Compte rendu du projet
"*Files de priorités, 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 *_dec.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`.

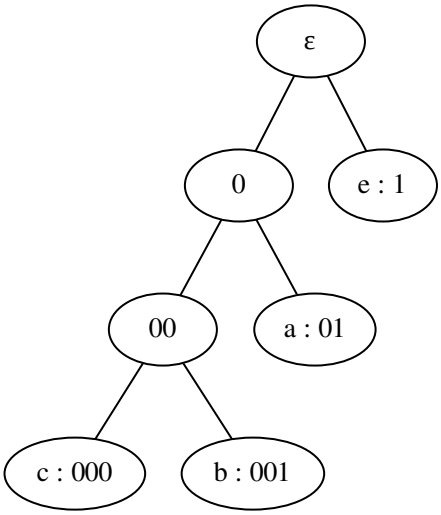
Nous avons également un champ `Taille` qui permet de savoir combien d'éléments sont réellement contenus, et finalement le type `Element` correspond au contenu des cases du `Tas`, et se compose donc d'un champ priorité et d'un champ donnée.

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*. On obtient notamment l'arbre suivant :



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 Vitesse d'exécution

3.2 Gains / pertes de place après compression

Pour évaluer les gains et les pertes de place après compression, nous avons utilisé plusieurs fichier 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

A l'issue de ces tests, on remarque que pour les fichiers relativements petits comme les deux premiers, il y a une légère augmentation de la taille : 5 octets en deviennent 11 pour *"hello"*, tandis que 8 en deviennent également 11 pour *"acbeceea"*.

En revanche, tous les *Lorem Ipsum* bénéficient bien d'un gain intéressant, avec le premier qui passe de 2 Ko à presque 1 Ko, le second de 3.4 Ko à 1.8 Ko, et le dernier de 7 Ko à 3.7 Ko.

Ainsi ces derniers fichiers bénéficient quasimment d'un gain de taille de 50 %, ce qui est relativement intéressant.

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