Compression et décompression à l'aide du codage de Huffman

PROJET PIM 2021-2022

Nicolas CATONI
Alice DEVILDER
Groupe MN03

Résumé

Le codage de Huffman (1952) repose sur le codage à taille variable des caractères dans un texte en tenant compte de leur fréquence d'apparition. Il permet ainsi de définir un nouveau codage optimale dont les caractères de fréquences élevées sont codés sur moins de bits que ceux de fréquences faibles. C'est pourquoi ce codage statistique est utilisé pour la compression de fichier sans perte de données.

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.

Dans ce rapport, nous présentons donc les méthodes de réalisation de ce projet, les décisions prises quant aux divers modules, sous-programmes, types de données utilisés et tests réalisés ainsi que les difficultés que l'on a pu rencontrer et ce dont on a appris.

Cahier des charges

- Pouvoir compresser un fichier de taille indéterminé et le décompresser

- Mettre en oeuvre une interface utilisateur conviviale et afficher les informations liées à la compression/décompression si l'option « -b » ou --bavard » est appelée.

Introduction

Le code de Huffman utilise la fréquence d'apparition (ou nombre d'occurrence) des caractères dans le fichier donc afin de réaliser l'algorithme de Huffman, il faut tout d'abord construire un tableau contenant les fréquences de chaque caractère.

Ensuite, l'algorithme de Huffman consiste à créer un arbre binaire parfait ayant pour feuilles les caractères du texte et pour valeur leur fréquence d'apparition. Les nœuds, ayant toujours deux fils, permettent d'organiser les feuilles de manière à ce que la feuille dont le caractère a une fréquence élevée soit proche de la racine. À chaque nœud est associé une valeur qui est la somme de la valeur de son fils gauche et de la valeur de son fils droit. Par ailleurs, le fils gauche est celui dont la fréquence est la plus faible. Ainsi, la valeur de la racine correspond au nombre de caractères du texte. De plus, chaque branche de l'arbre est étiquetée par la valeur 0 pour le sous-arbre gauche et la valeur 1 pour le sous-arbre droit.

Le code binaire de chaque caractère est alors obtenu en parcourant l'arbre de la racine jusqu'à la feuille (suivant un parcours infixe[1]) et en notant le parcours (0 ou 1) à chaque noeud.

