Compression de texte par le codage de Huffman

Gauthier Jolly – Thibault Meunier | Projet P302 | TR56

29 Janvier 2015

Ce rapport de projet décrit les étapes du développement d'une application de compression basée sur le codage de Huffman. Après une description précise des objectifs à remplir, nous exposerons brièvement comment nous avons choisi d'implanter l'algorithme de Huffman, puis nous expliquerons quels ont été les problèmes rencontrés et ce que ce projet nous à apporté.

Table des matieres

1 Introduction

Le codage de Huffman est une méthode de compression bien connue. Elle se base sur la construction d'un arbre binaire fonction de la fréquence des symboles du fichier à compresser.

Le but de ce projet est de concevoir un programme qui prend en entrée un fichier de texte à compresser et qui retourne un fichier compressé sans perte à l'aide de l'algorithme de Huffman.

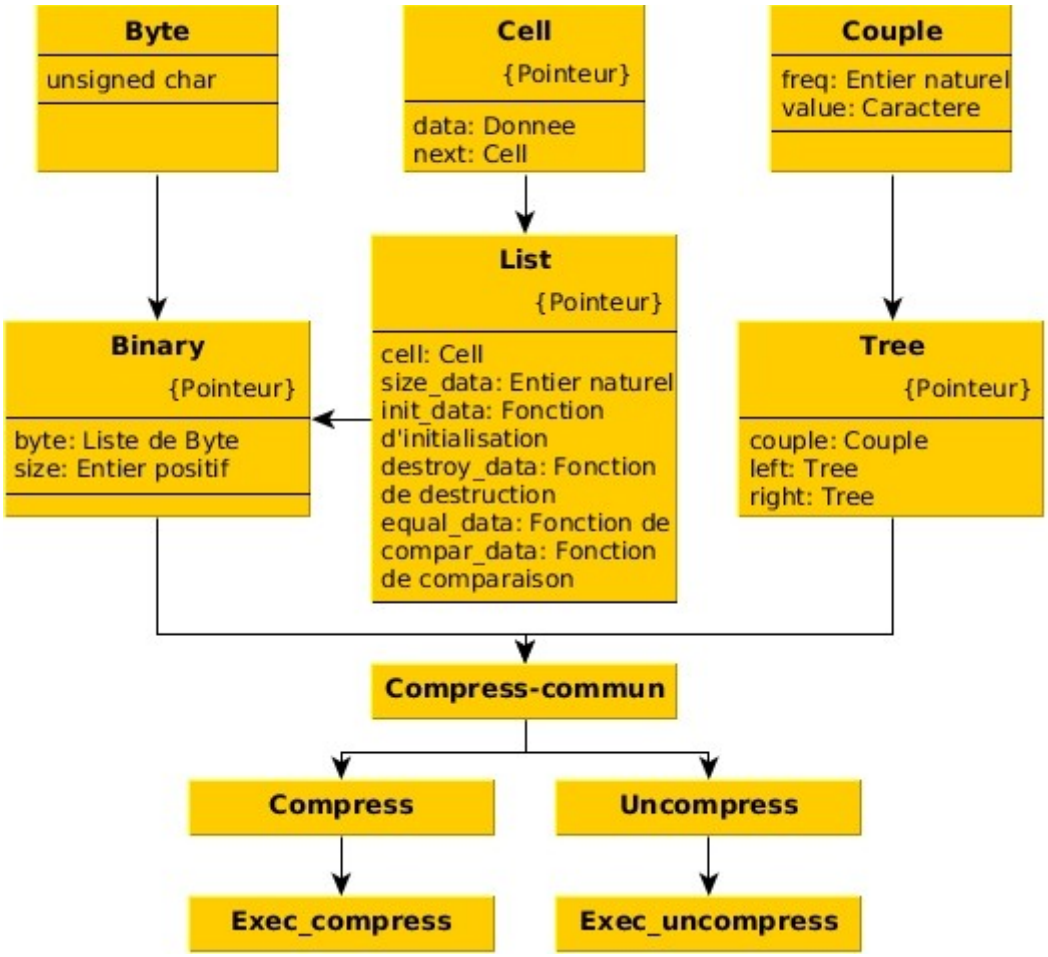
En terme pratique, l'objectif est de pouvoir, par l'intermédiaire d'une commande réduire la taille d'un fichier texte, puis de retrouver le texte original à l'issue de la décompression. Cette commande se présentera comme suit :

```
1   > compress <fichier a compresser> [nom du fichier compresse]
2   > uncompress <fichier a decompresser> [nom du fichier decompresse]
```

