École Centrale de Lille

Algorithmique Avancée et Programmation

Fil Rouge 2022 - Codage de Huffman

Compte-rendu

Nom	**Prénom**
AQUILO	Axel
GOSMANN	Gabriel
DOS SANTOS SILVA	Vitor
STEFFENS WONTROBA	Vinícius

Villeneuve-d'Ascq
2023

École Centrale de Lille

Fil Rouge 2022 - Codage de Huffman

Compte-rendu

Compte-rendu du projet présenté aux évaluateurs du sujet Algorithmique Avancée et Programmation de l'École Centrale de Lille.

Villeneuve-d'Ascq
2023

Table des matières

Table des figures

Trees sprout up just about
everywhere in computer science.

Donald Knuth

1. Introduction

1.1 Compréhension du problème

En informatique et en théorie de l'information, un code de Huffman est un type particulier de code de préfixe optimal couramment utilisé pour la compression de données sans perte. Le processus de recherche ou d'utilisation d'un tel code se déroule au moyen du codage de Huffman, un algorithme développé par David A. Huffman alors qu'il était Sc.D. étudiant au MIT, et publié dans l'article de 1952 "A Method for the Construction of Minimum-Redundancy Codes" [2].

La sortie de l'algorithme de Huffman peut être considérée comme une table de codes de longueur variable pour coder un symbole source (tel qu'un caractère dans un fichier). L'algorithme dérive cette table de la probabilité ou de la fréquence estimée d'occurrence (pondération) pour chaque valeur possible du symbole source.

Le problème consiste donc à coder de la manière la plus optimale possible la méthode de criblage de Huffman. Plus que cela, il a été demandé que des comparaisons soient faites - afin de vérifier l'efficacité de l'algorithme - avec des méthodes présentées précédemment en classe. En ce sens, le problème est divisé en trois parties :

1. Premier programme : *heapsort.exe* ;
2. Deuxième programme : *codage.exe* ;
3. Troisième programme : *huffman.exe*.

Notez qu'il existe une dépendance intrinsèque entre les programmes demandés. L'objectif de la méthode de Hufman étant justement d'effectuer la compression d'un certain nombre de données, il faut, au préalable, que ces données soient organisées de manière à ce que l'algorithme soit réellement efficace. Plus que cela, puisque le premier programme effectue le tri des éléments *T_elt* (dont la nature du tri peut être sélectionnée par une constante symbolique), le second effectue lui-même le codage de Huffman et le troisième comprime les données codées et filtrées, il Il est clair que les résultats et les conclusions dépendent de la mise en œuvre correcte de toutes les parties de tous les programmes développés. Par conséquent, la solution réussie du problème ne peut pas être attribuée uniquement à la parcelle qui effectue le codage de Huffman, mais à toutes les parcelles en

union.

1.2 L'organisation du programme et du groupe

1.2.1 Le programme

Comme mentionné, le développement de trois programmes différents a été demandé. Cela dit, une section spécifique sera attribuée à chacun de ces programmes. Les lignes de code ne seront pas présentées dans leur intégralité, seulement des détails importants qui peuvent réellement contribuer à la compréhension de la solution au problème, comme le prototypage de certaines fonctions ou lignes fondamentales de *output*.

1.2.2 Le groupe

L'équipe a été divisée pour que la mise en œuvre se fasse de manière optimale.

— AQUILO Axel : Comparaison expérimentales des vitesses des tris et révision du Compte-Rendu ;

— GOSMANN Gabriel : Développement du Programme 2 : coddage.exe ;

— DOS SANTOS SILVA Vitor : Développement du Programme 3 : heapsort.exe ;

— STEFFENS WONTROBA Vinícius : Développement du Programme 1 : heapsort.exe et élaboration du Compte-Rendu.

2. Développement

2.1 Programme 1 : heapsort.exe

Le programme en question effectue le tri des éléments de type *T_elt* (dont la nature peut être préalablement définie par l'utilisateur) en utilisant différentes méthodes : quickSort, mergeSort et la nouveauté, heapSort. Les deux premières avaient déjà été développées dans l'activité *TEA 3* et leur but dans ce travail est juste de comparer les différentes méthodes.

