Compte rendu du projet
"*Files de priorité, Arbres de Huffman*"

Equipe 82

GOUTTEFARDE Léo
PIERUCCI Dimitri

Jeudi 14 Mai 2015

Introduction

Nous avons traité l'ensemble du sujet, ainsi que l'optimisation de `Decodage_Code` et des fonctions de `Compression` / `Decompression` par l'encodage des arbres de Huffman directement dans le fichier, au lieu d'y laisser la table des fréquences intégrale.

L'exécutable compresse et décompresse sans problème les fichiers texte, et les libérations mémoire sont bien implémentées donc il n'y a pas de memory leak.

L'archive contient l'ensemble du code source au sein du dossier *src*, ainsi que les différents fichiers texte testés dans le dossiers *src/tests*, contenant pour chaque test le fichier *.txt* d'origine, le *.huff* compressé, le *.txt* décompressé, ainsi que l'arbre de Huffman correspondant en *dot* et *png*.

1 Choix d'implémentation

Pour l'implémentation des files de priorité, nous avons introduit quelques nouvelles structures :

```
type Element is record
    P : Priorite;
    D : Donnee;
end record;

type Tableau is array (integer range <>) of Element;

type File_Interne (Taille_Max : Positive) is record
    Tas : Tableau(1 .. Taille_Max);
    Taille : Natural := 0;
end record;
```

Nous avons ainsi simplement défini le type `File_Interne` comme discriminé par la taille maximale de la liste, permettant d'allouer un tableau de taille correspondante pour le `Tas`.

Nous avons également un champ `Taille` qui permet de savoir combien d'éléments sont réellement contenus, et finalement le type `Element` correspond au contenu des cases du `Tas`, et se compose donc d'un champ priorité et d'un champ donnée.

2 Démarche de validation

Pour la démarche de validation fonctionnelle, nous avons d'abord testé nos files de priorité à l'aide du fichier `test_file.adb`.

Puis, pour corriger notre code de génération de l'arbre de Huffman et donc des codes optimaux, nous avons principalement réutilisé la fonction de génération d'arbre *dot* pour facilement visualiser les problèmes de l'arbre *png* généré. Pour ces tests, nous avons repris l'exemple de construction du sujet avec le mot *"hello"*, au sein du fichier de test *src/tests/test0.txt*.

Finalement pour résoudre les problèmes restants après optimisation du code de `Compression` / `Decompression` par encodage de l'arbre de Huffman, nous avons réutilisé le premier exemple du sujet avec la chaîne *"acbeceea"*, à l'aide du fichier de test *src/tests/test1.txt* (Figure 1).

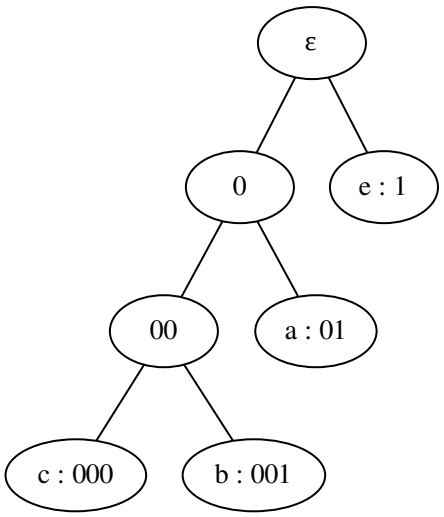

FIGURE 1 – Arbre de Huffman de *"acbeceea"*

Nous avons ainsi facilement pu vérifier la bonne fonctionnalité du code, et finalement corrigé l'erreur en remarquant une simple inversion des variables de `Stream` lors du décodage, qui était à l'origine du problème.

3 Tests de performance

3.1 Gains / pertes d'espace après compression

Pour évaluer les gains et pertes d'espace après compression, nous avons utilisé plusieurs fichiers de test :

1. *src/tests/test0.txt* : mot *"hello"*
2. *src/tests/test1.txt* : chaîne *"acbeceea"*
3. *src/tests/test2.txt* : *Lorem Ipsum* classique de 3 paragraphes
4. *src/tests/test3.txt* : *Lorem Ipsum* aléatoire de 10 paragraphes
5. *src/tests/test4.txt* : *Lorem Ipsum* aléatoire de 20 paragraphes

Ci-dessous les tailles avant / après compression :

Fichier	test0.txt	test1.txt	test2.txt	test3.txt	test4.txt
Non compressé	5 o	8 o	2 Ko	3.4 Ko	7 Ko
Compressé	11 o	11 o	1.1 Ko	1.8 Ko	3.7 Ko

A l'issue de ces tests, on remarque que pour les fichiers relativement petits comme *"hello"* et *"acbeceea"* (deux premiers tests), il y a une légère augmentation de la taille. On aurait pu réduire leur taille de 4 octets en retirant le nombre total de caractères écrit en tête de fichier, cependant nous l'avons gardé car c'est aussi un moyen de simplifier la décompression et de vérifier la taille du contenu décompressé.

En revanche, les *Lorem Ipsum* bénéficient bien d'un gain en espace intéressant, avec une réduction de quasimment 50 %.

Notre programme de compression par encodage de l'arbre de Huffman en tête de fichier est donc assez puissant.

3.2 Vitesse d'exécution

Les vitesses d'exécution du programme mesurées en moyenne pour les différents tests sont les suivantes :

Fichier	test0.txt	test1.txt	test2.txt	test3.txt	test4.txt
Compression	3 ms	3 ms	5 ms	6 ms	9 ms
Décompression	3 ms	3 ms	3 ms	4 ms	6 ms

On observe que pour les deux premiers tests, la compression et la décompression prennent autant de temps, soit trois millisecondes. C'est probablement une approximation de la borne inférieure de la durée minimale du programme, et vu que les deux textes sont courts la compression et la décompression durent quasimment autant de temps.

Pour les Lorem Ipsum de taille croîssante en revanche (trois derniers tests), la vitesse de compression devient un peu plus longue que celle de décompression, jusqu'à 9 ms de compression et 6 ms de décompression pour le Lorem Ipsum de 20 paragraphes contre 5 ms de compression et 3 ms de décompression pour celui de 3 paragraphes.

C'est normal puisque la compression nécessite l'analyse fréquentielle des caractères puis la construction de l'arbre de Huffman, tandis que pour la décompression il suffit d'utiliser l'arbre du fichier compressé.

Ce programme de compression est donc relativement performant d'après les tests effectués.