Programmation Impérative :
Rapport Codage de Huffman

Léo Meissner et Steven Zheng

Département Sciences du Numérique - Première année
2021-2022

Table des Matières

1. Résumé

Dans le cadre de notre premier projet en programmation impérative, nous devions refaire la compression et la décompression de texte selon la méthode de Huffman.

Pour réaliser ce projet, nous avons d'abord réparti les différentes tâches à faire entre les deux binômes puis raffiné les deux programmes séparément ce qui nous a permis de déterminer précisément les types de données dont nous avons eu besoin. Nous avons alors construit les modules et les programmes de tests associés à ces types de données pour enfin implémenter les deux programmes en ADA. Puis nous avons fait différents tests pour vérifier notre programme.

2. Introduction

Le codage de Huffman est un codage statistique utilisé pour la compression sans perte de données telles que les tel que les textes, les images ou les sons. L'objectif de ce projet est d'écrire deux programmes, le premier compressant des fichiers en utilisant le codage de Huffman et le second qui les décompresse.

La compression se fait à l'aide d'un arbre de Huffman. Ce dernier est construit à partir de la fréquence d'apparition de chaque caractère. L'enjeu de la compression et de la décompression est donc de reconstruire l'arbre de Huffman et la table de Huffman pour pouvoir ainsi compresser ou décompresser le texte.

La spécificité de ce codage est que la longueur d'un caractère codée est lié à sa fréquence d'apparition dans le texte. En effet plus le caractère sera utilisé plus son codage sera court permettant ainsi une meilleure compression. Et inversement, plus il sera peu utilisé plus il sera long pouvant même dépasser les 8 bits originalement codés.

3. Organisation de l’équipe

Steven s’est chargé du raffinage et de la mise en œuvre de l’algorithme de la compression et Léo du raffinage et de la mise en œuvre de l’algorithme de la décompression. Mais dans un souci de compréhension globale du sujet, une partie du raffinage de la compression et de la décompression s’est faite ensemble.

4. Conception

4.1 Modules utilisés

Lors du raffinage et dans le cadre limité aux concepts vus en cours, TD et TP, nous avons utilisé différentes listes chaînées associatives.
Donc le module choisi pour cela était le même utilisé lors du TP10 et un module d'arbre binaire.

Voici l'architecture de l'application en modules que nous avons choisi pour le programme compresser (figure-1) puis celle que nous avons choisi pour le programme décompresser (figure-2).

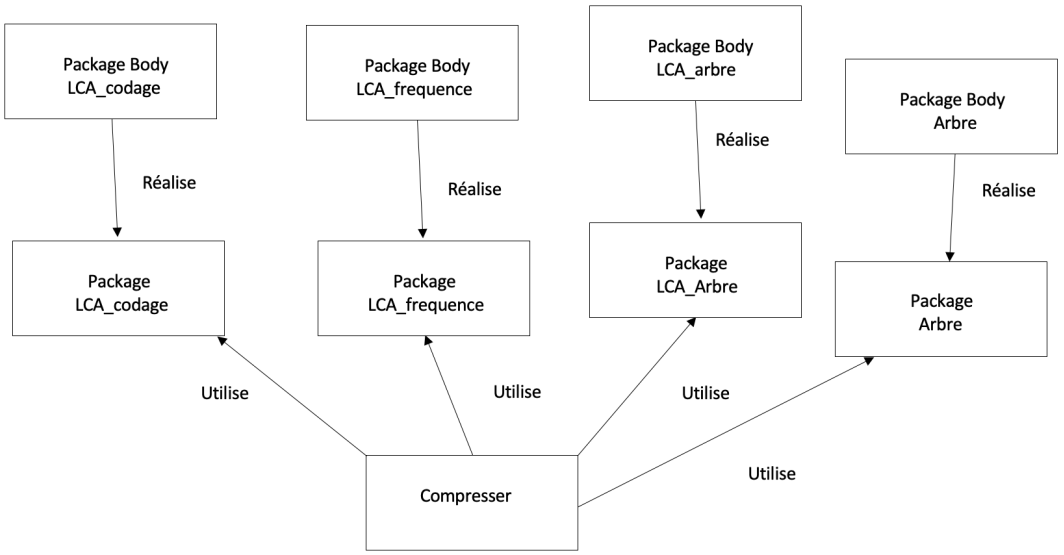

figure-1

Plus précisément, la table de fréquence est une LCA_frequence dont la clé est un unbounded string de 8 bits et sa donnée est sa fréquence. La table de Huffman est une LCA_codage dont la clé est le codage et sa valeur les 8 bits. Pour LCA_arbre, qui est utilisé pour créer l'arbre de Huffman, a pour clé les 8 bits et comme donnée un arbre binaire. Pour l'arbre binaire, chaque nœud possède deux éléments: les 8 bits et la fréquence.

