Codage de Huffman

Tom BERTRAND - Maël MATHURIN

Equipe MN09

Toulouse - ENSEEIHT

Projet PIM - 1ASN

Table des matières

1 Objectif et contenu

1.1 Objectif

L'objectif du projet était de créer deux programmes permettant respectivement de compresser puis décompresser un fichier. Pour cela, on utilise le principe du codage de Huffman qui consiste à coder chaque caractère en fonction de son nombre d'occurences dans le fichier.

1.2 Contenu

Le projet contient donc les sources des programmes, les modules qu'ils utilisent ainsi que les fichiers test des modules. Il contient aussi le manuel d'utilisation (manuel.pdf), du script de démonstration (demo.pdf ou demo.txt), et de la présentation oral (presentaton.pdf).
Il y a aussi un fichier installateur nommé "linuxinstall_huff" qui installe les commandes comp et dcomp. Respectivement, ces commandes permettent de compresser/decompresser un fichier ou un dossier entier.

2 Introduction

Le problème est donc de compresser un fichier rempli de données (octet) afin de réduire la place qu'il prend dans le disque dur. Pour cela, on compte d'abord le nombre d'occurences de chaque octet dans le fichier, puis grâce à cela, on crée l'arbre de Huffman qui va nous donner le codage de chaque caractère. Pour que la compression soit le plus efficace possible, il faut que le codage de chaque charactère soit le plus court possible (sur le moins de bit possible).
Une fois l'arbre de Huffman, et donc la table de codage (ou table de Huffman) construits, il ne reste qu'à relire le fichier et pour chaque octet trouvé, écrire dans le fichier compressé le codage de cet octet.
Il y a donc trois grandes phases dans la compression d'un fichier : le calcul des occurences, la création de la table de codage, et l'écriture du fichier compressé. Pour la décompression, il va falloir tout d'abord recréer la table de décodage (table de codage inverse). Pour cela lors de la compression, on doit laisser des informations sur la table de codage et l'arbre dans le fichier compressé afin de pouvoir reconstruire l'arbre. Une fois cela fait, il ne reste plus qu'à lire le fichier compressé bit par bit et parcourir l'arbre en fonction des bits lus, et dès qu'on arrive sur une feuille (un arbre qui n'a ni arbre gauche ni arbre droit accroché à lui), on écrit l'octet correspondant dans le fichier décompressé.

3 Modules

Afin de réaliser ce programme, nous avons créé trois modules différents, chacun générique et donc réutilisable

3.1 Module : arbre

Le module arbre implémente le principe d'arbre binaire. Un arbre est composé d'une racine (l'élément en "haut") de l'arbre ainsi que deux sous-arbres, nommés arbre gauche et arbre droit. A chaque élément (noeud) est associé une valeur permettant une mesure de l'arbre. On peut se repérer dans un arbre grâce à un "chemin". Ce chemin est représenté par un entier supérieur ou égal à 1. Le chemin 1 correspond à

la racine. Sinon, on doit suivre le chemin : si le chemin est paire, alors on doit aller à gauche de l'arbre, sinon, on doit aller à droite. On divise ensuite le chemin en deux (en ne gardant que la partie entière) et on recommence l'opération jusqu'à ce que le chemin vaille 1. Par exemple si le chemin vaut 9, alors on va a droite de l'arbre, puis a gauche, puis encore a gauche (car 9=1+(0+(0+1*2)*2)*2 donc on suit le chemin 1->0->0 : droite-gauche-gauche).
On peut fusionner deux arbres ensemble, accéder à un élément de l'arbre, changer sa valeur, ou le supprimer, le vider, etc... Ce module est utilisé dans compression afin de créer l'arbre de Huffman et dans décompression afin de le reconstruire. Il est utilisé avec comme type T_Element le type T_Octet. Ici notre arbre sera toujours un arbre binaire parfait, c'est-à-dire que tout noeud à soit 0 sous-arbre, soit 2, mais jamais 1.