Cela dit, traitant spécifiquement du programme effectivement demandé, le tri en tas peut être considéré comme un tri de sélection amélioré : comme le tri par sélection, le tri en tas divise son entrée en une région triée et une région non triée, et il réduit de manière itérative la région non triée en en extrayant le plus grand élément et en l'insérant dans la région triée. Contrairement au tri par sélection, le tri en tas ne perd pas de temps avec un balayage en temps linéaire de la région non triée ; plutôt, le tri par tas maintient la région non triée dans une structure de données de tas pour trouver plus rapidement le plus grand élément à chaque étape.

Notez que nous utilisons heapSort et tri en Tas ou même tri par Tas comme synonymes.

Dans la fonction *main.c*, l'utilisateur doit sélectionner la méthode de filtrage à utiliser. S'agissant spécifiquement de la méthode heapSort, la fonction en charge du travail est précisément *void heapSortTest()*. Cette fonction remplit un tableau de taille *MAX_ELT* avec des valeurs entières aléatoires. Une fois ceci fait, la fonction *T_elt * heapSort(T_elt * d, int n)* est appelée, qui applique en fait l'algorithme. Il est à noter que l'application de la méthode peut également être personnalisée entre Maximier et Minimier, une option également sélectionnée par l'utilisateur.

Le programme compte le nombre d'opérations et de comparaisons de l'algorithme et les renvoie à l'écran. Ces données sont exportées vers un fichier au format (de préférence) .csv, qui est ensuite tracé et analysé. Les résultats seront présentés dans le chapitre spécifique de conclusion et d'analyse des résultats.

Il convient de mentionner que la complexité de toutes les méthodes devrait se rapprocher d'une fonction *nlog(n)*.

2.2 Programme 2 : codage.exe

Le programme en question doit construire l'arbre de Hufmann, afin qu'il effectue le codage nécessaire pour la future compression des données. Ce programme est en fait divisé en trois fichiers : main.c, huffman.c et visuals.c. Leurs utilitaires respectifs, données d'entrée et de sortie et autres fonctionnalités seront discutés individuellement ci-dessous, dans lequel chaque section traitera de chaque fichier .c.

2.2.1 main.c

La fonction *main.c* est capable de recevoir des données, c'est-à-dire d'être appelée, de trois manières différentes :

1. Saisie de texte directement à partir du clavier ; ;
2. Rediriger n'importe quel fichier texte via la ligne de commande à l'aide de la fonction "<". ;
3. Rediriger n'importe quel fichier texte via la ligne de commande en utilisant la ressource "cat".

Initialement, une constante symbolique appelée *PRINT_HUFFMAN* a été définie, qui peut recevoir les valeurs 0 ou 1. Le programme exécutera la partie graphique si la valeur insérée est 1, et ne s'exécutera pas sinon. Ceci est très utile du point de vue de l'exécution, car vous ne souhaitez pas toujours voir la construction des images partie par partie. Par conséquent, changer la constante à zéro peut se traduire par un grand gain de temps pour l'utilisateur - et aussi, bien sûr, pour le programmeur.

Comme mentionné précédemment, l'une des bases de la méthode de Huffman est construite sur les fréquences avec lesquelles les caractères d'un texte apparaissent. La fonction chargée d'effectuer précisément cette comptabilisation est prototypée comme *int fréquence_count(t_ind_heap * heap, char * str)*. Notez qu'il renvoie une valeur de type *int*, qui est exactement 8 fois le paramètre *str* qu'il reçoit. Ceci est important car de cette façon, vous pouvez obtenir le nombre de bits de codage.

D'une façon générale, ce que la fonction *main.c* fait en fait, c'est appeler les fonctions qui font le "travail dur" et renvoient les résultats demandés. Ces résultats seront présentés en détail dans les prochains chapitres.