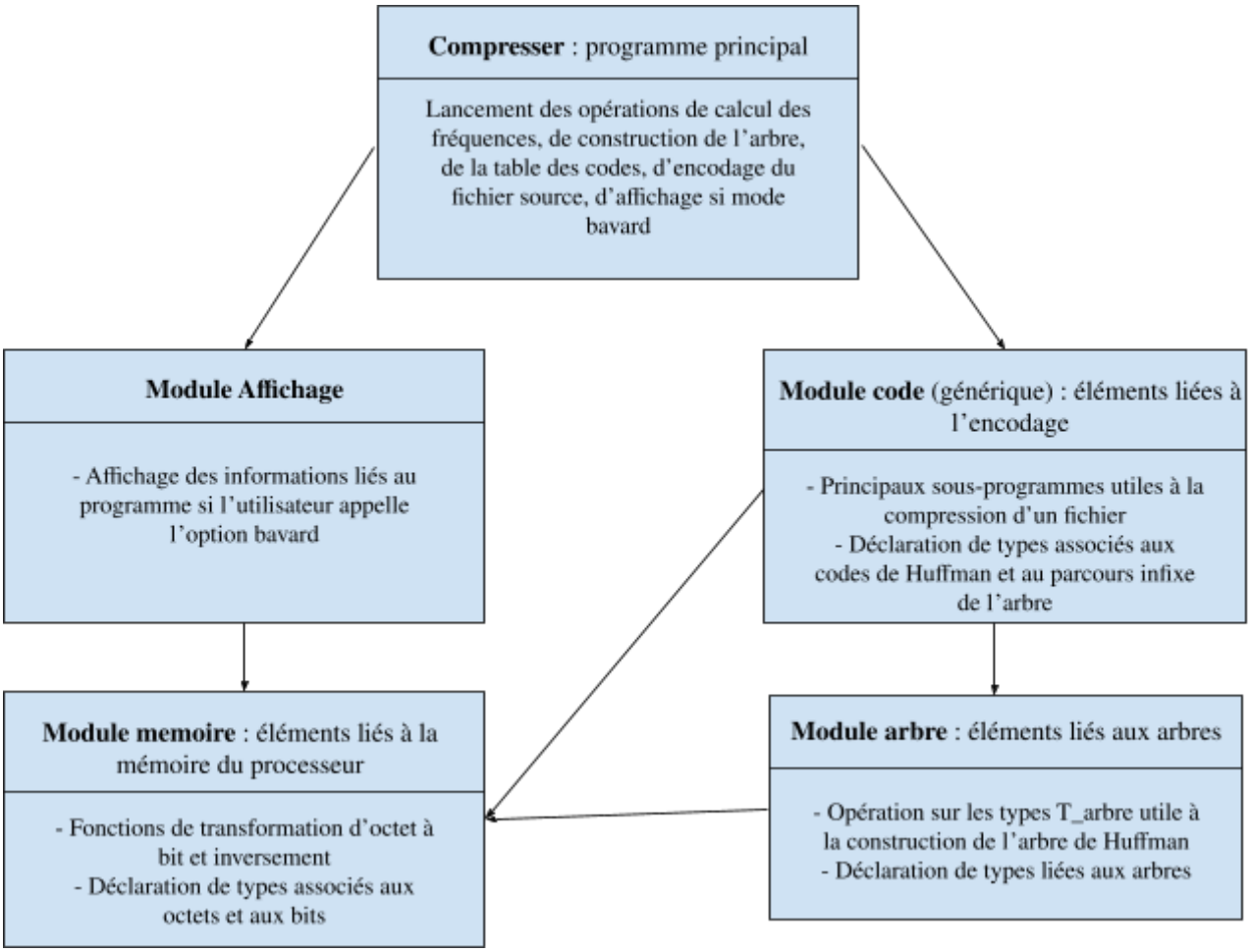
Une fois avoir construit l'arbre de Huffman, la décompression résulte à parcourir l'arbre jusqu'à obtenir un code correspondant à un caractère et à répéter cette opération jusqu'à la fin du texte encodé.

[1] Un parcours infixe consiste à parcourir le sous-arbre gauche, traiter le nœud, puis parcourir le sous-arbre droit.

Plan

I. Conception et codage

I.1 Architecture de l'application en modules



Architecture de l'application en modules pour le programme compresser

Afin de pouvoir utiliser des éléments de programme dans d'autres programmes, nous avons opté pour une architecture de l'application à l'aide de modules.

Un module regroupe des abstractions de données et des abstractions de contrôle sous forme d'unités d'encapsulation. Ainsi l'application (programme compresser et decompresser) sera vu comme un ensemble de modules qui utilisent les uns et les autres.

Au début du projet, nous avions créé un module *types* contenant les types que nous voulions utiliser dans les différents modules. Cependant ce module s'est avéré non essentiel car nous avons finalement défini les types associés à leur utilisation dans leur module attitré.

Nous avons donc implémenté un module *memoire* dans lequel sont définis les types T_byte, T_bit et T_bits (tableau de taille 8 de T_bit) que l'on utilisera tout au long de l'application. (Cf 1.3)

Ensuite, afin de lire les fichiers mis en argument de compression et réaliser différentes actions sur les fichiers, nous avons réalisé un module *fichier* dans lequel se trouve les sous-programmes Creer, Taille, Lire_bit, Lire_byte, Ecrire et Ajouter ainsi que les types T_tab_bit et T_tab_byte qui sont un tableau de taille size*8 de T_bit et un tableau de taille size de T_byte, pour lequel size est un élément générique. Ce module est donc générique, c'est pourquoi il faut l'instancier dans les autres modules lorsqu'il est appelé.