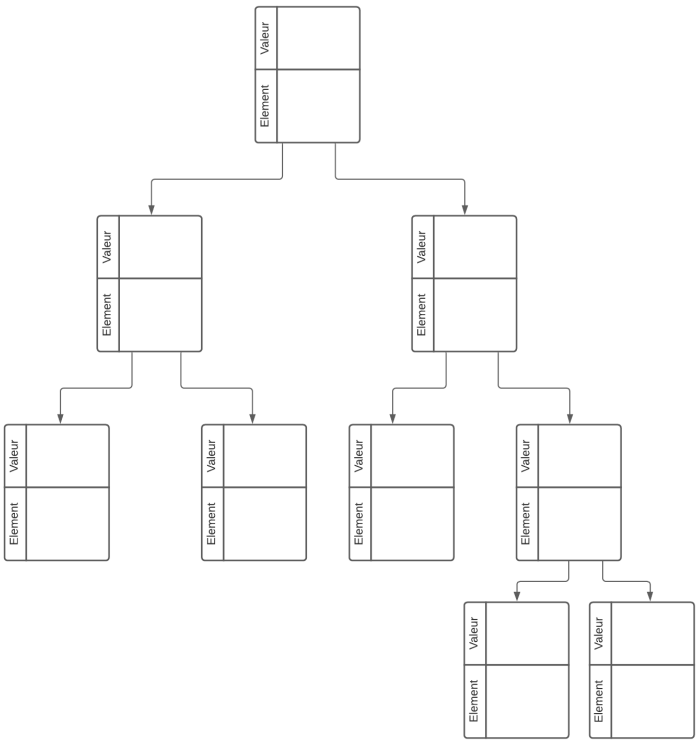

FIGURE 1 – Arbe binaire parfait

3.2 Module : liste_c

Le module liste_c implémente le principe de liste chaînée, c'est à dire une liste dynamique qui n'est limitée en taille que par la mémoire de l'ordinateur. Les éléments (du type que l'on souhaite) de cette liste sont indexés par des indices, commençant à 0 pour le premier élément. On peut bien sur ajouter des éléments à cette liste, en supprimer, lire leurs valeurs, trouver le minimum, le max, la vider, etc... Ce module est utilisé dans les programmes compression et décompression afin de définir ce qu'est un code

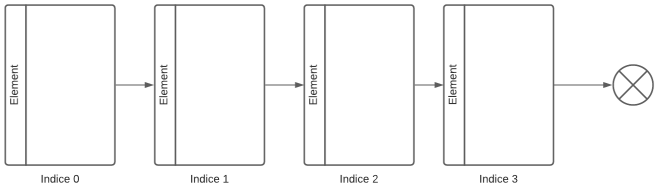

FIGURE 2 – Liste chainée

(une liste de bit), d'avoir une suite d'octet, ou encore une liste d'arbre.

3.3 Module : sda

Le module SDA (Structure de données associatives), implémenté ici sous la forme d'une LCA (liste chaînée associative) permet d'associer une donnée à une clé. Ainsi, chaque clé est présente une et une seule fois dans la sda et les recherches se font principalement grâce à celle-ci. Les clés peuvent être de n'importe quel type, mais il est nécessaire de donner la fonction qui permet de comparer deux clés pour savoir si celles si sont égales, afin d'accéder à la donnée recherchée. De même que pour les autres modules, on peut ajouter/supprimer des combinaisons de clé-donnée, ou de les modifier. Ce module est utilisé de deux façons différentes dans compression : une pour compter les occurrences de chaque caractère (Les clés sont des octets, et les données des entiers) et une autre pour la table de huffman (une clé est un octet, et la donnée le code associé). Ce module est aussi utilisé dans la décompression pour créer la table de décodage (une clé est un code et sa donnée est son octet associé).

