Compression de fichier texte avec codage de Huffman sous Ada

MOUTAHIR Jed
AIMI Mathis

Département Sciences du Numérique - Première année
2021-2022

Table des matières

1 Résumé

Le codage de Huffman (1952) est un codage statistique utilisé pour la compression sans perte de données telles que les textes, les images (fichiers JPEG) ou les sons (fichiers MP3). Dans le cas de textes, son principe est de définir un nouveau codage des caractères, codage à taille variable qui tient compte de la fréquence (le nombre d'occurrences) des caractères dans le texte : les caractères dont la fréquence est élevée seront codés sur moins de bits et ceux dont la fréquence est faible sur plus de bits.
L'objectif de ce projet est d'écrire deux programmes, le premier qui compresse des fichiers en utilisant le codage de Huffman et le second qui les décompresse.

2 Introduction

Le langage de programmation qui sera utilisé est Ada. Les exécutables produits permettrons à l'utilisateur de compresser et décompresser un fichier texte à travers une ligne de commande. Il aura également la possibilité d'observer l'arbre de Huffman produit lors de la compression.

3 Architecture de l'application

Le module Huffman contient toute la logique de compression/décompression. C'est donc le seul module du projet.
Contenu :
- Types utilisés
- Procédures et fonctions nécessaire à la compression/décompression
- Procédures et fonctions nécessaire à l'affichage

3.1 Choix, Algorithmes Principaux et Types Principaux

Pour réaliser ce module, plusieurs chois ont été faits :

3.1.1 Type pour la représentation de l'arbre de Huffman

$Huffman\_Tree$ est le type qui stocke l'arbre de Huffman. C'est un enregistrement de noeuds ($Node\_Access$) et d'une carte ($Encoding\_Maps.Map$).

-$Node\_Access$ est un pointeur sur $Huffman\_Node$.

-$Huffman\_Node$ est un enregistrement qui contient les données d'un noeud : fréquence, enfant gauche, enfant droit, caractère.

-$Encoding\_Maps.Map$ est une carte définie à l'aide du module $Ada.Containers.Indefinite\_Ordered\_Maps$ avec pour type d'éléments un sequence de bits ($Bit\_Sequence$) et un caractère comme clé.

-$Bit\_Sequence$ est une liste de booléens.

Ce type est complexe mais permet de définir les procédures de manipulation des données de façon plus claires est précises.

3.1.2 Type pour la construction de l'arbre de Huffman

-$Frequency\_Maps$ est le type qui stocke les fréquences associées à chaque caractères de manière ordonnée. C'est une carte définie à l'aide du module $Ada.Containers.Ordered\_Maps$ avec

pour type d'éléments un entier et un caractère comme clé.

-$Node\_Vectors$ est le type qui permet de manipuler les noeuds de l'arbre. C'est un vecteur défini à l'aide du module $Ada.Containers.Vectors$ avec pour type d'éléments un pointeur sur noeud ($Node\_Access$) et un entier positif ($Positive$) comme type d'indice.

3.1.3 Construire l'arbre

$Create\_Tree$ est la procédure permettant de construire l'arbre de Huffman.
Elle prend en entrée :
- un arbre d'Huffman vide qui sera remplit ($Tree$: $out\ Huffman\_Tree$)
- une carte de fréquence ($Frequencies$: $Frequency\_Maps.Map$)

3.1.4 Encoder le fichier

$Encode$ est la fonction permettant d'encoder le fichier.
Elle prend en entrée :
- un arbre d'Huffman ($Tree$: $Huffman\_Tree$)
- une suite de caractères ($Symbols$: $Symbol\_Sequence$)
Elle donne en sortie :
- une suite de bits ($Bit\_Sequence$)