Le nom du fichier destination est facultatif. Si il n'est pas spécifié, le fichier compressé se terminara par ".hf", et le fichier decompressé par ".nhf"

2 Structures des modules

2.1 Architecture des modules



2.2 Choix

Au cours de l'elaboration de ce projet, nous avons été amené à faire différents choix techniques.

- Module `List`, pouvant contenir des objets de types variés
- Module `Binary`, pour traiter les opérations bit à bit
- Une table de correspondance calculée une fois en début de programme, afin d'éviter des parcours répétitif de l'arbre
- Pour la lecture et l'écriture dans les fichiers, l'utilisation d'un tampon nous permet de faciliter l'interaction avec les bits

3 Raffinages

3.1 Types de donnees

3.1.1 Cell

Cellule, renferme une donnée ainsi qui l'indice vers la suivante

```
1   struct Cell{
2      void* data;
3      struct Cell* next;
4   };
```

3.1.2 List

Liste chainée d'éléments, ordonnée selon une fonction fournie par l'utilisateur

```
1   struct List{
2      Cell cell;                          //Premiere cellule
3      size_t size_data;                   //Taille de la donnee
4      void (*init_data)(void**);          //Constructeur
5      void (*destroy_data)(void*);        //Destructeur
6      void (*equal_data)(void*, void*);   //Operateur d'egalite =
7      int (*compar_data)(void*, void*);   //Operateur de comparaison ==, <, >
8   };
```

3.1.3 Byte

Octet codé sur un unsigned char

```
1   typedef unsigned char Byte;
```

3.1.4 Binary

Suite de bits, caractérisé par une liste d'octets et un nombre de bits

```
1   struct Binary{
2      List byte;
3      size_t size;
4   };
```

3.1.5 Tree

Arbre, servant de base à l'élaboration de l'algorithme de compression d'Hufmann

```
1   struct Tree{
2      Couple couple;
3      struct Tree* left;
4      struct Tree* right;
5   };
```

3.2 Presentation des principaux algorithmes

3.2.1 Raffinage de Compress

R0: Compresser un fichier texte

R1: "Compresser un fichier texte"

```
Déterminer la fréquence des caractères du fichier et leur nombre
Construire l'arbre de huffman
Ecrire l'entête du fichier compressé
Encoder le fichier
```

R2: "Construire l'arbre de Huffman"

```
Trier le tableau des Couples du tableau fréquence
Construire une liste chainée d'arbres à partir du tableau
Tant que la liste contient plus d'un élément
    Fusionner les deux premier arbres de la liste
FinTQ
```

R2: "Écrire l'entête du fichier compressé"

```
Écrire la taille du texte et de l'arbre
Écrire les caractères et la description de l'arbre (parcours
infixe)
```

R2: "Encoder le fichier"

```
Tant qu'il y a des caractères dans le fichier d'entrée Faire
    Concaténer le code du caractère au buffer
    Tant que le buffer contient au moins un octet complet
        Écrire dans le fichier compressé cet octet
    FinSi
FinTQ
```

R3: "Fusionner les deux premiers arbres de la liste"

```
Créer un arbre ayant comme fils droit le premier arbre de la liste
et comme fils gauche le second (cf raffinage Tree)
Placer l'arbre dans la liste
```

