Mission 5 : Rapport Final

Groupe 2.2

Vendredi 28 Novembre 2014

Introduction

Il nous a été demandé de réaliser un programme de compression et de décompression de fichiers. Ce programme devait s'inspirer de l'algorithme de Huffman. Le codage de Huffman faisant appel à une file de priorité, il était nécessaire d'implementer cette file de la manière la plus efficace possible.

Choix d'implémentation

Le programme comporte 5 classes, InputBitStream, OutputBitStream, Compress, Decompress, HTree. Les deux premières classes citées, nous ont été fournies. Vous trpuverez ci-dessous une explication pour chacune d'elles.

InputBitStream et OutputBitStream (Question 7)

De manière générale, des données textuelles sont codées soit en utilisant la norme ASCII, soit la norme UNICODE, qui font toutes deux correspondre un code binaire de 7 ou 16 bits à chaque caractère. L'algorithme d'Huffman va remplacer cette suite de bits de taille fixe par une suite de taille variable en fonction de la fréquence de chaque caractère dans le texte à traiter, et cela de manière optimale (les chaines de bits les plus courtes correspondront aux caractères apparaissant le plus souvent dans le texte et inversément).

Les deux classes InputBitStream et OutputBitStream permettent d'écrire un fichier directement en binaire, donc de contourner l'encodage ASCII ou UNICODE par défaut, et sont de ce fait nécessaire à la réalisation de l'algorithme d'Huffman.

Cependant, le fait d'utiliser des correspondances de taille variable entraine que la taille totale peut ne pas etre un multiple de 8 bits (1byte) et donc etre complété par des 0 pour terminer le dernier byte entammé. Pour gérer ce problème, nous avons intégré au header une suite de bits indiquant le nombre de caractères dans le fichier. Lorsque ce nombre de caractères déchiffrés est atteint à la décompression, cela signifie que l'ensemble des données ont été décompressées, et le programme ne tient donc pas compte des bits additionnels.

Compress

La classe Compress va lire une première fois le fichier source caractère par caractère pour créer une table de fréquence, qui est une hashMap ayant pour clé un caractère et comme valeur une fréquence. Avec les informations de la table de fréquence, nous créons l'arbre de Huffman et un table de traduction, c'est aussi une hashMap ayant pour clé le caractère compressé et pour valeur un string contenant la valeur du caractère compressé.
Nous créons un fichier ayant pour entête les informations suivantes :

$|byte(8b) -> 3siversion1.3$
$|short(16b) -> numberofcharactersindictionary(-> n)$
$|long(64b) -> numberofcharactersinfile$
$n * |char(16b) -> character$
$|int(32b) -> codeword, left - paddedwith00..01$
$|(variablelength, binary) -> compresseddata$

Ensuite, nous parcourons une seconde fois le fichier sources à l'aide de la table de traduction pour transcrire les différents caractères rencontrés en caractères compressés sur le fichier compressé.

