SAE 3.8 Algorithmique

Projet – Filtre de Bloom –

Objectif : L'objectif de ce projet est d'implémenter un filtre de Bloom en utilisant plusieurs structures de données et de comparer ces différentes implémentations par un banc d'essai (benchmark).

Langages et technologies : Java / NetBeans

J'ai choisi de faire un projet Java Ant. D'origine j'avais faire un Maven mais je ne pouvais pas lancer le projet depuis chez moi.

Principe

Un filtre de Bloom est une structure de données permettant de façon efficace de tester la présence ou non d'un élément dans le filtre. En particulier, le filtre de Bloom permet de tester :

• avec certitude : l'absence d'un élément (si le test indique que le filtre ne contient pas une valeur, alors il ne la contient pas)

• avec une certaine probabilité : la présence d'un élément (si le test indique que le filtre contient une valeur, il peut se tromper avec une certaine probabilité)

Un filtre de Bloom est constitué d'un tableau de bits (ou de booléens) de m cases et de k fonctions de hachage. Chacune de ses k fonctions de hachage associe, à chaque élément à stocker, un indice dans le tableau (donc entre 0 et m-1).

Lorsqu'on ajoute un élément e dans le filtre, il faut calculer les k indices en utilisant les k fonctions de hachage et de passer à 1 les k cases correspondantes. Pour tester la présence d'un élément dans le filtre, il suffit de s'assurer que les k cases correspondantes contiennent bien la valeur 1. Si une ou plusieurs cases ne contiennent pas de 1, alors l'élément n'est pas présent (avec certitude). En revanche, il est possible que plusieurs valeurs déjà ajoutées au filtre utilisent les k cases correspondant à l'élément e, et donc que la recherche retourne vrai alors que e n'est pas dans le filtre.

On peut alors noter que les problèmes évoqués ci-dessus est fonction de la taille m du tableau, du nombre k de fonctions de hachage utilisée, et du nombre d'éléments précédemment insérer dans le filtre.

Attendus :

• Implémenter plusieurs variantes des filtres de Bloom utilisant pour le « tableau » de booléens :

- Un simple tableau
- Un ArrayList
- Une LinkedList

• Implémenter un banc d'essai (benchmark) permettant de

- Comparer les temps d'exécution de la recherche pour chacune de ces implémentations (on fixera alors k, m et le nombre d'éléments ajoutés dans le filtre)
- Pour l'une des implémentations, analyser le taux d'erreur du test d'appartenance en fonction de k et m - par exemple, si n est le nombre d'éléments dans le filtre, prendre k=1,3 et 5, et m=1%, 5% et 10% de n)

Réalisation :

Recherches sur le principe du filtre de Bloom :

On a un tableau de booléens à False. Une liste de mots qu'on va insérer dans le tableau. Des fonctions de hachage pour hacher de plusieurs manières un mot à insérer. Lorsqu'on hache un mot avec les fonctions de hachage, on obtient des index compris dans la taille du tableau qui seront les index qui passeront à True. Après avoir insérer les mots dans le tableau on peut vérifier leur présence.

On peut ensuite prendre des mots que nous n'avons pas insérer dans le tableau, calculer leur index avec les fonctions de hachage et voir si ces index sont à True. Si c'est le cas alors cela veut dire que le mot est inséré alors que non, c'est un faux-positif car c'est l'insertion d'autres mots qui a provoqué le passage à True des index du mot. Donc il y a une certaine probabilité de faux-positif, en revanche si le mot est indiqué absent, c'est sûr.

Réalisation du code :

- Création de la classe abstraite du filtre de Bloom
- Création des classes pour faire les différentes structures de données (Tableau simple, ArrayList, LinkedList)
- Création des classes filtre de Bloom pour chaque structure
- Création des classes abstraites :
  - Constructeur pour un filtre avec une taille de tableau et le nombre de fonction de hachage
  - Méthode pour afficher la structure de données
  - Méthode qui va ajouter un mot à la structure
  - Méthode qui vérifie la présence d'un mot
  - Méthode qui calcule la probabilité de faux-positif
- Implémentation des méthodes précédentes dans les classes FiltreTableauSimple, FiltreArrayList et FiltreLinkedList.
- Implémentation des méthodes suivantes, dans les classes TableauSimple, ArrayListBloom et LinkedListBloom :
  - Constructeur de la structure de données
  - Méthode pour récupérer la taille de la structure
  - Méthode qui récupère la valeur d'une case
  - Méthode qui passe à True une case
  - Méthode qui affiche la structure sous forme de 0 et 1
- Création de la classe Main qui teste le filtre de Bloom pour chaque structure :
  - Méthode principale qui gère les inputs permettant de choisir les valeurs désirées (Ban d'essai, taille structure, nombre de mots, nombre de fonction de hachage). Lance le test pour chaque structure et renvoie les temps d'exécution et la probabilité de faux-positif.
  - Méthode de Test qui lancer le filtre de Bloom pour chaque structure.
  - Méthode qui génère des mots aléatoires avec une taille maximum.

J'ai détaillé plus précisément le fonctionnement de mon code dans la javadoc et les commentaires.

Conclusion :

On constate qu'il y a une différence de temps d'exécution entre les différentes structures de données dû à leur fonctionnement. Pour le stockage de données et y accéder l'ArrayList est plus efficace que la LinkedList. En revanche pour la modification de données c'est l'inverse car la LinkedList comme son nom l'indique est une liste chainée, il y a donc des liens entre les éléments permettant un accès plus rapide.

Le temps d'exécution varie en fonction de divers paramètres, tel que la taille de la structure. On constate, dans notre cas, où l'on insère des éléments et on en modifie, avec une taille de structure grande, plusieurs fonctions de hachage et un gros pourcentage de mot insérer, la LinkedList est plus lente que les 2 autres structures. Pour la probabilité de faux-positif qu'on calcule pour chaque structure, elle se rapproche bien de la probabilité théorique au préalable calculée.

Exemple résultat :