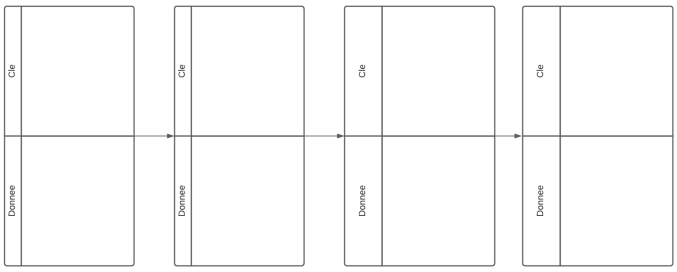

FIGURE 3 – SDA sous la forme d'une LCA

4 Types de données et algorithme

4.1 Les types de données

4.1.1 Types venants de ADA

On utilise plusieurs types prédéfinis en Ada comme :

1. T_Bit correspondant aux entiers de $\frac{\mathbf{Z}}{2\mathbf{Z}}$
2. T_Octet correspondant aux entiers de $\frac{\mathbf{Z}}{256\mathbf{Z}}$ C'est le type qui permet de lire le fichiers. Nous ne manipulons jamais ce qui à l'intérieur d'un fichier sous forme de caractère.
3. Integer le type correspondant aux entiers
4. String et Unbounded_String pour les chaînes de caractères

On utilise aussi les types provenant de modules de Ada comme :

1. Stream_Access permettant la lecture et l'écriture de fichier
2. File_Type permettant de "stocker" un fichier

4.1.2 Types provenants de nos modules

Les trois modules présentés au (3) définissent des types :

1. L'arbre contenant des octets : T_Arbre
2. Une liste chaînée contenant des bits : Pack_Bit_Liste.T_Liste
3. Une liste chaînée contenant des octets : Pack_Octet_Liste.T_Liste
4. Une SDA contenant des octets en clé et des entiers en éléments : Pack_Octet_Integer_SDA.T_SDA
5. Une SDA contenant des octets en clé et des codes en éléments : Pack_Octet_BitListe_SDA.T_SDA

4.1.3 Type option

Enfin, on défini aussi notre propre type énumération T_Option qui peut prendre comme valeur : NORMAL, MINIBAVARD ou BAVARD afin de connaître l'option choisie par l'utilisateur.

4.2 Algorithme

Lors de la compression d'un fichier, il est possible qu'un caractère soit codé sur plus de un bit (cela arrive même souvent). On peut alors se dire que le fichier n'est pas compressé puisque les caractères sont codés sur plus de bits qu'avant. En réalité, la compression fonctionne comme même car seul les caractères qui n'apparaissent peu de fois sont codés sur beaucoup de bits. Ceux qui apparaisse très souvent, et donc les plus importants à réduire sont bien souvent en dessous des 8 bits. Il est tout de même possible que dans certains cas, le fichier compressé prennent plus de place qu'avant.
Afin de reconstruire l'arbre de Huffman, nous plaçons au tout début du fichier décompressé l'octet correspondant à la position de l'octet de fin de fichier dans l'octet. On est assuré que cette position peut être codé sur un octet, car chaque octet différent est codé une fois, donc il existe au plus 255 code, en comprenant celui du fin de fichier. Donc la position de l'octet de fin de fichier est comprise entre 0 et