3.1.5 Décoder le fichier

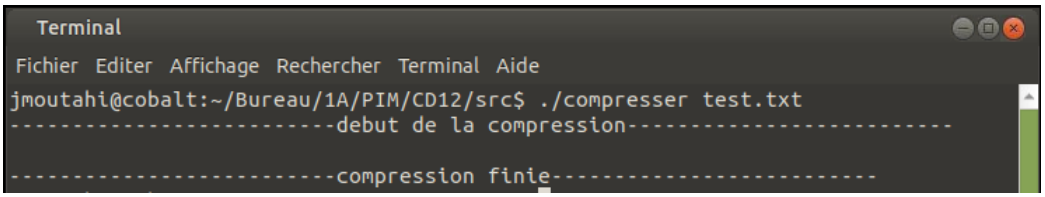
$Decode$ est la fonction permettant de décoder le fichier.
Elle prend en entrée :
- un arbre d'Huffman ($Tree$: $Huffman\_Tree$)
- une suite de bits ($Code$: $Bit\_Sequence$)
Elle donne en sortie :
- une suite de caractères ($Symbol\_Sequence$)

4 Test du module

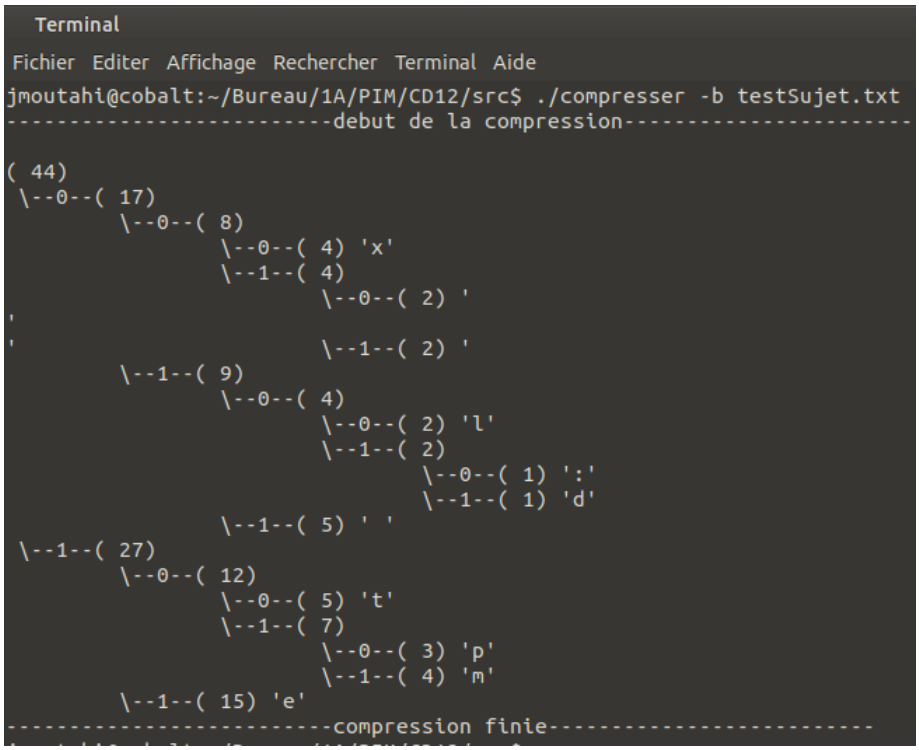
Le module a été testé de manière à vérifier qu'il convient au cahier des charges. De même, on vérifie sa rapidité et la limite de taille qu'il peut supporter.

4.1 Cahier des charges

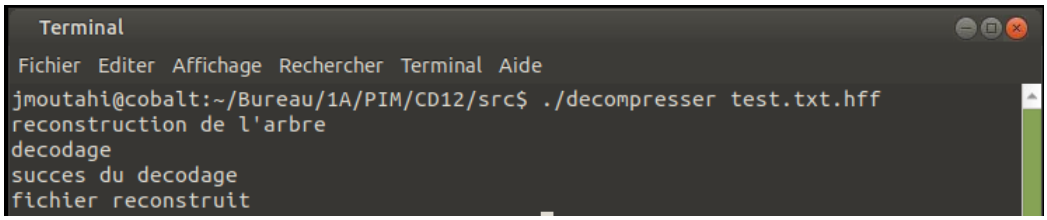
Le module devait permettre à l'utilisateur de compresser un fichier texte à l'aide d'une commande : *./compresser exemple.txt*
Voici ce qui est observé sur l'invité de commande :



Le module devait également permettre à l'utilisateur de visionner l'arbre de Huffman associé au fichier avec : *./compresser −b exemple.txt*
Voici ce qui est observé sur l'invité de commande :



Enfin, le module devait permettre à l'utilisateur de décompresser un fichier avec : *./decompresser exemple.txt.hff*
Voici ce qui est observé sur l'invité de commande :



On vérifie bien que le fichier décompressé est identique au fichier original sur plusieurs fichiers.

