Rapport de laboratoire

N° de laboratoire	1
Étudiant(s)	Israel Hallé Nicholas Gromko
Code(s) permanent(s)	HALI17049101 GRON16089007
Cours	LOG320
Session	Automne
Groupe	1
Professeur(e)	Pierre Dumouchel
Chargé(e) de laboratoire	Patrick Cardinal
Date	2013-10-9

Description du programme

Le programme réalisé lors du premier travail pratique en LOG320 permet de compresser ou décompresser un fichier à de l'aide l'algorithme d'Huffman.

Le programme est utilisable en exécutant la classe etsmtl.ca.log320.tp.Log320Tp1.

L'usage est:

```
huffman -compress|decompress filename
```

En utilisant `-compress`, une nouvelle version compressée du fichier du même nom avec l'extension .huf sera créé dans le même dossier que le fichier source.

En utilisant `-decompress`, une nouvelle version décompressée du fichier sans l'extension .huf sera créée dans le même dossier que le fichier source.

Analyse de complexité (asymptotique) des algorithmes

Compression

La compression peut être séparée en plusieurs opérations pour simplifier l'analyse. Ainsi, nous commençons par la génération de la table de fréquence, la construction de l'arbre binaire et finalement la compression des données.

Génération de la table de fréquence

Cette fonction itère les octets des données à compresser et compte la fréquence de chaque octet. Par la suite, nous trions les octets selon leur fréquence dans une liste chaînée.

L'analyse est basée sur la fonction etsmtl.ca.log320.tp.Log320Tp1.Compressor.getNodeList où N est le nombre d'octets dans le fichier qu'il faut compresser.

getSortedFrequencyList(données[1..N])	**O**	**Ω**
tableOccurence[1..256] ← { 0 }	O(1)	Ω(1)
pour chaque *b* ∈ *données*	O(N)	Ω(N)
incrementer *tableOccurence*[*b*]	O(N)	Ω(N)
pour chaque (*octet, fréquence*) ∈ *tableOccurence*	O(1)	Ω(1)
si *octet* != 0	O(1)	Ω(1)
insèrer (*octet, fréquence*) dans *listeTriée*	O(1)	Ω(1)
Complexité: Θ(N)	O(N)	Ω(N)

Noter que le nombre d'insertion dans la liste chaînée triée est limité à 256 insertions, donc nous pouvons considérer la complexité de l'insertion à l'intérieur de la boucle comme étant indépendante de N et constante, donc O(1).

Construction de l'arbre binaire

Cette fonction prend la liste chaînée triée générer par la table de fréquence et crée un arbre binaire en suivant l'algorithme de Huffman.

L'analyse est basée sur la fonction etsmtl.ca.log320.tp.Log320Tp1.Compressor.buildTree où N est le nombre d'éléments dans la liste triée.

getHuffmanTree(listreTrié[1..N])	**O**	**Ω**
noeudCourant ← *listeTriée*[0]	O(1)	Ω(1)
tant que *noeudCourant* a un noeud suivant	O(N)	Ω(N)
noeudSuivant ← noeud suivant de *noeudCourant*	O(N)	Ω(N)
Créer *noeudParent* avec enfant *noeudCourant* et *noeudSuivant*	O(N)	Ω(N)
noeudParent.fréquence← *noeudCourant.frequence+noeudSuivant.frequence*	O(N)	Ω(N)
Retiré *noeudCourant* et *noeudSuivant* de *listeTriée*	O(N)	Ω(N)
insérer *noeudParent* dans *listeTriée*	$O(N^2)$	Ω(N)
noeudCourant ← *noeudParent*	O(N)	Ω(N)

Compression des données

Cette fonction utilise les fonctions précédentes afin de générer l'arbre de Huffman nécessaire à la compression des données.

L'analyse est basée sur la fonction etsmtl.ca.log320.tp.Log320Tp1.Compressor.compress où N est le nombre d'octets dans le fichier qu'il faut compresser.

compress(données[1..N])	**O**	**Ω**
listeTriée ← getSortedFrequencyList(*données*)	O(N)	Ω(N)
huffmanTree ← getHuffmanTree(*listeTriée*)	O(1)	Ω(1)
écrit *huffmanTree*	O(1)	Ω(1)
pour chaque *octet* ∈ *données*	O(N)	Ω(N)
encodage ← chemin de *octet* dans *huffmanTree*	O(N)	Ω(N)
écrit *encodage*	O(N)	Ω(N)
Complexité: Θ(N)	O(N)	Ω(N)

getHuffmanTree est considéré comme étant O(1) étant donné que la taille de *listeTriée* est indépendante de N.