255, donc peut-être codé sur un octet.
Lors de la décompression, pour reconstruire la table de Huffman, nous le faisons en deux étapes. Premièrement, nous recréons la liste des octets à mettre dans la table de Huffman, dans l'ordre où ils doivent apparaître, grâce a la suite d'octets mise au début du fichier. Ensuite pour créer l'arbre, nous initialisons un code vide et un arbre vide. Puis, nous lisons bit par bit le fichier compressé. Si nous tombons sur un 0, nous ajoutons un fils gauche à la position actuelle donné par le code, puis nous ajoutons un zéro au code. Si on tombe sur un 1, on remplace l'élément de la position actuelle donné par le code de l'arbre par le prochain élément de notre table d'octet, puis, tant que le dernier élément du code est un 1, nous le supprimons. Enfin on remplace le dernier élément du code (donc un 0) par un 1. On recommence jusqu'à ce que la table d'octets soit vide, ou que le code soit vide (cela revient au même).
Enfin, pour décoder le fichier compresser, on réinitialise un code, et chaque fois que nous lisons un bit, nous l'ajoutons à ce code. Puis a chaque étape, nous vérifions si le code correspond à une feuille dans l'arbre de Huffman. Si oui, alors nous ajoutons l'octet correspondant au fichier décompressé. Sinon, nous continuons. On s'arrête jusqu'à lire le code correspondant à celui de fin de fichier et que nous sommes en train de lire le dernier octet du fichier compressé.

5 Tests du programme

Pour le test du programme nous l'avons exécuté sur plusieurs fichiers. Notre premier test est sur un fichier texte simple écrit en alphabet latin. Notre deuxième test est encore une fois sur un fichier texte mais avec des caractères cyrilliques à l'intérieur. Ces deux fichiers textes contiennent entre 40000 et 60000 caractères chacun. Notre troisième test est sur un fichier exécutable illisible par l'homme. Cet exécutable est l'exécutable de compression lui-même. Notre quatrième et dernier test est sur un dossier contenant plusieurs fichiers et sous-dossiers à l'intérieur. Ce dernier test sera fait grâce aux commandes "comp" et "dcomp" installées dans les commandes de l'ordinateur grâce a l'exécutable : linuxintstall_huffman. Le dossier sera composé des fichiers listés ci-dessus et d'une copie de lui même.