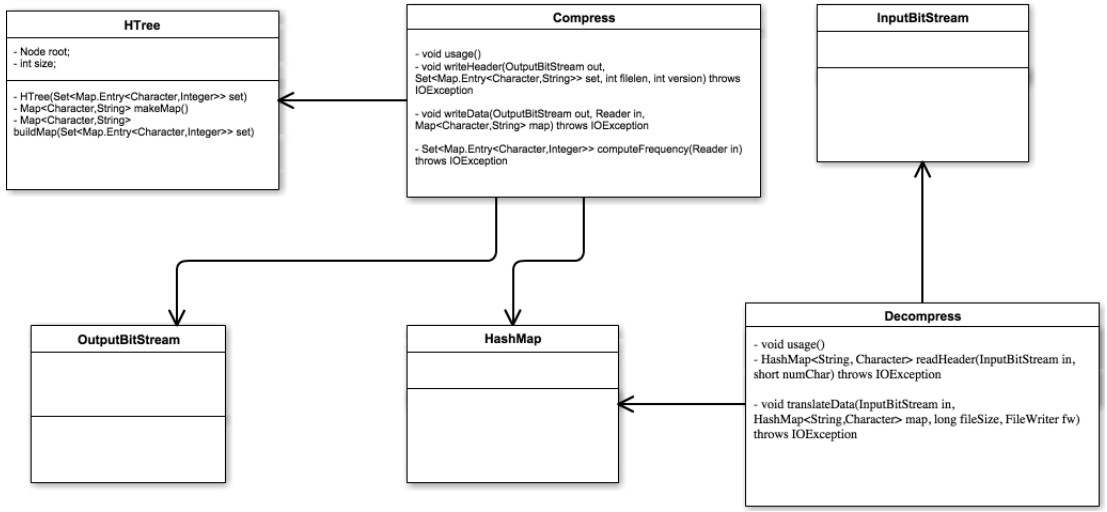
Decompress

Dans un premier temps, nous lisons les informations contenue dans l'entête du fichier compressé. A partir de ces informations, nous créons une nouvelle table de traduction, ayant comme clé un string représentant la valeur d'un caractère compressé et comme valeur le caractère référencé. Nous parcourons le fichier bit par bit et à chaque nouveau bit, nous cherchons une correspondance entre le buffer de bits et une clé de la table de traduction. Lorsqu'il y a correspondance, nous écrivons la valeur de la table de traduction dans le fichier décompressé et nous vidons de buffer. Nous continuons ainsi de suite jusqu'à la fin du fichier qui est connue car nous connaissons le nombre de caractères présents dans le fichier grâce à l'entête.

HTree

La classe HTree utilise un SetMap contenant tous les caractères utilisés dans le fichier ainsi que leur fréquence pour créer un arbre de Huffman à l'aide d'une priorityQueue. L'abre de Huffman ainsi créé est ensuite parcouru de manière récussive pour créer un Hashmap contenant les caractères comme clé et comme valeur un string représentant ce caractère compressé. Nous avons pris la décision d'implémenter complétement la classe HTree et donc de ne pas étendre la classe générique TreeSet. L'avantage de cette approche, c'est que cette classe correspond exactement à ce qu'on attend d'elle. Par contre, l'inconvénient c'est que l'on perd en portabilité.

Diagramme de classe (Question 8)



Test (Question 9 et 10)

Le ratio de compression observer sur différent fichiers de texte varie, et en effet lié à la taille du fichier compressé. Nous pouvons le voir grâce à ces différent chiffre :

Avant	Après
3.6 ko	2.7 ko
1.6 Mo	1.1 Mo
8 Mo	5 Mo
11 Mo	7 Mo
14 Mo	9 Mo
17 Mo	10 Mo

Vu que nous avons utilisé l'algorithme de compression par codage de Huffman pour notre code, nous pouvons supposer que notre implémentation est normalement sans perte. Mais qu'est-ce que ça signifie un compression sans perte ?
Une compression est sans perte lorsqu'il y a autant d'informations avant et après la compression. Les fichiers compressés et ensuite décompressés doivent être en tout point identique.
L'approche que nous utilisons pour compresser les fichiers nous permet d'affirmer que la compression est sans perte. En effet, nous gardons dans l'entête du fichier compressé différentes informations permettant la bonne décompression de ce fichier, dans cette entête se trouve un "dictionnaire" nous permettant de faire correspondre chacun des caractères compressés avec son caractère source. La table de traduction a été construite avec l'ensemble des caractères rencontrées dans le fichier source.

La méthode la plus fastidieuse de vérifié que notre implémentation de compression et de décompression est sans perte serait de comparer caractère par caractère le fichier source et le fichier décompressé. D'après nous, la manière la plus efficace de tester la validité de cette affirmation est de comparer le nombre de caractères du fichier source et de fichier décompresser.

Conclusion

La décomposition des classes dans notre programme, nous a facilité la répartition des taches, l'utilisation de multiple classe permet aussi de pouvoir être plus modulable. Tant au niveau de la repartition du travail, qu'au niveau de l'ajout de nouvelles fonctions pour adapter notre programme.