Rapport du projet de PIM
Codage de Huffman

Martin GUIDEZ et Héloïse LAFARGUE

Département Sciences du Numérique - Première année
2020-2021

Table des matières

I Introduction

L'objectif de ce projet estla réalisation d'un algorithme de compression et décompression des fichiers en utilisant le codage de Huffman. C'est une technique de compression de données sans perte. Cette technique de codage utilise des arbres, dits arbres de Huffman. Dans un arbre on retrouve le nouveau codage du caractère avec la fréquence d'occurences du caractère dans le texte. Ce codage utilise un code à longueur variable pour représenter les caractères, le code est optimal au sens de la plus courte longueur pour un codage par symbole.

II Résumé du projet

Nous cherchons à écrire les algorithmes de compression et de décompression. Il faut pour cela décrire les raffinages des deux programmes ainsi que les principaux types de données et fournir les interfaces des modules et programmes de tests associés.

La méthode de Huffman consiste à remplacer les caractères les plus fréquents par des codes courts et les caractères les moins fréquents par des codes longs. On réalise un parcours infixe de l'arbre.

III Projet

III.1 Architecture de l'application des modules

Pour les modules Compresser et Decompresser, on utilise les Package Arbres.

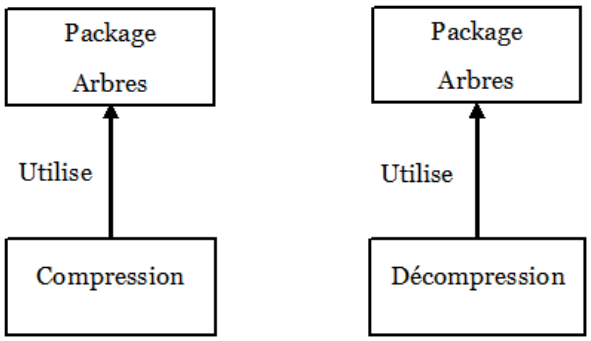
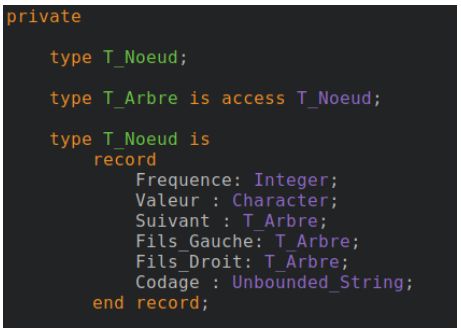


FIGURE 1 – Architecture de l'application des modules

III.2 Présentation des principaux choix réalisés

- On représente l'arbre de Huffman comme un ensemble de noeud comportant un fils droit et un fils gauche, le suivant, la valeur du caractère, sa fréquence d'apparition dans le fichier et son codage (Figure 2).
- Par convention, le fils de gauche d'un noeud est étiqué par 0 et le fils de droite par 1.
- On choisit d'utiliser une structure d'arbre et de liste chaînée.
- Pour représenter le code d'un caractère on utilise un UnboundedString plutôt qu'un type octet car sa taille varie et on doit garder les zéros du début.



```
private

    type T_Noeud;

    type T_Arbre is access T_Noeud;

    type T_Noeud is
        record
            Frequence: Integer;
            Valeur : Character;
            Suivant : T_Arbre;
            Fils_Gauche: T_Arbre;
            Fils_Droit: T_Arbre;
            Codage : Unbounded_String;
        end record;
```

FIGURE 2 – Type abstrait de données, Package Arbres

— Le symbole de fin de traitement est le caractère vide (code ASCII(0)).
— Lors de la décompression, décoder les symboles du texte d'origine consiste à ajouter un carctère du texte jusqu'à tomber sur une suite de bits qui correspond à un code existant dans la table de décodage, ceci jusqu'à la fin du texte.

III.3 Présentation des principaux algorithmes et types de données

Les principaux types de données utilisés sont :
— les arbres " **Arbres**" de type T Arbre,
— les noeuds " **Noeud**" de type T Noeud,
— les fréquences " **Frequence**" de type Integer,
— les valeurs " **Valeur**" de type Character,
— les arbres suivants " **Suivant**" de type T Arbre,
— les fils gauches " **Fils Gauche**" de type T Arbre,
— les fils droits " **Fils Droit**" de type T Arbre,
— les codages " **Codage**" de type Unbounded String,
— l'arbre des codages " **Codages**" de type T Arbre.

Pour le module **Compresser**, qui construit l'arbre, vérifie son intégrité et construit le fichier de l'arbre de Huffman, on utilise :
— La fonction **ConstruireArbre(nomFichier)** qui à partir d'un fichier texte de type String construit la liste chaînée, la tri et construit l'arbre de Huffman associé.
— La procédure **ConstruireFichier(Huff, nomFichier)** qui ajoute le codage, puis l'encodage du parcours infixe et enfin le texte en utilisant les algorithmes Ajout(Arbre), CreationListeInfixe, EcrireOctect(Arbre), EcrireCodage et EcrireUnboundedString.