Le module *arbre* regroupe les sous-programmes associés aux types T_arbre (cf I.3). Ce type étant limited private, ces sous-programmes nous permettent d'accéder aux éléments de T_arbre et de réaliser des opérations sur ce type. Nous réutilisons ce module dans la plupart des autres modules (code, decode, compresser, decompresser, affichage).

Afin de rendre réutilisable nos sous-programmes associés à la compression d'un fichier, nous avons implémenté un module *code* comprenant les principaux éléments liés à la construction de l'arbre et de la table de codage de Huffman mais également les procédures de réécriture du fichier à l'aide du codage de Huffman.
De même, le module *decode* contient les éléments nécessaires à la décompression : le programme va appeler des sous-programmes de *decode* afin de reconstruire l'arbre de Huffman suivant un parcours infixe et pouvoir décoder le fichier compressé.

Enfin, nous avons créé un module *Affichage* pour l'affichage de l'arbre et l'affichage de la table de Huffman. Les sous-programmes présents dans ce module vont donc être appelés dans le programme principal de compression et celui de décompression.

I.2 Principaux choix réalisés pour les algorithmes

Nous avons choisi de créer un module *Arbre* qui sera utilisé pour la compression et la décompression en ce qui concerne les opérations sur les structures T_arbre.

Pour gérer la construction de l'arbre, on utilise une liste d'arbres de taille 256. La construction de l'arbre se fait en respectant plusieurs conditions notamment que la fréquence du fils gauche d'un arbre est plus faible que celle du fils droit. De plus, la fréquence d'un nœud correspond à la somme des fréquences de ses fils droit et gauche. Ainsi, nous avons décidé de chercher les 2 arbres de la liste avec les plus faibles fréquences. Puis l'arbre de Huffman est construit itérativement.

Pour stocker le code de huffman de chaque symbole, on utilise un tableau d'enregistrement d'une valeur (tableau de 256 bits) et d'une longueur qui nous permet d'obtenir le code de Huffman. Par exemple, si la longueur du ième tableau vaut n, alors les n premiers bits de la valeur correspondent au code de Huffman du ième caractère de la table ASCII.

Pour vérifier si un caractère est une fin de fichier, nous avons introduit une variable "ff" (Booléen) dans le type T_noeud qui est vrai quand c'est une fin de fichier et faux

sinon.

Pour décompresser le fichier on commence par identifier les segments du fichier compresser en déterminant la taille du tableau des bytes dans l'ordre du parcours infixe puis la taille du parcours infixe. On lit ensuite ces deux premiers segments ce qui nous permet de reconstruire l'arbre de huffman. On lit ensuite le segment message du fichier compressé. On parcourt l'arbre en fonction des bits lus et on écrit dans le fichier de sortie lorsque l'on arrive à une feuille. On s'arrête au caractère de fin.

I.3 Principaux algorithmes et types de données

Principaux types de données :

— Mémoire —

```
Type T_bit is mod 2;
for T_bit'Size use 1;

Type T_byte is mod 2 ** 8;
for T_byte'Size use 8;

Type T_bits is array (1..8) of T_bit;
```

— Code —

```
Type T_trad is record
        length : integer;
        value : T_byte;
end record;

Type T_tab_code is array (0..256) of T_trad;

Type T_tab_branche is array (0..256) of T_arbre;

Type T_pi is array (1..(nbu_byte) * 2) of T_bit;
Type T_bpi is array (0..nbu_byte) of T_byte;

Type T_code is array (1..256) of T_bit;
```

— Arbre—

```
Type T_noeud;

Type T_arbre is access T_noeud;

Type T_noeud is record
        freq : integer;
        byte : T_byte;
        branche_d : T_arbre;
        branche_g : T_arbre;
        ff : Boolean;
end record;
```

— Decode —

```
Type T_parcours_infixe is array (1..
lng_parcours_infixe) of T_bit;

Type T_tab_feuilles is array (1..lng_
tab_feuilles) of T_byte;
```

Principaux algorithmes :

COMPRESSION

Interpréter la ligne de commande.
Construction du tableau de fréquence.
Construction de l'arbre de Huffman.
Construction du tableau de codage de Arbre.
Traduction du texte (à l'aide du tableau de codage) dans le fichier compressé.
Affichage de l'arbre de Huffman si l'option bavard est appelé.

