

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'implémenter 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 trouverez 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 chaînes de bits les plus courtes correspondront aux caractères apparaissant le plus souvent dans le texte et inversement).

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 nécessaires à la réalisation de l'algorithme d'Huffman.

Cependant, le fait d'utiliser des correspondances de taille variable entraîne que la taille totale peut ne pas être un multiple de 8 bits (1byte) et donc être complétée par des 0 pour terminer le dernier byte entamé. 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.

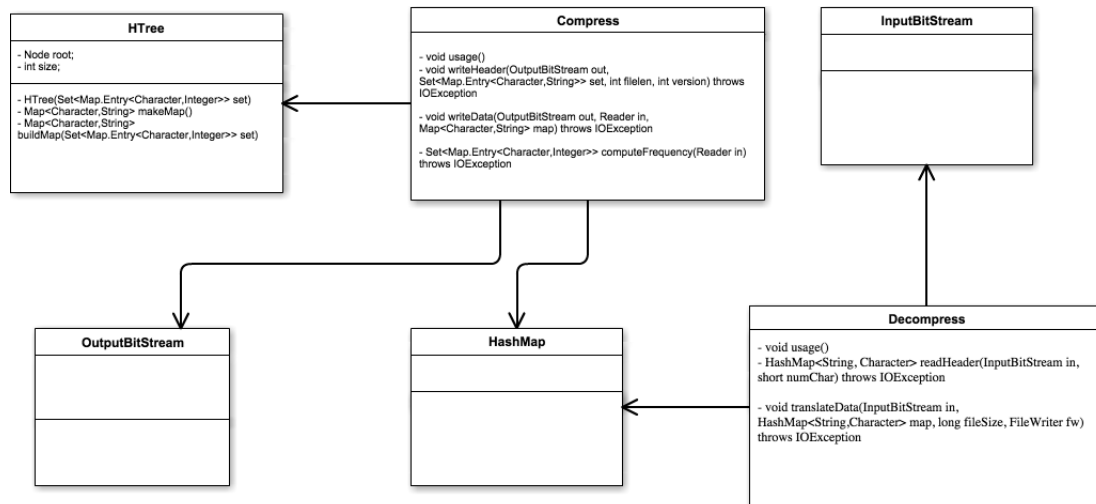
Compress

Decompress

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'arbre de Huffman ainsi créé est ensuite parcouru de manière récursive 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 complètement 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)

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 identiques.

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 une table de traduction 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 rencontrés dans le fichier source.

La méthode la plus fastidieuse de vérifier 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 tâches, l'utilisation de multiple classe permet aussi de pouvoir être plus modulable. Tant au niveau de la répartition du travail, qu'au niveau de l'ajout de nouvelles fonctions pour adapter notre programme.