2.2.2 huffman.c

C'est le fichier qui est en fait responsable du tri via la méthode huffman. En ce sens, tous les "déplacements" entre les nœuds d'un arbre sont effectués par ce fichier jusqu'à ce qu'une condition prédéfinie soit atteinte. Ces mouvements sont effectués afin d'organiser les fréquences d'apparition de chaque caractère utilisé dans une séquence de texte donnée. Cette procédure est fondamentale car c'est elle qui optimise la méthode. Ces mouvements sont effectués par des fonctions telles que :

— int extract_root (t_ind_heap * tas)

— void move_up (t_ind_heap * tas, int index)

— void move_down (t_ind_heap * tas, int index)

— int look_for (t_ind_heap tas , int huffman[], int target, t_trace trace, int * encoding)

— void swap(t_ind_heap * tas , int index_parent, int index_child)

La mise en œuvre de Huffman peut être résumée, en termes généraux, comme la corrélation entre trois vecteurs distincts :

— Un vecteur appelé *tree* qui stocke chaque caractère représenté en ASCII et sa fréquence respective ;

— Un vecteur appelé *data*, d'une taille maximale de 255, qui stockera la fréquence de chaque caractère de la boîte dont l'index est équivalent à sa représentation ASCII, et donc, les fréquences jointes de l'union des caractères ;

— Un vecteur appelé *huffmanTree*, également d'une taille maximale de 255, qui décrit l'organisation des nœuds au sein du TAS ;

Une fois qu'on dispose d'un arbre équilibré et des fréquences de chaque caractère, une compression - ou traduction en un "pseudo-binaire" [1] - du texte initial est effectuée. Chaque caractère reçoit une séquence de zéros et de uns qui le représente. L'arbre de codage, le code généré et la conclusion sont alors affichés à l'écran.

1. C'est une représentation textuelle binaire d'un fichier, c'est-à-dire rien de plus qu'une traduction dans le but de simplifier et d'optimiser l'*utilisation* de la mémoire. La nomenclature "pseudo-binaire" a été utilisée afin qu'il n'y ait pas de confusion avec l'application binaire réelle concernant l'*allocation* de mémoire, dans laquelle chaque caractère est effectivement alloué et représenté par une séquence de zéros et de uns.

En même temps que l'intégralité de la méthode huffman est exécutée, des fichiers sont générés qui représentent graphiquement l'ordre de l'arbre et la construction du codage huffman. Ce tracé est réalisé avec les outils *graphviz* et sa présentation est donnée par les fonctions :

— void print_huffman (t_ind_heap tas , int huffman[], int target, int data[])

— void print_huffman_rec (t_ind_heap tas , int huffman[], int target, FILE * fp, int data[])

Comme auparavant, le processus de production graphique étant très coûteux (en temps et en mémoire), l'utilisateur a la possibilité de décider de réaliser ou non cette création. En ce sens, au début du code, une constante symbolique *PRINT_TREE* est fournie, qui ne prend que 0 ou 1.

Les sorties, arbres, tableaux, codes compressés et autres résultats sont présentés dans le chapitre de conclusion.

Certaines des fonctions utilisées ici ont été écrites de toutes pièces, sans reposer sur des codes tout faits ou fournis en cours. Pour d'autres cas, cependant, ce qui a été fourni comme matériel pendant les cours a été utilisé.

2.2.3 visuals.c

Il génère en effet des fichiers images des arbres à équilibrer, depuis leur origine et tout au long de leur parcours. Comme précédemment, nous avons utilisé les ressources fournies par l'outil *graphviz*. Les résultats sont présentés dans la section correspondante.

2.3 Programme 3 : huffman.exe

Ce programme remplit enfin la fonction de compression et de décompression de n'importe quel fichier texte. Le choix de la compression ou de la décompression se fait en fonction du nombre d'arguments (argc) que reçoit la fonction main. Si cette valeur est égale à 3 (fichier .exe, fichier à compresser, fichier résultat), l'opération sera une compression ; si égal à 2 (fichier .exe, fichier à décompresser), l'opération sera une décompression.