DECOMPRESSION

Obtention du nombre de caractères uniques, de la longueur du parcours infixe et l'octet de fin de fichier du fichier encodé
Construction du tableau de chaque caractère présent dans le fichier (tab_feuille) et du parcours infixe
Reconstruction de l'arbre de Huffman
Décodage du fichier compressé

II. Réalisation et codage

II.1 Présentation des algorithmes

COMPRESSION

Pour la compression, nous avons utilisé principalement les algorithmes du module code, qui lui-même appelle les sous-programmes du module arbre notamment pour la construction de l'arbre de Huffman, les types associés aux arbres étant privés.

Interpréter la ligne de commande :

Afin de récupérer le nom du fichier à compresser (ou décompresser) et savoir si l'utilisateur veut obtenir des informations par rapport à l'exécution du programme (option bavard), nous avons créé une procédure "gestion_ligne_commande" dans le programme principal "compresser" qui compte le nombre d'arguments de "compresser" entré par l'utilisateur et lui indique s'il y a une erreur ou récupère le nom du fichier si celui-ci existe. Si l'option bavard est appelée, la procédure affecte True à la variable Bavard sinon celle-ci reste à False. Cette variable sera ensuite utilisée dans le sous-programme "Affichage".

Construction du tableau de fréquence :

Le tableau de fréquence correspond en fait à la première branche de notre arbre de Huffman. Ainsi nous avons appelé “Premiere_branche” la procédure qui modifie un tableau de taille 256 de T_arbre. Ce programme va lire le fichier et va incrémenter de 1 la fréquence de l’arbre de tab_branche d’indice correspondant à l’octet lu. Ensuite, pour les octets non présents dans le fichier (leur fréquence est nulle), il va affecter la taille du fichier à leur fréquence pour pouvoir par la suite reconnaître l’octet de fin de fichier qui aura alors une fréquence de 0.

```
64
65      -- Construction de la premier branche de l'arbre correspondant au tableau des fréquences
66      -- de chaque caractère présent dans le fichier texte
67      procedure Premiere_branche (file_name : in string ;
68                                                  tab_branche : in out T_tab_branche) is
69              neant : T_arbre;
70              file      : Ada.Streams.Stream_IO.File_Type;
71              S         : Stream_Access;
72              byte      : T_byte;
73
74      begin
75
76              -- Lire le fichier "file_name"
77              Open(file, in_file, file_name);
78              S := Stream(file);
79              Initialiser_tab_branche(tab_branche);
80
81              while not End_Of_File(File) loop
82                      byte := T_byte'Input(S);
83                      -- Incrémenter de 1 la fréquence associée à un octet présent dans le fichier
84                      Affecter(tab_branche(Integer'Val(byte)), byte, La_freq(tab_branche(Integer'Val(byte))) + 1, neant, neant);
85              end loop;
86
87              Close(file);
88
89              -- Considérer les octets non présents dans le fichier
90              for i in 1..2**8 loop
91                      if La_freq(tab_branche(i)) = 0 then
92                              -- Affecter à chaque arbre dont la fréquence est nulle une fréquence égale à file_size
93                              Affecter(tab_branche(i), Le_byte(tab_branche(i)), file_size + 1, neant, neant);
94                      end if;
95              end loop;
96
97      end Premiere_branche;
98
```

Construction de l'arbre de Huffman :