Décompression

La décompression est beaucoup plus simple. Celle-ci débute par la reconstruction de l'arbre binaire et est terminée par la décompression des données.

L'analyse est basée sur la fonction etsmtl.ca.log320.tp.Log320Tp1.Uncompressor.uncompress.

decompress(données[1..N])	**O**	**Ω**
huffmanTree ← lire arbre d'huffman dans *données*	O(1)	Ω(1)
noeudCourant ← Noeud racine de *huffmanTree*	O(1)	Ω(1)
pour chaque *bit* ∈ *données*	O(N)	Ω(N)
si *bit* = 1	O(N)	Ω(N)
noeudCourant ← enfant droit de *noeudCourant*	O(N)	O(N)
sinon	O(N)	O(N)
noeudCourant ← enfant gauche de *noeudCourant*	O(N)	O(N)
si *noeudCourant* est une feuille	O(N)	O(N)
écrit *octet* de *noeudCourant*	O(N)	O(N)
noeudCourant ← Noeud racine de *huffmanTree*	O(N)	O(N)
Complexité: Θ(N)	O(N)	Ω(N)

Ainsi la complexité de notre programme pour la compression et la décompression est de Θ(N) où N est la taille des données à compresser.

Description des problèmes rencontrés

Parmi les problèmes rencontrés, il y avait celui de ne pas interpréter les derniers bits du dernier octet dans le cas où les bits compressés n'utilisent pas le dernier octet au complet. Ainsi, lors de la décompression il aurait été possible que des '0' soient lus et interprétés comme certains octets décompressés. Le fichier décompressé aurait donc été un peu plus gros que le fichier original.

La solution utilisée a été d'ajouter un dernier octet au fichier compressé qui contient le nombre de bits à ignorer. Ainsi, lors de la lecture du fichier, nous regardons toujours deux octets plus loin au cas où la fin du fichier est atteinte. Dans ce cas, nous récupérons le nombre de bits à ignorer et utilisons cette valeur afin de ne pas lire les bits d'extra.

Une solution alternative qui pourrait nous permettre de gagner un octet dans quelques cas aurait été d'ignorer tout ce qui suit le dernier '1'. Ainsi, si un octet contient deux bits à décompresser, le troisième bit serait définit à '1' et les bits suivants à '0'. Lors de la lecture, nous pouvons donc vérifier si la fin du fichier est atteinte et chercher le dernier '1' pour ne pas lire de bit d'extra.

Description des améliorations que vous avez implémentées

Une amélioration que nous avons ajoutée au cours de l'implémentation a été l'écriture de l'arbre dans le fichier compressé. Au départ, nous avons utilisé une méthode simple où nous écrivions l'encodage d'Huffman pour les 256 octets en ordre séparé par un caractère d'échappement. Ainsi, le premier encodage lu était celui de l'octet 0x0, le second, 0x1, etc.

Par contre, cette méthode est peu efficace pour les fichiers qui n'utilisent pas toutes les valeurs d'un octet (par exemple, les fichiers texte qui sont limités aux plages de l'ASCII). La taille de l'entête avec cette méthode variait au-dessus 512 à 768 octets selon la structure de l'arbre.

La méthode alternative utilisée consiste à encoder la structure de l'arbre et par la suite y écrire la valeur en ordre de chaque feuille de l'arbre. La structure est sérialisée en parcourant l'arbre binaire d'étage en étage, de gauche à droite, et d'écrire le bit '0' si l'élément est un noeud ou '1' si c'est une feuille. Par la suite, toujours dans le même ordre, nous reparcourons l'arbre binaire et écrivant dans le fichier la valeur de l'octet de chaque feuille parcouru.

L'avantage de cette méthode est que la reconstruction de l'arbre est beaucoup plus simple et la taille de l'entête se retrouve réduite et s'adapte à la plage de valeur utilisée par le fichier à compresser. Ainsi, dans la taille de l'entête varie de 2 octets, dans le cas d'un fichier avec un seul caractère, à 320 octets dans le pire des cas.