ÉCOLE DE TECHNOLOGIE SUPÉRIEURE  
UNIVERSITÉ DU QUÉBEC

LOG320 – LABORATOIRE #1  
ALGORITHME DE HUFFMAN

PAR  
MARC-ANDRÉ DESTREMPES – AJ86290  
SAMUEL BEAUCHEMIN – AK06890

MONTRÉAL, 23 mai 2013

Table des matières

[Description de votre programme 3](#_Toc356590736)

[Analyse de complexité des algorithmes 4](#_Toc356590737)

[Description des problèmes rencontrés 5](#_Toc356590738)

[Description des améliorations que vous avez implémentées 6](#_Toc356590739)

# Description de votre programme

Figure - Diagramme de classe

Le programme se divise en 2 sections, l’encodage et le décodage. L’encodage est ce qui permet de compresser un fichier en utilisant l’algorithme de Huffman, tandis que le décodage permet de faire l’opération inverse. L’arbre de fréquence est stocké dans un objet *Tree* qui contient des *Node* et/ou des *Leaf*. Les *Leaf* représentent chacune des lettres avec leur fréquence, tandis que les *Node* sont l’intersection de deux objets, soit *Node* ou *Leaf* , et contient l’addition de la fréquence de ces derniers. La classe *MapValueComparator* permet de trier un objet de type *HashMap* par valeur en ordre décroissant. La classe *FileHelper* permet de gérer toutes les opérations qui ont rapport avec les fichiers textes, que ce soit pour la lecture ou l’écriture. La classe *TreeHelper* gère tout ce qui est en lien avec les manipulations d’arbre, soit la création ainsi que l’obtention de diverses informations. La classe *CompressHelper* permet d’effectuer les opérations de compression ou de décompression de fichiers textes.

L’encodage commence par la lecture du fichier texte correspondant au chemin passé en paramètre. Ensuite, l’arbre binaire est construit et permet ensuite d’acquérir chacun des codes pour les différents caractères retrouvés dans le fichier texte. Chacun des caractères ASCII dans le fichier est ensuite remplacé par son code correspondant. Finalement, la nouvelle chaîne de caractères binaires est écrite dans un nouveau fichier ainsi qu’un en-tête contenant les informations nécessaires pour reconstruire l’arbre binaire.

Le décodage commence par lire le fichier compressé et sépare l’en-tête du contenu. À partir de l’en-tête, il est possible de reconstruire l’arbre binaire afin de pouvoir réacquérir les informations sur les codes binaires de chacun des caractères. L’arbre est ensuite parcouru en fonction de la chaîne de caractère binaire pour retrouver chacun des caractères contenus dans le texte d’origine. Pour finir, lorsque toutes les opérations de décodage ont été exécutées, un nouveau fichier, identique à l’original, est créé en y insérant la chaîne de caractères décodée.

Le programme a comme limitation les 256 premiers caractères ASCII[[1]](#footnote-1).

# Analyse de complexité des algorithmes

|  |  |
| --- | --- |
| Module.Méthode | Complexité |
| TreeHelper.getInstance() |  |
| TreeHelper.createTree() |  |
| TreeHelper.getCharCodes() |  |
| TreeHelper.createTreeCode() |  |
| Leaf.getCharacter() |  |
| Tree.getFrequency() |  |
| Tree.compareTo() |  |
| Node.getLeftBranch() |  |
| Node.getRightBranch() |  |
| MapValueComparator.compare() |  |
| FileHelper.getInstance() |  |
| FileHelper.readTextFile() |  |
| FileHelper.readByteFile() |  |
| FileHelper.writeToTextFile() |  |
| FileHelper.writeToByteFile() |  |
| CompressHelper.getInstance() |  |
| CompressHelper.CompressFile() |  |
| CompressHelper.DecompressFile() |  |
| CompressHelper.getSortedFrequencyTable() |  |
| CompressHelper.encodeText() |  |
| CompressHelper.decodeText() |  |
| CompressHelper.encodeHeader() |  |
| CompressHelper.decodeHeader() |  |

Tableau 1 - Complexité des méthodes

# Description des problèmes rencontrés

Un premier problème rencontré a été l’écriture du fichier lors de l’encodage. Le contenu du nouveau fichier n’était pas écrit en binaire; les « 0 » et les « 1 » était écrit comme une chaîne de caractères. Alors, au lieu de se compresser, le fichier augmentait en taille. Le problème a été réglé alors que nous avons complètement réécrit notre algorithme de compression.

Un deuxième problème rencontré a été la limitation du nombre de fréquence d’un caractère. Le nombre maximal initial était de 256 pour un même caractère. Le problème a été réglé en changeant la façon dont l’information était stockée dans l’en-tête.

Un troisième problème rencontré a été l’information à stocker dans l’en-tête du fichier pour minimiser l’espace que celui-ci occupe tout en maximisant l’information. Nous avons donc opté pour garder les fréquences de chacun des caractères, mais uniquement ceux qui sont présents dans le fichier, et non pas les 256 premiers caractères ASCII.

Un quatrième problème rencontré a été de choisir la structure de données utilisé pour construire l’arbre. Une première version a été faite avec seulement des *Node* qui contenaient toute l’information. Cette version avait des problèmes lorsqu’il venait le temps de parcourir l’arbre en entier. Nous avons donc décidé d’en venir à une deuxième version, celle qui est contenu dans la Figure 1 - Diagramme de classe, qui contient une superclasse *Tree* et des enfants *Node* et *Leaf*.   
Un *Node* a nécessairement un enfant alors qu’un *Leaf* contient l’information sur un caractère.

# Description des améliorations que vous avez implémentées

D’abord, l’en-tête du fichier a été optimisé en enlevant les octets qui correspondaient à des caractères qui n’étaient pas présent dans le texte compressé. Alors, nous avons décidé d’utiliser les 32 premiers octets pour définir quels caractères étaient présents dans le texte. Ainsi, la taille de l’en-tête s’en trouvait réduite de plusieurs octets.

Une autre amélioration que nous avons implémentée est au niveau de la compression du contenu. Une fois que les caractères ont été encodés avec leur code respectif, ils ont tous été concaténé et diviser en petites chaînes de caractères de huit caractères de longueur afin de les transformer en octets. Ceci évitait d’avoir un octet pour chacun des codes.

1. <http://www.asciitable.com/> [↑](#footnote-ref-1)