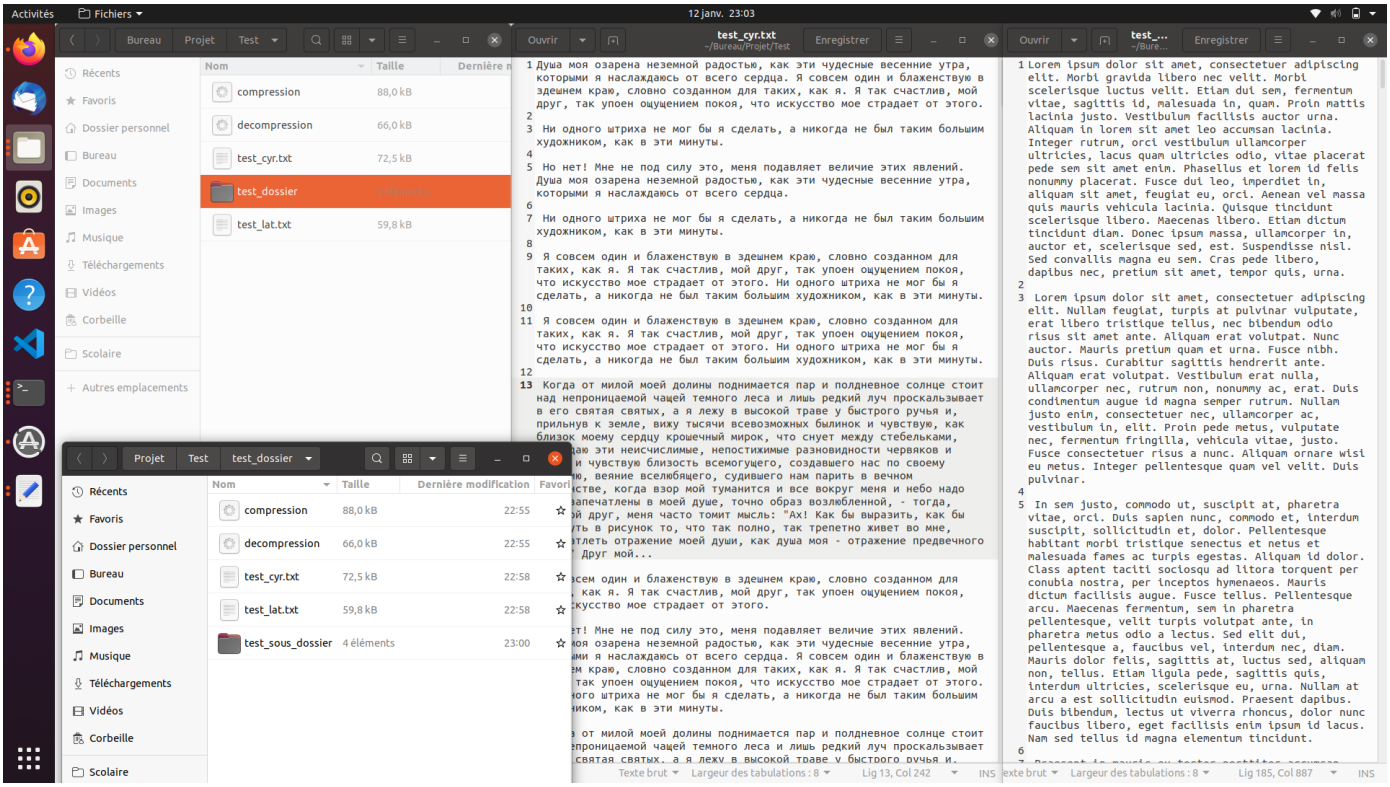

FIGURE 4 – Les différents fichiers de tests
Le fichier en cyrillique se trouve au milieu, celui en latin à droite, l'intérieur du dossier test est en bas à gauche, et nos exécutable sont dans le dossier en haut à gauche.