Pour le module **Decompresser**, qui reconstruit l'arbre, vérifie son intégrité, construit la liste des codages et reconstruit le fichier et achève la décompression, on utilise :
— La fonction **ConstruireArbre(nomFichier)** qui récupère la liste de caractères, puis récupère l'encodage, purge les zéros surnuméraires et enfin ferme le fichier. Pour cela on utilise les algorithmes ReconstArbre(Parcours) et RemplirHuff(Huff, Liste).
— La procédure **ConstruireFichier(nomFichier, Codages)** qui reconstruit le fichier texte à partir de l'arbre des codages. Elle utilise les algorithmes OctectToString et GetCharacter.

Pour le package **Arbres**, on instancie :
— Initialiser(Huff) pour initialiser un arbre vide,
— InitCellule(Cellule) pour initialiser un noeud,
— EstVide(Huff) pour savoir si l'arbre est vide,
— Frequence(Arbre) qui renvoie la racine d'un arbre,

— Valeur(Arbre) renvoie la feuille d'un arbre,
— Suivant(Arbre) renvoie l'arbre suivant,
— Codage(Arbre) renvoie le codage d'un arbre par le type Unbounded String,
— ModifierFrequence(Arbre, Frequence),
— ModifierValeur(Arbre, Valeur),
— ModifierSuivant(Arbre, Suivant),
— ModifierFG(Arbre, Fils),
— FG(Arbre) qui renvoie le fils gauche,
— MofidierFD(Arbre, Fils),
— FD(Arbre) qui renvoie le fils droit,
— DelSuivant(Huff) qui supprime le suivant dans l'arbre,
— IncrFreq(Huff, Caractère) qui incrémente la fréquence du caractère,
— Ajouter(Huff, Arbre) qui ajoute l'arbre à la fin,
— AjouterDeb(Huff, Arbre) qui ajoute l'arbre au début,
— Inserer(Huff, Arbre) qui insère un arbre au bon endroit dans la liste chaînée (triée dans l'ordre croissant des fréquences),
— InsererAIndice(Liste, Arbre), Indice(Liste, Valeur),
— Tri(Huff) qui renvoie l'arbre trié par tri par insertion de la liste chaînée,
— EstFeuille(Arbre) qui permet de vérifier si on a une feuille,
— GenererCodages(Huff) qui renvoie l'arbre contenant les codages,
— Enregistrer (Arb1 ; Arb2) qui crée un nouvel arbe à partir de Arb1 et Arb2,
— CreerFeuille (Frequence ; Valeur) qui crée une feuille,
— Code (Codages ; Caractere) qui renvoie le code en UnboundedString,
— Supprimer(Liste ; Valeur) pour supprimer une valeur d'un arbre,
— PourChaque (Liste) pour appliquer dans tous l'arbre ,
— DupliquerDernier (Liste),
— CreerCodageInfixe (Huff)qui renvoie le codage en UnboundedString,
— AfficherListe (Liste) pour afficher la liste,
— AfficherCodages (Codages),
— AfficherArbre(Huff) qui affiche la version textuelle de l'arbre de Huffman.

III.4 Démarche adoptée pour tester le programme

On teste le programme sur des fichiers contenant un seul caractère, contenant la touche retour "entrée" et des fichiers contenant une ligne, un paragraphe et un texte.
On peut compresser et décompresser deux fichiers en même temps.
On affiche bien la version textuelle de l'arbre de Huffman et la version textuelle de la table de Huffman (Figure 4).
Pour un fichier de 2000 lignes, la compression prend moins de 25s.
Pour un fichier vide, on a un CONSTAINT ERROR levé (Figure 3).

```
hlafargu@n7-ens-lnx022:~/Annee_1/s5/PIM/projet/CD02/src$ ./compression test.txt


### Ecriture de test.txt ###
--/ Construction de l'arbre /--
--------- Liste tri◆e ---------
F: /$ - Etage:           0
--------
--------- Construction arbre ---------
-- Huff final --
F: /$ - Etage:           0
int◆grit◆ :
TRUE--/ Construction du fichier /--
Indice du char de fin dans la liste :  0


raised CONSTRAINT_ERROR : arbres.adb:304 access check failed
```