Pour le premier cas, la fonction qui effectue la compression est prototypée comme *void compress(char * str, FILE *fp)*. En termes généraux, la mise en œuvre de cette

fonction est très simple, c'est-à-dire qu'elle correspond exactement à ce à quoi vous vous attendez. Cependant, la grande différence de cette implémentation se trouve dans la façon dont l'en-tête huffman a été conçu. Celui-ci est présenté à l'écran à l'utilisateur en deux lignes, contenant les deux informations clés pour la compression :

— Le nombre de caractères uniques dans le texte ;

— Chaque caractère suivi de sa fréquence *représenté en caractère ASCII.*

Le choix de la représentation ASCII devient un atout qui rend la mise en œuvre beaucoup plus efficace, utilise moins d'informations pour véhiculer la même idée et évite d'éventuels problèmes de transformation et de traduction entre caractères pseudo-binaires et caractères simples (pensez à une fréquence de 10, par exemple. Est-ce vraiment une fréquence ou un caractère représenté par la séquence pseudo-binaire 10 ?). En ce sens, la fréquence peut être obtenue précisément grâce à la valeur représentée par le caractère en ASCII soustrait de 32. La justification de la soustraction par la valeur 32 provient précisément de la table ASCII elle-même. Le premier symbole traduit graphiquement par le tableau est précisément le symbole de l'espace, qui est représenté par le nombre 32. Ce n'est qu'à partir de cet index que l'on peut obtenir des symboles non seulement visuellement possibles mais aussi qui apportent une optimisation à la mise en œuvre.

Ces deux éléments sont fondamentaux pour la compression de tout fichier texte, car ils fonctionnent comme la clé de cryptage appliquée. De plus, les premières données détermineront le point final des secondes données, qui à leur tour déterminent l'ensemble du fonctionnement de l'algorithme de Huffman, montrant la fréquence de chaque caractère et donc quelle sera la représentation respective. Ceci est essentiel car de cette façon le programme peut savoir où commence et où se termine le "chiffrement" et de cette façon où il doit réellement commencer à traduire le texte.

Toujours traitant de la sortie, les lignes suivantes montrent, enfin, la représentation en zéros et en uns du texte initialement inséré. Ainsi, le résultat attendu est obtenu qui est concaténé dans un fichier .txt préalablement établi. On s'attend à ce qu'il y ait une réduction significative de la taille occupée en mémoire par le fichier. Ce résultat, entre autres, est présenté dans la section suivante.

Pour le deuxième cas, en supposant que seuls deux arguments ont été passés via la ligne de commande, il y aura un cas de décompression. Le fichier à utiliser sera exactement le fichier qui a été précédemment compressé. La fonction *void decompress(char * str)* fait

ce travail et, comme avant, c'est une idée plutôt intuitive. Sauf, bien sûr, la représentation d'en-tête de Huffman, qui suit le modèle appliqué dans le cas précédent.

Mis à part les fonctions de compression et de décompression, le programme en question est similaire au programme précédent, codage.exe, si bien que le fichier lui-même *huffman.c* est le même que celui utilisé dans le cas précédent. Cela rend clair la dépendance qui existe entre un programme et un autre, comme mentionné précédemment.

3. Conclusion et analyse des résultats

Ce chapitre sera séparé en 3 sections, chacune correspondant à chacun des programmes demandés.

3.1 Programme 1 : heapsort.exe

L'image ci-dessous présente les courbes de complexité de chacun des tri algorithmes demandés.