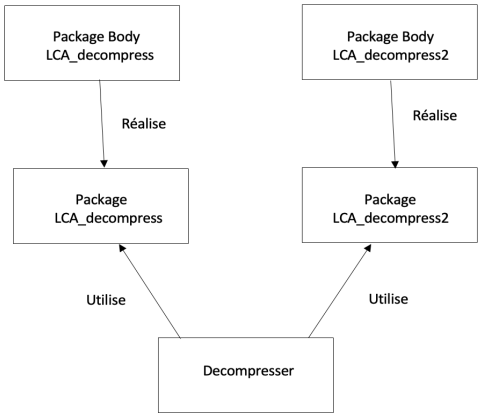

figure-2

Plus précisément le tableau de caractères est une LCA_decompress avec en T_clé un entier et en T_Donnee un Unbounded_String qui est le symbole du texte d'origine. Ensuite, la table de codage de Huffman est une LCA_decompress2 avec en T_cle un Unbounded_String qui est le code du symbole et en T_Donnee, un T_Octet qui est le symbole en question.

Types de données utilisées :

Les principaux types de données que nous avons utilisés sont les suivants :

- Listes chaînées associatives (LCA)
- Arbres Binaires
- Unbounded_String
- String
- T_Octet

4.2 La présentation des principaux algorithmes et types de données

Créer la table de fréquence:

Pour créer la table de fréquence, il fallait d'abord lire l'octet du fichier, le convertir en bits. Ensuite, il fallait vérifier si cette chaîne de bits était dans la table de fréquence ou non: si oui on augmentait la fréquence de 1 sinon on enregistrait dans la table.

Créer l'arbre de Huffman:

Avant de créer l'arbre de Huffman, nous avons créé une liste d'arbre élémentaire pour chaque caractère et sa fréquence. Puis nous avons fait comme dans le sujet en prenant les deux fréquences minimales puis fusionner ses deux arbres jusqu'à avoir plus qu'un seul arbre dans la liste.

Créer la table de Huffman:

Pour créer la table de Huffman, nous avons créé une procédure qui parcourait l'arbre de manière infixe et de façon récursive. La procédure Parcours_infixe prenait comme paramètre l'arbre ou le sous arbre, le code et aussi le parcours infixe que l'on complétait de proche en proche. La procédure s'appelait deux fois une fois sur le sous arbre gauche en ajoutant un 0 au code et au parcours infixe et une fois sur le sous arbre droit en ajoutant cette fois-ci un 1.
Lorsqu'on arrivait à une feuille, on enregistrait les 8 bits de la feuille et le code dans la table de Huffman puis on retirait le dernier bits du code.

Afficher l'arbre de Huffman:

Pour représenter l'arbre de Huffman dans le mode bavard, nous avons, comment pour créer la table de Huffman, parcouru l'arbre de manière infixe et de façon récursive. La procédure afficher_abr nécessite l'arbre ou le sous arbre et son chemin. Et donc grâce à ce chemin, on peut connaître la profondeur du sous arbre et aussi s'il fallait afficher "|" (si c'est 0) ou bien juste un espace " " (si c'est 1).

Problème principale de la décompression : reconstruire l'arbre de Huffman :

Le problème principal rencontré lors de la conception du programme décompresser a été de reconstruire la table de codage de Huffman.

Une fois le parcours infixe récupéré, deux options nous observions 2 moyens de récupérer les caractères codées. La première était de reconstruire l'arbre de Huffman avec le parcours infixe en utilisant une Liste Chainé associative de type T_Arbre comme lors de la compression. Une fois l'arbre reconstitué, pour reconnaître un caractère, il suffit de partir de la base de l'arbre, d'aller à gauche si le caractère lu dans le texte est un 0 ou à droite si celui-ci est un 1 et lorsque l'on arrive sur une feuille, il faut ajouter le caractère décodé dans le texte décompressé. Mais après plusieurs heures de recherches, nous n'avons pas trouvé d'algorithme permettant de reconstruire l'arbre de Huffman à partir des caractères et du parcours infixe.