4.2 Rapidité

Le temps de compression a été mesuré sur plusieurs fichiers avec la commande : *time ./compresser exemple.txt*
- 50 octets $\Longrightarrow$ 0.007s
- 2.5 ko $\Longrightarrow$ 0.014s
- 10 ko $\Longrightarrow$ 0.095s
- 26 ko $\Longrightarrow$ 0.668s

Voici le temps pour la decompression avec la commande : *time ./decompresser exemple.txt.hff*
- 50 octets $\Longrightarrow$ 0.005s
- 2.5 ko $\Longrightarrow$ 0.009s
- 10 ko $\Longrightarrow$ 0.011s
- 26 ko $\Longrightarrow$ 0.017s

4.3 Limites

Les procédures et fonctions étant majoritairement récursives, à partir d'une certaine taille, on a surcharge du système (stack overflow). De ce fait, le module est limité à une taille maximale de fichier texte : 50 ko.

5 Difficultés Rencontrées

Le projet étant complexe, plusieurs problèmes sont survenu lors de sa mise en oeuvre.

5.1 Lecture/Écriture de fichier

L'écriture du fichier compressé a soulevé plusieurs problèmes. En effet, au début, nous avons pensé qu'écrire le fichier encodé était fait avec : $Bit\_Sequence'Write(S, Code)$ pour chaque caractère encodé. Cependant, cette méthode résulte en une absence de compression car elle ne fait que remplacer un octet (code ASCII d'un caractère) en un nouvel octet (nouveau code associé au caractère). Il n'y a donc aucun changement de taille.
Pour résoudre ce problème, nous avons concaténé toute les chaine de bits associées au nouveau fichier dans une grande chaine de bits. Ensuite, il suffit de découper cette chaine en Octets (donc 8 élément) et de les écrire les uns après les autre avec : $T\_Octet'Write(S, Octet)$. Il a aussi fallu faire attention au fait que la chaine n'étant pas forcement un multiple de 8, des bit du dernier octet ne font pas parti du fichier encodé.

5.2 Récupération de l'arbre de Huffman

Pour reconstruire le fichier texte lors de la décompression, il faut récupérer les données permettant de reconstruire l'arbre de Huffman. Ceci de manière à pouvoir décoder le fichier.
Cette difficulté a été surmonté grâce à la définition non ambiguë des types du module. En effet pour cette partie, nous avons décidé de ne pas faire comme le sujet le propose. À la place de construire une entête compliquée, il suffit de mettre la carte des fréquences : $Frequency\_Maps.Map'Write(S, Frequencies)$.
La procédure $Create\_Tree$, a seulement besoin de cette carte pour reconstruire l'arbre.

6 Répartition

Les module ont été réfléchit à deux mais la répartition plus précise est :
$Huffman.adb$ a principalement été codé par Jed MOUTAHIR.
Mathis AIMI s'est chargé de faire le raffinage, les tests ainsi que *decompression.adb*.
compression.adb a été codé par Jed MOUTAHIR.

7 Conclusion

Le module est fonctionnel et suit le cahier des charges.
Cependant, plusieurs améliorations peuvent être envisagées :
- Faire un module plus générique pouvant compresser d'autre type de fichier (mp4, mp3, png, jpg, ...)
- Modifier les fonction/procédure récursives de manière à ne pas surcharger le système et accepter des fichier plus lourds.

8 Bilan personnel

Le langage de programmation imposé étant Ada, l'implantation a pris la majorité du temps passé sur le projet. Ceci a également entrainé un retard lorsque l'échange de code entre nous était trop long.