FIGURE 3 – Test pour un fichier vide

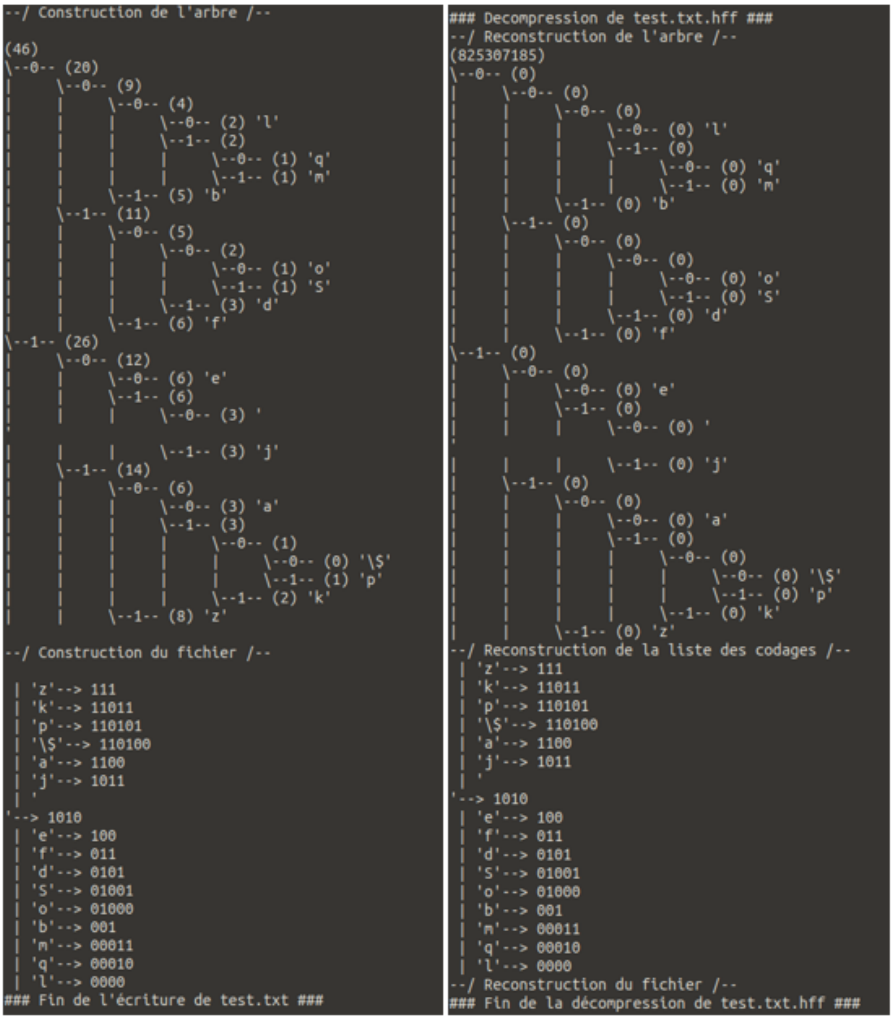


FIGURE 4 – Test Compression et Décompression

III.5 Difficultés rencontrées et solutions adoptées

Difficulté - Choix de structure : Notre première idée était de créer une liste d'arbres réduits puis de combiner les arbres deux à deux. Cette structure ne permet pas d'obtenir un nouvel arbre.

Solution : On ne fait plus une liste d'arbre mais on utilise Suivant pour construire l'arbre. Nous avons dû refaire les raffinages.

Difficulté - Touche retour : Le programme ne prend pas en compte les retours à la ligne.

Solution : On choisi une lecture d'octets au lieu de texte.

Difficulté - Traitement de gros fichiers : L'éxécution ne finit pas pour les très gros fichiers.

Solution : Il faut écrire les octets au fur et à mesure au lieu de construire le code complet et de l'écrire ensuite.

III.6 Organisation de l'équipe

Martin
— Raffinages
— Programme compression
— Programme arbres.adb
— Programmes de tests arbres.ads

Héloïse
— Raffinages
— Programme affichage arbre dans arbres.adb
— Programme décompression
— Rapport

IV Conclusion

IV.1 Bilan technique

Nos programmes compressent et décompressent des fichiers texte. On pourrait optimiser le code pour que la compression et la décompression prennent moins de temps.

IV.2 Bilan personnel

Nous avons passé environ 40h sur les programmes (conception 3h, implantation 25h, mise au point 10h) et 7h sur le rapport.

Martin :
Ce projet m'a permis tout d'abord de découvrir un algorithme de compression mais aussi de mieux comprendre la manipulation des octets. Malgré certaines difficultés d'ordre organisationnel, le programme final est plutôt complet mais il pourrait encore être amélioré notamment en ce qui concerne la gestion des erreurs du côté de l'utilisateur, et la rapidité de la compression. Au final, j'ai découvert à travers ce projet les différentes étapes de la conception d'un programme et la collaboration, choses essentielles au bon déroulement d'un projet.

Héloïse :
Le projet m'a permis de mettre en pratique les connaissances reçues en cours. J'ai pu mieux appréhender la conception de programme et de tests du début à la fin. Cela m'a aussi permis de découvrir un premier algorithme de compression.