La seconde option, trouvée ultérieurement, est de reconstruire directement la table de codage de Huffman. Pour cela il faut parcourir le parcours infixe de l'arbre en utilisant l'algorithme suivant : Initialiser un code vide, lorsque le caractère lu est un zéro ajouter un zéro, lorsque le caractère lu est un 1 : ajouter le code et le premier caractère dans la table de codage puis supprimer le dernier 0 du code et encore supprimer chaque 1 qui se trouve après ce 0 puis changer le dernier élément en 1 et lire l'élément suivant. Une fois le parcours infixe entièrement parcouru, la table de codage est reconstituée.

Décoder les symboles du texte d'origine :

Maintenant, pour décoder les symboles du texte d'origine il faut initialiser un code vide et ajouter chaque caractère du texte compressé jusqu'à ce qu'il y ait un des codes de la table de codage qui lui correspondent. Une fois ce code trouvé, nous ajoutons le symbole au texte décompressé et réinitialisons le code pour réitérer l'opération jusqu'à l'apparition du caractère spéciale.

5. Tests

5.1 Tests effectués

Nous avons commencé par vérifier nos modules avec les programmes test_lca_projet.adb (programme de test similaire a celui du TP10) et test_arbre_bin qui permet de vérifier le module Arbre_binaire.

Ensuite, une fois les programmes compresser et décompresser réalisés nous avons compressé puis décompressé des fichiers pour voir comment nos programmes fonctionnaient. Une exception est levée lorsque le fichier à décompresser n'a pas l'extension .hff et l'option bavard a été également vérifiée.

```
'01101010' --> 000
'01101111' --> 001
'00100000' --> 010
'01110000' --> 011
'01100101' --> 100
'01110010' --> 101
'01110101' --> 1100
'11111111' --> 11010
'01110011' --> 11011
'00001010' --> 1110
'01110100' --> 1111
(13)
\--0--(5)
|      \--0--(2)
|      |      \--0--(1) '01101010'
|      |      \--1--(1) '01101111'
|      \--1--(3)
|             \--0--(1) '00100000'
|             \--1--(2) '01110000'
\--1--(8)
       \--0--(4)
       |      \--0--(2) '01100101'
       |      \--1--(2) '01110010'
       \--1--(4)
              \--0--(2)
              |      \--0--(1) '01110101'
              |      \--1--(1)
              |             \--0--(0) '11111111'
              |             \--1--(1) '01110011'
              \--1--(2)
                     \--0--(1) '00001010'
                     \--1--(1) '01110100'
```

```
lmeissne@n7-ens-lnx045:~/Annee_1/PIM/Projet_Hff/MN07/src$ ./decompresser test.txt
Le fichier n'a pas l'extension .hff, decompression impossible
lmeissne@n7-ens-lnx045:~/Annee_1/PIM/Projet_Hff/MN07/src$
```

6. Bilan

6.1 Bilan Technique

Les programmes, compresser et décompresser fonctionnent et les exigences du projet ont été remplies au moment du rendu de ce projet. Mais nous avons des perspectives d'amélioration pour nos programmes.

Dans l'option bavard de la compression, nous pourrions notamment ajouter le taux de compression du texte donné en entrée. Avoir une compression et une décompression plus rapide.

6.2 Bilan Personnel

6.2.1 Bilan Personnel : Léo Meissner

La réalisation de la compression et de la décompression d'un fichier texte via le codage de Huffman m'a beaucoup intéressé. J'ai apprécié de travailler sur ce projet qui m'a notamment appris à coder à plusieurs ainsi qu'à séparer les différentes tâches. Réaliser ce projet en groupe est un réel avantage car lorsque l'un des membres du groupe est bloqué, son binôme peut venir l'aider et il est plus facile de débuguer un programme à 2.

Avec Steven, nous avons passé environ 7 heures sur le raffinage afin que celui-ci soit le plus précis possible ce qui nous a permis de gagner du temps à l'implantation des programmes. J'ai ensuite passé plus de 11 heures sur l'implantation de décompresser, puis 5 heures sur la mise au point, 2 heures sur le rapport et 1 heure sur le manuel d'utilisation.

Ce projet m'a permis de mieux comprendre les actions liées à la bibliothèque command_line ainsi que la manière de lire des octets dans fichier.

6.2.2 Bilan Personnel : Steven Zheng

Ce projet a été très enrichissant et m'a permis de mettre en pratique les notions vues en cours, en TP et en TD. Cela m'a permis aussi d'identifier et de résoudre plus rapidement les problèmes que j'ai pu avoir dans mon code, j'ai senti vers la fin que j'étais bien plus à l'aise. Le fait d'être en binôme a été très utile lorsque l'un de nous deux était bloqué ou avait besoin d'un avis.

J'ai passé environ 18 heures sur l'implantation de décompresser et la mise au point et 3 heures sur le rapport.