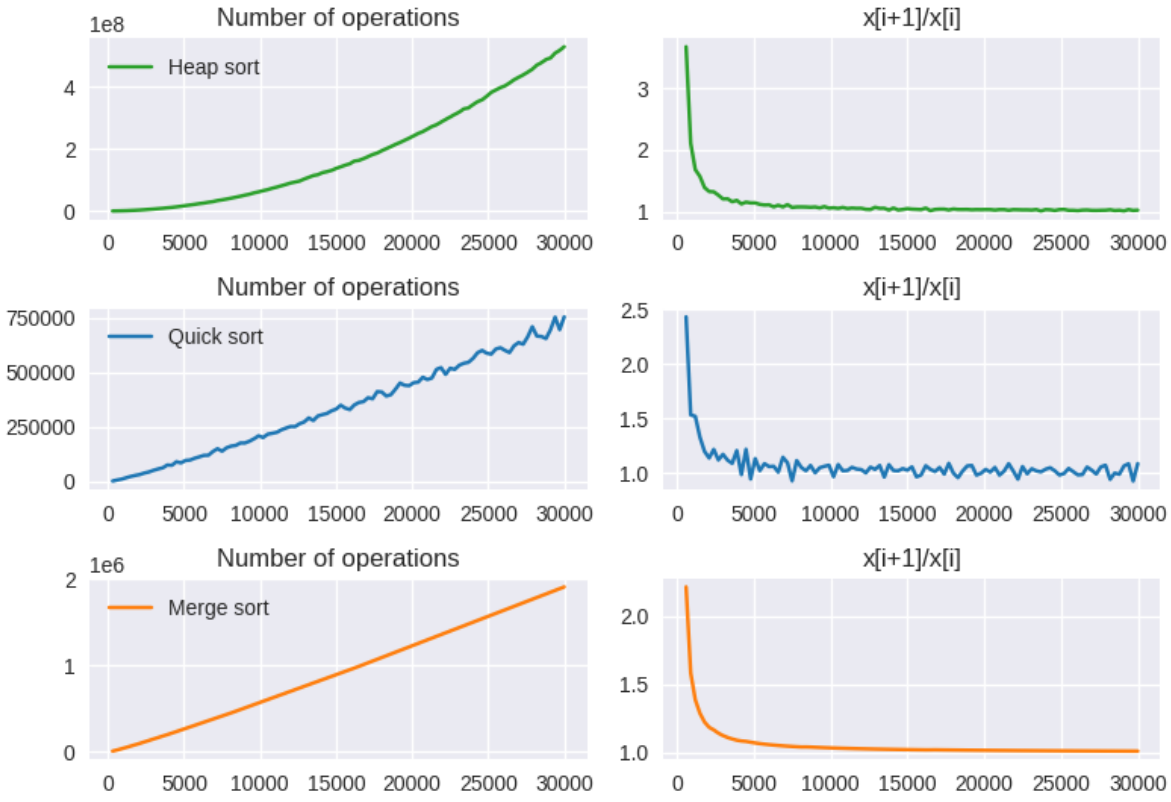

FIGURE 3.1 – Comparaison entre les différents algorithmes

Comme prévu, toutes les courbes montrent un comportement semblable à *nlog(n)*. Cependant, il n'est pas du tout évident que ce fait soit vrai en observant simplement les courbes présentées. Pour donner plus de crédibilité à l'argument selon lequel il s'agit en fait de courbes *nlog(n)*, les courbes de dérivées numériques des fonctions générées ont également été tracées. Notez qu'elles convergent toutes, d'une manière ou d'une autre, vers 1. C'est un comportement typique des fonctions de type *nlog(n)*, ce qui prouve l'argument initial.

De plus, il est intéressant d'observer la différence entre l'efficacité des algorithmes. La courbe en bleu, qui représente le *quickSort* est clairement la vitesse la plus élevée, qui en fait était attendue. Plus que cela, la courbe verte (*heapSort* est environ 100 fois plus lente que la courbe *mergeSort*.

Enfin, il est intéressant d'observer la "nature" de chacune des courbes. Comme mentionné, il n'est pas du tout évident qu'il s'agisse de courbes *nlog(n)*. Dans le premier algorithme, par exemple, on peut voir un comportement qui se rapproche d'une fonction parabolique/polynomiale. Dans le second, une fonction linéaire qui subit des perturbations pour de très grandes valeurs. Dans ce dernier cas, une fonction très similaire à une fonction linéaire.