Pour construire l’arbre de Huffman, la procédure “ArbreH” appelle le sous-programme “Mins” qui recherche les deux arbres dont leur fréquence est minimale et affecte à la fréquence de l’arbre ayant les deux arbres trouvés par “Mins” en fils droit et gauche la somme des fréquences minimum. On continue ce processus jusqu’à ce que la fréquence d’un noeud soit égale à la taille du fichier.

```
153
154         -- Construction de l'arbre de Huffman
155         procedure ArbreH (tab_branche : in out T_tab_branche ;
156                                   arbre : in out T_arbre) is
157
158                 min1 : T_arbre;
159                 min2 : T_arbre;
160                 imin1 : integer;
161                 imin2 : integer;
162                 freq : integer;
163                 neant : T_arbre;
164                 zero : T_byte := T_byte(0);
165
166         begin
167                 Initialiser(neant);
168                 loop
169                         Initialiser(arbre);
170                         -- Chercher les arbres du tableau possèdant les deux plus faibles
171                         -- fréquences
172                         Mins(tab_branche, min1, imin1, min2, imin2);
173                         freq := La_freq(min1) + La_freq(min2);
174                         -- Affecter à l'arbre la nouvelle fréquence et les min1 et min2
175                         -- en fils_gauche et fils_droit respectivement
176                         Affecter(arbre, zero, freq, min1, min2);
177                         Initialiser(tab_branche(imin2));
178                         Affecter(tab_branche(imin2), zero, file_size + 1, neant, neant);
179                         tab_branche(imin1) := arbre;
180                         exit when freq >= file_size;
181                 end loop;
182
183         end ArbreH;
184
```

Construction du tableau de codage de Arbre :

```
203
204         -- Construire la table de codage de Huffman suivant un parcours infixe de l'arbre
205         procedure Tableau_code (arbre : in T_arbre ;
206                                           tab_code : out T_tab_code ;
207                                           c_code : in T_code ;
208                                           c_code_l : in integer ;
209                                           pi : out T_pi ;
210                                            bpi : out T_bpi ;
211                                           ipi : in out integer ;
212                                           ibpi : in out integer) is
213                 cp_arbre : T_arbre := arbre;
214                 c_code_l_n : integer := c_code_l;
215                 c_code_n : T_code := c_code;
216
217         begin
218                 ipi := ipi + 1;
219
220                 if Terminal(arbre) then
221                         ibpi := ibpi + 1;
222                         if Est_ff (arbre) then
223                                 tab_code(0).value := c_code_n;
224                                 tab_code(0).length := c_code_l;
225                                 bpi(0) := T_byte(ibpi);
226                                 ibpi := ibpi - 1;
227                         else
228                                 tab_code(Integer'Val(Le_byte(arbre)) + 1).value := c_code_n;
229                                 tab_code(Integer'Val(Le_byte(arbre)) + 1).length := c_code_l_n;
230                                 bpi(ibpi) := T_byte(Le_byte(arbre));
231                         end if;
232
233                 else
234                         c_code_l_n := c_code_l_n + 1;
235                         pi(ipi) := T_bit(0);
236                         c_code_n(c_code_l_n) := 0;
237                         Tableau_code (Branche_g(cp_arbre), tab_code, c_code_n, c_code_l_n,
238                                       pi, bpi, ipi, ibpi);
239                         c_code_n(c_code_l_n) := 1;
240                         pi(ipi) := T_bit(1);
241                         Tableau_code(Branche_d(cp_arbre), tab_code, c_code_n, c_code_l_n, pi,
242                                                 bpi, ipi, ibpi);
243                 end if;
244
245         end Tableau_code;
-- INSERTION --                                                    242,7-49      59%
```

Affichage de l'arbre de Huffman si l'option bavard est appelé :

Lorsque l'utilisateur entre "-b" ou "--bavard" dans la ligne de commande de compresser ou decompresser, la procédure "Affichage" du programme principal est appelée. Celle-ci va ensuite afficher l'arbre de Huffman, la table de Huffman, le nombre de caractères uniques dans le fichier, la taille du fichier initial et celle du fichier compressé.
L'affichage de l'arbre de Huffman est assuré par le sous-programme récursif "Afficher_arbreH" qui va afficher les branches indexé de 0 et 1 en suivant le parcours infixe de l'arbre mis en argument. Il affiche également les fréquences de chaque nœud. Si le nœud est terminal (c'est une feuille), la procédure affiche le caractère correspondant à l'octet contenu dans le nœud terminal.

