Algorithmique et structures de données : Mission 5

Groupe 1.2: Ivan Ahad - Jérôme Bertaux - Rodolphe Cambier
Baptiste Degryse - Wojciech Grynczel - Charles Jaquet

21 novembre 2014

Question 1

Question 2 (Baptiste Degryse) L'algorithme de Knuth-Morris-Pratt, contrairement à l'algorithme de Boyer-Moore, a une sorte de mémoire qui va lui permettre d'identifier les caractères posant régulièrement problème, et de s'adapter afin de limiter le nombre de comparaisons. Il va éviter de refaire x fois la même erreur en comparant des lettres sans importance.

La complexité temporelle de cet algorithme est O(m+n), m et n étant la longueur des chaines de caractères à comparer contre une complexité temporelle O(n*m) dans l'algorithme Brute Force. Cette différence est due à la mémoire de la failure fonction dans KMP. Cette mémoire permet de ne comparer que les caractères les plus problématiques. Exemple de l'utilisation de cette fonction:

j	0	1	2	3	4	5
$P[j]$	a	b	a	c	a	b
$f(j)$	0	0	1	0	1	2

Question 3 (Charles Jacquet) Premièrement, on utilise les tries lorsque l'ont a besoin de faire une recherche d'un string dans un texte. Le problème posé est bien le même que pour la question précédente, mais la façon de faire les choses est totalement différente, c'est-à-dire que pour le KMP, on travail sur le string pattern, tandis qu'ici, on travail directement sur le texte, ce qui nous permet d'ailleurs, de faire des recherches différentes en analysant une seule fois le texte.

- Compressed trie : c'est un trie sauf que lorsqu'un noeud à un seul enfant, on rassemble le noeud et son enfant pour éviter ce qu'ils appellent les "redondances".
- Gain de place : Il faut sauvegarder moins de noeud, de plus, il stock les string en dehors de l'arbre, ce qui signifie que dans l'arbre, la complexité spaciale d'un élément est en O(1).
- Relation entre un Trie et Suffix Tree : Un suffix tree, est un trie pour lequel tous les strings stocké dans la collection sont suffix d'un String X.

Question 4 (Grynczel Wojciech) L'algorithme de construction d'un code de Huffman utilise une file de priorité. Est-il avantageux qu'il s'agisse d'une file de priorité adaptable ? Pourquoi ?

Pas vraiment, la fille de priorité adaptable fournit 3 méthodes en plus :

- Entry remove(Entry e);
- Object replaceKey(Entry e, Object k);
- Object replaceValue(Entry e, Object x);

Aucune de ses méthodes ne sera pas utilisée dans l'algorithme de huffman, on supprime toujours les valeurs avec la méthode removeMin();

L'utilisation d'une file de priorité est-elle indispensable pour pouvoir construire un code de Huffman ? Pouvez-vous imaginer une autre solution en utilisant un algorithme de tri ?

C'est possible de construire un code de Huffman sans la fille de priorité. Par exemple avec une ArrayList qui sera triée après chaque opération d'insertion.

Sa complexité calculatoire serait-elle meilleure que l'algorithme original ? Pourquoi ?
La complexité calculatoire ne sera pas meilleure. Si on utilise une ArrayList on sera obligé de retrier la liste après chaque insert(K key, V velue) afin garder les éléments dans le bon ordre.

Question 5 (Ivan) Il y a trois étapes dans l'algorithme du Huffman Coding :

- L'algorithme calcule d'abord la fréquence de chaque caractère se trouvant dans un string X, et initialise une Priority Queue permettant de stocker les valeurs nécessaires pour créer le coding.
- Pour chaque caractère distinct se trouvant dans le string X, l'algorithme crée un arbre binaire T avec un seul noeud, contenant le caractère en question. Ensuite, l'algorithme insert T dans la priority Queue, à la clé qui est égale à la fréquence de ce caractère dans le string X.
- Enfin, la boucle while de l'algorithme prend les deux arbres binaires possédant les plus petites fréquences et les fusionne, et ce jusqu'à qu'il ne reste plus qu'un arbre binaire dans la Priority Queue. Chaque itération de cette étape se fait en O(log d) comme complexité temporelle, où d représente le nombre de caractères distincts dans le string X, et ce à l'aide d'une Priority Queue représentée par un Heap. En sachant que cette opération prend deux arbres pour n'en retourner qu'un, et en sachant que cette opération est répétée autant de fois qu'il y a de caractères distincts dans le String X moins une fois, on en déduit que cet algorithme a une complexité temporelle de O(n+d*log d) où d représente le nombre de caractères distincts.

Question 6 (Bertaux Jérôme) Un algorithme de décompression d'un code de Huffman peut se décrire en deux étapes formant une fonction récursive. A chaque itération :

- Soit le noeud de l'arbre n'est pas une feuille donc on rappel la fonction avec le prochain bit et le fils de gauche si le bit actuel est 0 ou le fils de droite si le bit actuel est 1.
- Soit le noeud de l'arbre est une feuille, on note alors l'élément se trouvant dans celle-ci. Et on rappel la fonction avec le bit suivant et la racine de l'arbre.

La complexité calculatoire dépend du nombre de bits (n) et est donc de O(n). Pour effectuer l'algorithme nous avons besoin de l'arbre de priorité, d'un texte codé sur forme d'une liste de bit.