Projet LSINF1121 - Algorithmique et structures de données

-

Rapport intermédiaire Mission 5

Groupe 26

Laurian Detiffe (6380-12-00)

Sundeep Dhillon (6401-11-00)

Alexis Macq (5910-12-00)

Xavier Pérignon (8025-11-00)

Thibaut Piquard (4634-13-00)

Thomas Wyckmans (3601-12-00)

Année académique 2015-2016

Questions et réponses

1.
2.
3.
4.
5.
6.
7.
8.
9.
10. Quelles sont les différentes étapes d'un algorithme de compression de texte qui prend en entrée un texte et fournit en sortie une version comprimée de ce texte à l'aide d'un codage de Huffman ? Soyez précis dans votre description en isolant chaque étape du problème. Précisez notamment pour chaque étape les structures de données utiles et la complexité temporelle des opérations menées. (Xavier)

    La méthode de Huffman consiste à remplacer les caractères les plus fréquents par des codes courts et les caractères les moins fréquents par des codes longs. La phase d'encodage se compose de trois étapes :

    (a) **Comptage des fréquences des caractères** : cette étape consiste à parcourir tous les caractères du texte et à calculer le nombre d'occurence de chaque lettre dans ce texte. Si n est le nombre de caractères dans le texte, alors la complexité de cette étape est de $O(n)$.

    Compte(fentrée)

    ```
    1 pour a ∈ A faire
    2     fréq(a) ← 0
    3 tantque non fdf(fentrée) et a est le prochain caractère faire
    4     fréq(a) ← fréq(a) + 1
    5 fréq(FIN) ← 1
    ```

    (b) **Construction du code préfixe** : Un code préfixe est un ensemble de mots tel qu'aucun mot de l'ensemble n'est préfixe d'un autre mot de l'ensemble. Un code préfixe sur l'alphabet binaire $\{0, 1\}$ peut être représenté par un tri qui est un fait un arbre binaire dont tous les nœuds internes ont exactement deux successeurs. Les feuilles sont étiquetées avec les caractères originaux, les branches par 0 ou 1 et les chemins depuis la racine jusqu'aux feuilles épellent

les codes des caractères originaux. L'utilisation d'un code préfixe assure que les codes sont bien représentés par les feuilles. Par convention, le fils gauche d'un nœud est étiqueté par 0 et le fils droit par 1.

Const-arbre()

```
1  pour a ∈ A ∪ {FIN} faire
2    si fréq(a) ≠ 0 alors
3      créer un nœud t
4      poids(t) ← fréq(a)
5      étiq(t) ← a
6  lfeuilles ← liste des nœuds dans l'ordre croissant des poids
7  larbres ← liste vide
8  tantque LONGUEUR(lfeuilles) + LONGUEUR(larbres) > 1 faire
9    (g, d) ← extraire les 2 nœuds de plus faible poids parmi
       les 2 premiers éléments de lfeuilles et
       les 2 premiers éléments de larbres
10   créer un nœud t
11   poids(t) ← poids(g) + poids(d)
12   gauche(t) ← g
13   droit(t) ← d
14   insérer t à la fin de larbres
15 Retourner t
```

(c) **Codage du texte** : Après la construction de l'arbre, il est possible de retrouver le code de chaque caractère par un parcours en profondeur de l'arbre.

Const-code(t, ℓ)

```
1 si t n'est pas une feuille alors
2   temp[ℓ] ← 0
3   CONST-CODE(gauche(t), ℓ + 1)
4   temp[ℓ] ← 1
5   CONST-CODE(droit(t), ℓ + 1)
6 sinon code(étiq(t)) ← temp[0 . . ℓ − 1]
```

Cette étape nécessite de stocker les codes de chaque caractère avant le code du texte.

Code-arbre(fsortie, t)

```
1 si t n'est pas une feuille alors
2    écrire un 0 dans fsortie
3    CODE-ARBRE(fsortie, gauche(t))
4    CODE-ARBRE(fsortie, droit(t))
5 sinon écrire un 1 dans fsortie
6    écrire binaire(étiq(t)) dans fsortie
```

On peut ensuite coder le texte.

Code-texte(fentrée, fsortie)

```
1 tantque non fdf(fentrée) et a est le prochain caractère faire
2    écrire code(a) dans fsortie
3 écrire code(FIN) dans fsortie
```

La complexité de cette étape est le la complexité d'un parcours d'un arbre binaire, c'est-à-dire en $O(nlog(n))$.

11.

12. **[Question liée spécifiquement au problème posé]** En quoi les deux classes qui vous sont fournies, `InputBitStream` et `OutputBitStream`, peuvent-elles être utiles pour le problème de compression et de décompression avec un codage de Huffman ? La postcondition de la méthode `close` dans la classe `OutputBitStream` précise notamment que *si le nombre de bits déjà écrits ne correspond pas à un multiple de 8 (un octet), des bits à 0 sont écrits pour compléter l'octet courant.* Quand la situation décrite peut-elle se présenter ? Quelle est la conséquence de cette postcondition sur votre programme de compression de texte ? Quelle est la conséquence de cette postcondition sur votre programme de décompression ? (Xavier)

    Les classes `InputBitStream` et `OutputBitStream` ont pour but de lire et d'écrire des bits, dont l'entrée et la sortie standard sont orientés vers les flux de caractères encodés en Unicode. La valeur d'un *int* sur la sortie standard est une séquence de caractères (représentation décimale), tandis que la valeur d'un *int* sur `OutputBitStream` est une séquence de bits (représentation binaire). Ces classes fondent leur I/O sur 8-bit bytestreams. Les données sur l'entrée standard ne sont pas nécessairement alignés sur les frontières d'octets. La méthode `close` n'est pas indispensable mais, pour une terminaison propre, les utilisateurs devraient appeler

close pour indiquer qu'il n'y a plus de bits à lire (pour la compression). Pour la décompression, la méthode *close* est essentielle. En effet, les utilisateurs doivent appeler *close* pour veiller à ce que tous les bits spécifiés avec les appels *write* s'écrivent dans la *bitstream* et que le dernier octet se remplisse avec des 0 afin que output s'aligne avec le système de fichiers.

13.

14.