Nos tests on été compressés avec le mode BAVARD activé. Voici ce que cela donne pour le fichier en latin (seul exemple au vu de la place que cela prend) :

```
~/Bureau/Projet/Test$ ./compression -b test_lat.txt
Compression du fichier 'test_lat.txt' ...
Calcul des occurrences ...
'\$'-->0
'L'-->22
'o'-->1965
'r'-->2780
'e'-->5322
'm'-->2476
' '-->8714
'i'-->4685
'p'-->1217
's'-->3878
'u'-->4552
'd'-->1145
'l'-->2813
't'-->4064
'a'-->3986
','-->1105
'c'-->1787
'n'-->2853
'g'-->580
'.'-->1462
'M'-->187
'b'-->615
'v'-->695
'q'-->653
'E'-->93
'f'-->292
'P'-->161
'j'-->110
'V'-->74
'A'-->140
'I'-->178
'h'-->251
```

```
'y'-->51
'F'-->93
'Q'-->19
'D'-->103
'S'-->61
'C'-->103
LF-->199
'N'-->217
'w'-->46
'U'-->11

Calcul de l'arbre de Huffman ...
(59758)
  \--0--(24337)
  |     \--0--(10915)
  |     |     \--0--(5322) 'e'
  |     |     \--1--(5593)
  |     |           \--0--(2780) 'r'
  |     |           \--1--(2813) 'l'
  |     \--1--(13422)
  |           \--0--(5792)
  |           |     \--0--(2853) 'n'
  |           |     \--1--(2939)
  |           |           \--0--(1462) '.'
  |           |           \--1--(1477)
  |           |                 \--0--(695) 'v'
  |           |                 \--1--(782)
  |           |                       \--0--(377)
  |           |                       |     \--0--(187) 'M'
  |           |                       |     \--1--(190)
  |           |                       |           \--0--(93) 'E'
  |           |                       |           \--1--(97)
  |           |                       |                 \--0--(46) 'w'
  |           |                       |                 \--1--(51) 'y'
  |           |                       \--1--(405)
  |           |                             \--0--(199) LF
  |           |                             \--1--(206)
  |           |                                   \--0--(103) 'C'
  |           |                                   \--1--(103) 'D'
  |           \--1--(7630)
  |                 \--0--(3752)
  |                 |     \--0--(1787) 'c'
  |                 |     \--1--(1965) 'o'
  |                 \--1--(3878) 's'
  \--1--(35421)
        \--0--(16764)
        |     \--0--(8050)
        |     |     \--0--(3986) 'a'
        |     |     \--1--(4064) 't'
        |     \--1--(8714) ' '
        \--1--(18657)
              \--0--(8980)
              |     \--0--(4428)
              |     |     \--0--(2088)
              |     |     |     \--0--(983)
              |     |     |     |     \--0--(440)
              |     |     |     |     |     \--0--(217) 'N'
              |     |     |     |     |     \--1--(223)
              |     |     |     |     |           \--0--(110) 'j'
              |     |     |     |     |           \--1--(113)
              |     |     |     |     |                 \--0--(52)
              |     |     |     |     |                 |     \--0--(22) 'L'
              |     |     |     |     |                 |     \--1--(30)
              |     |     |     |     |                 |           \--0--(11)
              |     |     |     |     |                 |           |     \--0--(0) '\$'
              |     |     |     |     |                 |           |     \--1--(11) 'U'
              |     |     |     |     |                 |           \--1--(19) 'Q'
              |     |     |     |     |                 \--1--(61) 'S'
              |     |     |     |     \--1--(543)
              |     |     |     |           \--0--(251) 'h'
              |     |     |     |           \--1--(292) 'f'
              |     |     |     \--1--(1105) ','
              |     |     \--1--(2340)
              |     |           \--0--(1145) 'd'
              |     |           \--1--(1195)
              |     |                 \--0--(580) 'g'
              |     |                 \--1--(615) 'b'
              |     \--1--(4552) 'u'
              \--1--(9677)
                    \--0--(4685) 'i'
                    \--1--(4992)
```

```
                        \--0--(2476) 'm'
                        \--1--(2516)
                              \--0--(1217) 'p'
                              \--1--(1299)
                                    \--0--(646)
                                    |     \--0--(301)
                                    |     |     \--0--(140) 'A'
                                    |     |     \--1--(161) 'P'
                                    |     \--1--(345)
                                    |           \--0--(167)
                                    |           |     \--0--(74) 'V'
                                    |           |     \--1--(93) 'F'
                                    |           \--1--(178) 'I'
                                    \--1--(653) 'q'

Calcul de la table de codage ...
'e'-->000
'r'-->0010
'l'-->0011
'n'-->0100
'.'-->01010
'v'-->010110
'M'-->01011100
'E'-->010111010
'w'-->0101110110
'y'-->0101110111
LF-->01011110
'C'-->010111110
'D'-->010111111
'c'-->01100
'o'-->01101
's'-->0111
'a'-->1000
't'-->1001
' '-->101
'N'-->11000000
'j'-->110000010
'L'-->11000001100
'\$'-->1100000110100
'U'-->1100000110101
'Q'-->110000011011
'S'-->1100000111
'h'-->11000010
'f'-->11000011
','-->110001
'd'-->110010
'g'-->1100110
'b'-->1100111
'u'-->1101
'i'-->1110
'm'-->11110
'p'-->111110
'A'-->111111000
'P'-->111111001
'V'-->1111110100
'F'-->1111110101
'I'-->111111011
'q'-->1111111

Écriture du fichier compresse 'test_lat.txt.hff' ...
Calcul de la suite de bits ...
000101100101010010101101011001110001110000001010010011110111010111010101000110011111
Compression de 'test_lat.txt' terminée.
Taux de compression : 46%.
```

On peut voir que le taux de compression est 46%, donc que le fichier compressé prend quasiment deux fois moins de place qu'auparavant. Voici le tableau des taux de compression que l'on obtient pour chaque test :

Fichier test	Taux de compression
Fichier en latin	46%
Fichier en cyrillique	50%
Fichier binaire exécutable	35%
Dossier test	43%

