premier, on utilise :

```java
prime = BigInteger.probablePrime(30,gen).intValue();
```

qui génère un nombre premier sur 30 bits. Ici, `gen` doit être un générateur aléatoire, objet de la classe `Random`. Il sert aussi à générer les valeurs de a et b, *cf.* Figure 2 pour l'utilisation. Attention, chaque instance de filtre de Bloom génère k couples a, b lors de sa construction, et utilise ensuite ces k paires pour toutes les opérations d'insertion et de tests. Pour trouver le nombre de bits et le nombre de fonctions de dispersions, vous pouvez utiliser ce calculateur en ligne : `http://hur.st/bloomfilter?n=4&p=1.0E-20`.

```java
public class Random {

  public Random(); // constructor

  public int nextInt(int maxi); // ∈ [[0, maxi]]

  public double nextDouble(); // ∈ [0.0, 1.0]

  // etc.
}
```

FIGURE 2 – Un extrait de la classe `Random`.

```java
File nameOfSuspects = new File(argv[0]);
ArrayList<Person> listOfSuspects = new ArrayList<>();
Scanner bufPositive = new Scanner(nameOfSuspects);
bufPositive.useDelimiter("\n");
while (bufPositive.hasNext()) {
  listOfSuspects.add(readPerson(bufPositive));
}
bufPositive.close();
```

FIGURE 3 – Lecture d'un fichier de personnes

Pour la lecture du fichier, comme dans le premier devoir on utilise la classe `Scanner`, Figure 3. Dans ce code `readPerson` est une méthode statique qui utilise `bufPositive.next()` trois fois pour obtenir le prénom, le nom, et la ville de naissance de la personne.

N'oubliez pas de comparer votre filtre de Bloom à l'utilisation directe d'une table de dispersion créée avec `HashSet`, ou d'un arbre binaire de recherche équilibré crée avec `TreeSet`. Les deux s'utilisent avec les méthodes `add` et `contains`. Utilisez `System.nanoTime()` pour mesurer les temps de calculs.

Enfin, pour ceux voulant traiter le gros fichier de 10.000.000 personnes, vous risquez de dépasser la capacité en mémoire de la machine virtuelle. Pour contrer cela, le fichier de noms est constitué depuis une petite liste de chaines de caractères différentes. Il faut lorsque vous créez une chaîne de caractères vérifier si elle n'existe pas déjà, afin de partager au maximum la mémoire. Pour cela, utiliser la classe `StringSet`, Figure 4 : lisez-la et essayez de comprendre comment vous pourriez en profiter.

Bon travail !

```java
import java.util.HashMap;

public class StringSet {

  static HashMap<String,String> dict = new HashMap<>();

  public static String create(String str) {
    if (dict.containsKey(str)) return dict.get(str);
    dict.put(str,str);
    return str;
  }
}
```

FIGURE 4 – Une classe pour le partage des chaînes de caractères.