DECOMPRESSION

Pour la compression, nous avons utilisé principalement les algorithmes du module decode, qui lui-même appelle les sous-programmes du module arbre notamment pour la construction de l'arbre de Huffman, les types associés aux arbres étant privés.

On rappelle que le fichier compressé est encodé par bloc :

1er bloc de un octet → la position de l'octet de fin de fichier dans le parcours infixe
2ème bloc → une fois chaque octet présent dans le fichier d'origine dans l'ordre du parcours infixe
3ème bloc de un octet → le délimiteur entre le bloc 2 et 4 consistant en la recopie du dernier octet du bloc 2
4ème bloc → parcours infixe de l'arbre
5ème bloc → codage de Huffman du fichier d'origine
6ème bloc de un octet → caractère de fin de fichier

Obtention du nombre de caractères uniques, de la longueur du parcours infixe et l'octet de fin de fichier du fichier encodé :

La procédure "Tailles" du module decode consiste à lire le fichier encodé en affectant à différentes variables le contenu du fichier. Plus précisément, le premier octet lu sera affecté à la variable i_ff et correspond à la position de l'octet de fin de fichier. Ensuite, la variable nbu_byte est incrémentée de 1 jusqu'à ce que deux octets soient égaux. La lecture sera alors arrivée au bloc 3 du fichier encodé. Dès lors, la variable lng_parcours_infixe sera incrémentée de 1 jusqu'à ce que le nombre de 1 dans le parcours infixe soit égal au nombre d'octets différents dans le fichier d'origine. (cf démonstration en annexe)

Reconstruction de l'arbre de Huffman :

La procédure "ReconstruireH" construit l'arbre de Huffman à partir du parcours_infixe construit préalablement suite à la lecture du fichier encodé (procédure "Lire" dans le module decode). Celui-ci est à un tableau de taille lng_parcours_infixe de type T_bits et correspond au 4ème bloc du fichier encodé. Dans ce même sous-programme, le tableau de feuilles est rempli par les différents caractères du fichier d'origine. Ainsi connaissant le parcours infixe, les feuilles de l'arbre de Huffman et l'octet de fin de fichier, la procédure "ReconstruireH" reconstruit l'arbre récursivement.

Décodage du fichier compressé :

A présent, il nous reste à lire le codage de Huffman du fichier d'origine qui correspond au 5ème bloc du fichier encodé. Ainsi la procédure "décoder" va lire le fichier et lorsqu'elle arrive au 5ème bloc du fichier encodé, elle va parcourir l'arbre de Huffman en suivant le parcours infixe déterminé précédemment jusqu'à arriver à une feuille puis écrire l'octet correspondant à la feuille dans un nouveau fichier. Le sous-programme va ensuite recommencer ces opérations jusqu'à ce qu'il lise le code de fin de fichier.

II.2 Tests

Le fichier de test passe pour la partie compresser mais un message d'erreur abscon est retourné lors de l'exécution pour la partie décompression. Ce message n'est cependant pas lié au programme de décompression en lui-même qui marche sans problème lorsqu'il est utilisé avec son interface en ligne de commande y compris avec le fichier du test.

II.3 Difficultés et solutions apportées (état d'avancement)

- Problème lié à la généricité de T_byte : initialement, nous avions mis le type T_byte générique cependant il s'est avéré trop compliqué de devoir tout instancier donc nous avons enlevé la généricité

- Certains fichiers .adb ne compile pas donc nous avons créé de nouveaux types et remodulé nos programmes afin qu'il fonctionne.