En tout, près de 404ko ont été sauvés sur 866ko, soit 53% d'espace récupéré. Une fois cela fait, on déplace tous les fichier dans un autre dossier afin de les décompresser sans supprimer les originaux, pour pouvoir comparer. On peut alors constater que tous les fichiers décompressés et originaux sont bien exactement identiques.
Il s'agit ensuite de vérifier les fuites mémoires. Pour cela, nous avons utilisé la commande valgrind : valgrind ./compression test_lat.txt. Il nous dit donc que l'on a fait 258,762 allocation, autant de libération et que tout a bien été désalloué. On obtient le même résultat pour décompression. Nous avons répété le processus pour chaque fichier test, et à chaque fois il n'y a pas de fuite mémoire. On peut donc supposer que notre code ne génère pas de fuite.

6 Difficultés rencontrées

Il n'y a pas forcément eu de grosses difficultés, mais plutôt plein de petites erreurs ou problèmes lors de l'implémentation. Par exemple, il y a eu des problèmes avec les listes lors de la création de la table de codage lors de compressions. On modifiait notre code avant de l'injecter dans notre table, mais du coup, lorsque on re-modifiait le code pour les octets d'après, on modifiait aussi notre ancien code. Une solution pour régler ce problème à été de copier notre code dans un autre code, et de mettre cette copie dans la table de codage. Un autre problème était le temps de décompression qui été bien plus long que celui de compression. Ce problème venait du fait que l'on utilisait une table de décodage (SDA) lors de la décompression , et donc qu'à chaque lecture d'un bit dans le fichier compressé, on devait parcourir entièrement la table pour vérifier si notre code correspondait à un code existant, ce qui ralentit le programme. Pour corriger cela, nous avons décidé de recréer l'arbre de Huffman à la place du tableau de décodage, et il suffit alors de vérifier si notre code correspond à une feuille dans notre arbre pour savoir si il correspond à un code réel. Notre première méthode avait une complexité de $O(n \times m)$ avec n le nombre de caractères et m le nombre de caractères différents présents de le fichier. Notre deuxième méthode quand à elle a une complexité de $O(n \times log_2(m))$, ce qui est bien plus efficace.

7 Bilan technique

Nos programmes semblent très bien fonctionner pour tous types de fichier. Il ne peut pas compresser encore plus un fichier déjà fortement compressé (comme un pdf par exemple) où dans ce cas le taux de compression est très faible (voir négatif). On peut cependant reprocher à notre code plusieurs défauts :

1. Dans notre code pour compression nous utilisons une variable globale afin de stocker le nombre d'octets écrits dans le fichier, afin de calculer le taux de compression. On pourrait mettre cette variable en In Out

pour chaque sous-programme qui s'en sert.

2. Dans notre code pour compresser encore une fois, nous utilisons à plusieurs reprises des sous-sous-programmes à l'intérieur de sous-programmes qui utilisent les variables du sous-programme sans les avoir en entrée, dans le but de créer des procédures "Pour_Chaque" qui agissent sur d'autre donnée que uniquement sur la donnée rentrée en paramètre. Exemple :

```
procedure Initialiser_Liste_Arbre_Huffman(Tableau_Occurrences : in out Pack_Octet_Integer_Sda.T_SDA;
                                          Liste_Arbre_Huffman : out Pack_Arbre_Liste.T_Liste) is
    procedure Arbre_Elementaire(Octet : in out T_Octet; Occurrence : in out Integer) is
        Arbre : T_Arbre;
    begin
        Initialiser_Valeur(Arbre, Occurrence, Octet);
        Pack_Arbre_Liste.Ajouter_Fin(Liste_Arbre_Huffman, Arbre);
    end Arbre_Elementaire;

    procedure Arbre_Elementaires is new Pack_Octet_Integer_Sda.Pour_Chaque(Arbre_Elementaire);
begin
    Pack_Arbre_Liste.Initialiser(Liste_Arbre_Huffman);
    Arbre_Elementaires(Tableau_Occurrences);
end Initialiser_Liste_Arbre_Huffman;
```

Ici, la sous-procédure "Arbre_Elementaire" ne prend pas en argument la liste d'arbre de Huffman mais modifie celle rentrée en argument lors de l'appel à la procédure principale "Initialiser_Liste_Arbre_Huffman". Nous n'avons pas trouvé un moyen efficace de contourner ce problème sans perdre de la généricité de nos modules.