3.2.2 Raffinage de Uncompress

R0 : Decompression du fichier fourni en argument

R1 : Decompression du fichier fourni en argument

```
Ouverture du fichier
Lecture de l'en-tete
Decodage du fichier
```

R2 : Lecture de l'en-tete

```
Determination de la taille
Construction de l'arbre de Huffman
```

R2 : Decodage du fichier

Repeter

Ecriture du caractere

Jusqu'a Caractere = Fin de Fichier

R2 : Construction de l'arbre de Huffman

A chaque fois que l'on tombe sur le couple 01 dans la description de l'arbre, il y a ajout d'une feuille.

Les 00 cree des nœuds vides

R3 : Ecriture du caractere

Tant que le Binaire ne correspond a aucun code (feuille dans l'arbre)

Rajout le dernier bit au binaire

Stockage du decode caractere dans le fichier

4 Tests et problemes

Pour les points apportés ci-dessous, une grande partie des solutions a été trouvé avec l'utilisation de l'outil Valgrind (http://valgrind.org/)

4.1 Tests

Afin d'optimiser l'interraction entre les différents modules, ainsi que de minimiser le temps de tests, nous avons testé chaque module indépendament au fur et à mesure de l'avancement du projet. On retrouve les différents fichiers de tests dans `tests-<module>.c`

Pour ce qui est du programme dans sa globalité, son évaluation s'est divisée en plusieurs parties

- Sur plusieurs exemples simples : 1 caractere, quelques mots
- Sur des textes plus conséquents, avec des caractères parfois exotiques
- Sur deux textes de grandes tailles : Hamlet, Le Petit Prince

Ces tests ont été générés dans le répertoire `tests` par le fichier `test.c`.

4.2 Problemes rencontrés

- La gestion d'une liste d'arbre fut quelque peu délicate, et n'apportait pas une efficacité significativement plus élevée, nous avons donc décidé de passer outre et de lui préférer un tableau
- Les opérations sur les binaires ont été difficiles à définir, tout particulièrement la gestion de la taille à leur issue (les 0 en début de binaire par exemple)
- La manipulation du tampon (lecture/écriture), qui conduisait parfois à une altération des données
- L'allocation et la désallocation des multiples objets manipulés

5 Conclusion

5.1 Avancement du projet

Les commandes `compress` et `uncompress` sont pleinement fonctionnelles. Leur utilisation se fait comme décrite plus haut dans l'introduction.

5.2 Réutilisabilité des modules

Tant que l'ensemble des dépendances d'un module est satisfaite, il est possible de l'intégrer dans un nouveau projet. Par exemple, si un projet à besoin d'une liste d'éléments, il suffit d'importer le module `List`.

Ceci peut être utile pour la compression : en effet, il suffit d'imclure le module, et d'appeler la fonction `compress(in, out)` avec deux fichiers, et le travail se fait.

5.3 Performance

Afin de comparer l'efficacité de l'algorithme de compression, on lui oppose la compression zip de 7zip sur plusieurs points. On teste les deux programmes sur Hamlet, disponible dans `hamlet.html` .

5.3.1 Taux de compression

On définit ici le taux de compression comme le rapport de la volume final sur la volume initial.

`Compress` : 65 %

`7z` : 23 %

5.3.2 Efficacité temporelle

`Compress` :0s181

`7z` :0s065

`Uncompress` :2s35

`7z` :0s021

5.3.3 Efficacité mémoire

On la détermine en fonction du nombre d'octets utilisé par le programme au cours de sont déroulement.

`Compress` : 76 Mo

`Uncompress` : 163 Mo

`7z` : Non testé

5.4 Evolutions futures

Dans une optique d'amélioration du programme, il serait possible de réaliser:

- Une lecture de la table de correspondance directement depuis la description de l'arbre
- Une gestion améliorée de la mémoire
- L'utilisation des `List` pour les arbres
- Une interface graphique
- Reprendre right_shift_Binary pour le faire de maniere lineaire