- Problèmes avec SVN

- S'occuper des warnings et résoudre les problèmes de compilation actuels : Bien que certains fichiers compilent, ils affichent de nombreux warnings. D'autres nous affichent des erreurs

- Renforcer le programme (robustesse) : S'occuper des cas de fichiers très courts ou vide

- Gérer les fuites de données : Utiliser valgrind et vérifier qu'il n'y a pas de fuite de données

III. Organisation de l'équipe

Problème d'organisation → Nous nous sommes mal organisés et nous y sommes pris trop tard. Alice devait s'occuper de la compression et Nicolas de la décompression sauf qu'Alice a eu beaucoup de mal avec les sous-programmes donc Nicolas a fait principalement du codage. Alice a toutefois construit les algorithmes d'affichage de l'arbre et de la table de Huffman ainsi que la gestion de commande et a rédigé le programme principal compression. Elle s'est particulièrement occupée du rapport et des documents rédigés à rendre.

IV. Bilan

IV.1 Bilan technique

COMPRESSION

Nous avons tout d'abord essayé de compresser le fichier donné dans le sujet pour vérifier notre arbre de Huffman. Le fichier exemple_huff.txt contient le texte suivant :

exemple de texte :
exempte tempete lexeme

Pour compresser ce fichier, l'utilisateur entre la ligne de commande ci-dessous :

$ ./compresser –b exemple_huff.txt

L'exécution de la ligne de commande devrait nous donner ceci :

```
Voici l'arbre de Huffman :
(42)
 \--0--(17)
 |    \--0--(8)
 |    |    \--0--(4) 'x'
```

```
|    |    \--1--(4) 'm'
|    \--1--(9)
|         \--0--(4)
|         |    \--0--(2) 'l'
|         |    \--1--(2)
|         |         \--0--(1) ':'
|         |         \--1--(1)
|         |              \--0--(0) '/$'
|         |              \--1--(1) 'd'
|         \--1--(5) 't'
\--1--(25)
     \--0--(10)
     |    \--0--(5) ' '
     |    \--1--(5)
     |         \--0--(2) '/n'
     |         \--1--(3) 'p'
     \--1--(15) 'e'
```

Voici la table de Huffman :
'/$' --> 00000000
'/n' --> 0001
' ' --> 001
':' --> 11000
'd' --> 111110
'e' --> 01
'l' --> 0000
'm' --> 000
'p' --> 1001
't' --> 001
'x' --> 000

Le nombre de caractère unique dans le fichier est 10

La taille du fichier initiale était 42

La taille du fichier compressé est 42
[2022-01-15 01:36:03] process terminated successfully, elapsed time: 00.20s

Ce résultat est issu de l'exécution du fichier test_interface.adb sur le fichier “exemple_huff.txt” car nous avons un problème pour compression (cf II.3).

On constate que l’arbre de Huffman est bien construit et correspond bien à l’arbre présenté dans le sujet cependant les codes de Huffman sont de la bonne taille mais ne correspondent pas au parcours de l’arbre. Ainsi il y a une erreur de parcours lors de la construction du tableau de code dans le module code mais nous n’avons pas réussi à la résoudre.

Cependant notre programme compresse bien le fichier d’entrée.

DECOMPRESSION

Notre programme décompresse bien.

IV.2 Bilans personnels

Alice Devilder :

Ayant quelques difficultés en programmation, ce projet a été très difficile à réaliser notamment au niveau de la démarche informatique. La compréhension du sujet a été également compliquée. Cependant, malgré les nombreuses difficultées rencontrées, ce projet a été très instructif.

Nicolas Catoni :

Ce projet a été marqué par de nombreuses difficultés principalement liées à une mésestimation du temps nécessaire à sa réalisation et à un manque d'organisation. Il a cependant été très instructif concernant la gestion d'un projet informatique d'une ampleur plus importante que ceux que j'avais pu mener auparavant. Il m'a par ailleurs permis d'acquérir une plus grande maîtrise du langage Ada. Nous avons par ailleurs rencontré des difficultés techniques liées à svn qui rencontre régulièrement des erreurs de connexions d'origine inconnu ou qui synchronise une version serveur antérieure sans soulever de conflit entraînant la perte des données locales. Ce qui est particulièrement gênant lorsque celà arrive le dernier soir.