3. Enfin, le troisième problème qu'on a pu noter, et ceci valable pour les deux programmes, est le nombre de sous-fonctions présentes dans nos codes, ce qui a causé une longue perte de temps pour l'écriture des spécifications. Nous avons voulu créer une sous-fonction pour chaque raffinage produit afin d'avoir une compréhension simplifiée de notre progression dans notre programme et d'éviter de se perdre.

8 Organisation lors du projet

Pour le projet, nous avons essayé de nous diviser équitablement les tâches.

Raffinage compression	Raffinage décompression	Pack SDA	Pack liste_c	Pack arbre
MATHURIN Maël	BERTRAND Tom	MATHURIN Maël	BERTRAND Tom	MATHURIN Maël
Écriture compression	Écriture décompression	Tests des modules	Rapport	Diapo présentation
MATHURIN Maël	BERTRAND Tom	BERTRAND Tom	MATHURIN Maël	BERTRAND Tom
Spécification compression	Spécification décompression	Manuel d'utilisation	Démo	Installateur linux
BERTRAND Tom	MATHURIN Maël	MATHURIN Maël	BERTRAND Tom	MATHURIN Maël

Bien sûr, chacun a regardé ce que faisais l'autre à chaque étape de la création pour être sûrs de comprendre et d'éviter les erreurs. Certaines étapes étaient plus longues que d'autres, nous avons donc essayé de partager en fonction du temps des tâches et pas en fonction du nombre de tâches.

9 Bilans personnels et individuels

9.1 Tom BERTRAND

Le projet m'a beaucoup intéressé et a pris beaucoup de temps à réaliser. Le raffinage de décompression a pris entre 3 et 6h, un peu plus court que celui de compression car j'ai pu m'inspirer de celui de mon camarade pour certaines étapes et entre 15 et 20h pour écrire le code de décompression. Le module liste_c a été plus rapide puisqu'il n'a pris que 6h, le plus long étant de créer les nouvelles fonctions pour rendre le module le plus générique et polyvalent possible. La structure que nous avons choisi utilise énormément de sous programmes dans compresser et décompresser, une 60aine en tout ce qui nous a poussé à écrire autant de spécifications uniquement pour les codes de compresser et décompresser, pour ma part compresser comptait 40 sous-programmes, il m'a fallu environ 5h pour les spécifier. Les programmes test des modules quant à eux ont été réalisés en environ 6h. La plupart du code a été fait dans les locaux de l'N7 car ma connexion supportait mal la connexion à distance. La partie la plus désagréable était les raffinages mais je pense que c'était une étape indispensable au bon déroulé du développement de nos programmes. Je pense que ce projet m'a beaucoup apporté, il m'a appris à manipuler de nouveaux types de données et à structurer un programme long.

9.2 Maël MATHURIN

Personnellement, le projet m'intéressait beaucoup. J'ai passé plusieurs heures dessus pour avoir un programme efficace. Les raffinages de compression m'ont pris entre 5 et 10 heures je pense, et environ une vingtaine d'heures pour l'écriture des programmes/modules/etc.. J'ai programmé aussi en bonus l'installateur qui permet d'avoir un accès rapide aux programmes de compression et décompression depuis n'importe quel dossier, et de compresser des dossiers entiers. Cela m'a pris environ 5h aussi. Personnellement, les parties que j'ai le moins aimé sont les raffinages, qui m'ont pris beaucoup de temps même si cela a été utile pour la conception après.