3.2 Programme 2 : codage.exe

Le premier test a été effectué à l'aide d'un texte fourni par la ligne de commande. Exactement le même texte fourni dans le matériel a été utilisé. La sortie du programme est illustrée ci-dessous :

```
$ ./codage "algorithme de huffman pour la compression de chaines"

car : occ | long | bits
----+-----+------+--------
' ' :   7 |    3 | 000
'a' :   4 |    4 | 0100
'c' :   2 |    5 | 11010
'd' :   2 |    5 | 11011
'e' :   5 |    3 | 001
'f' :   2 |    5 | 11100
'g' :   1 |    6 | 111110
'h' :   3 |    4 | 0101
'i' :   3 |    4 | 0110
'l' :   2 |    5 | 11101
'm' :   3 |    4 | 0111
'n' :   3 |    4 | 1000
'o' :   4 |    4 | 1001
'p' :   2 |    4 | 1010
'r' :   3 |    4 | 1011
's' :   3 |    4 | 1100
't' :   1 |    6 | 111111
'u' :   2 |    5 | 11110

010011101111111010011011011011111101010111001000110110010000101111101110011100011101001000000101010011111010110001110101000001101010010111101010110011100110001101001100000011011001000110100101010001101000001110
total normal: 416
total compressed: 210
ratio: 50.48 %
```

FIGURE 3.2 – Output entrée au clavier

Ci-dessous, les arbres construits sont affichés.

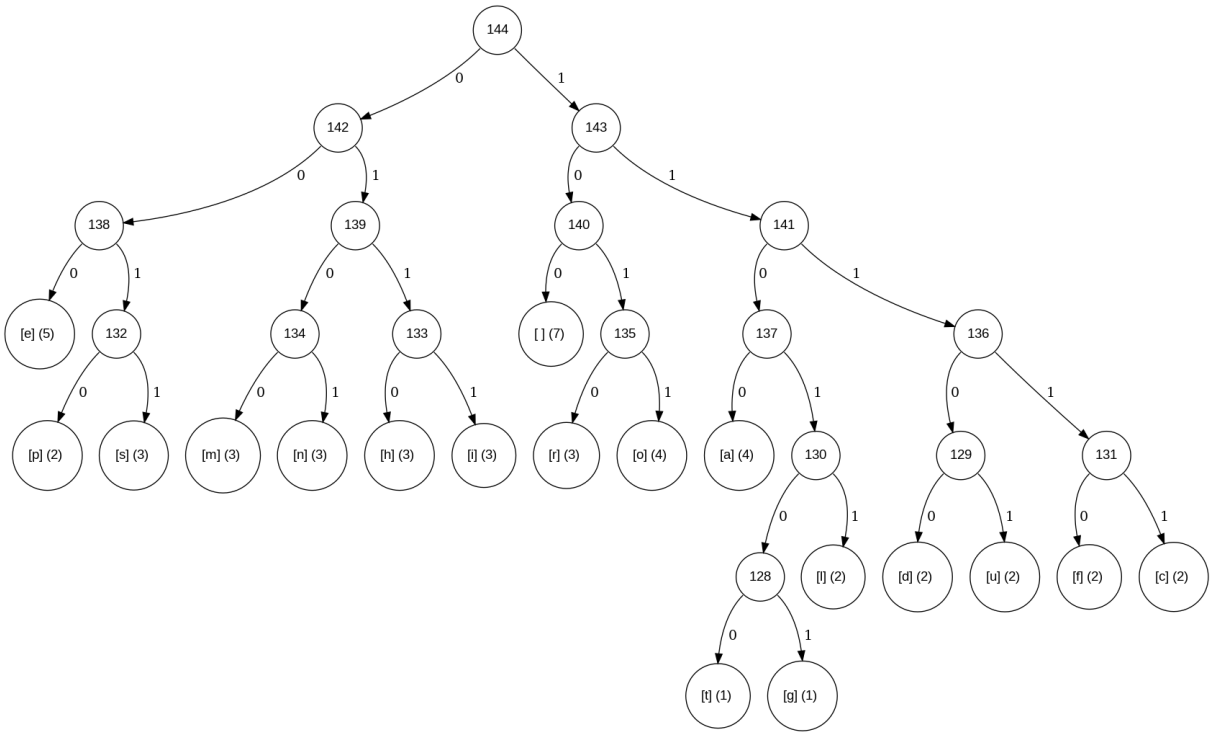

FIGURE 3.3 – Arbre Huffman entrée au clavier

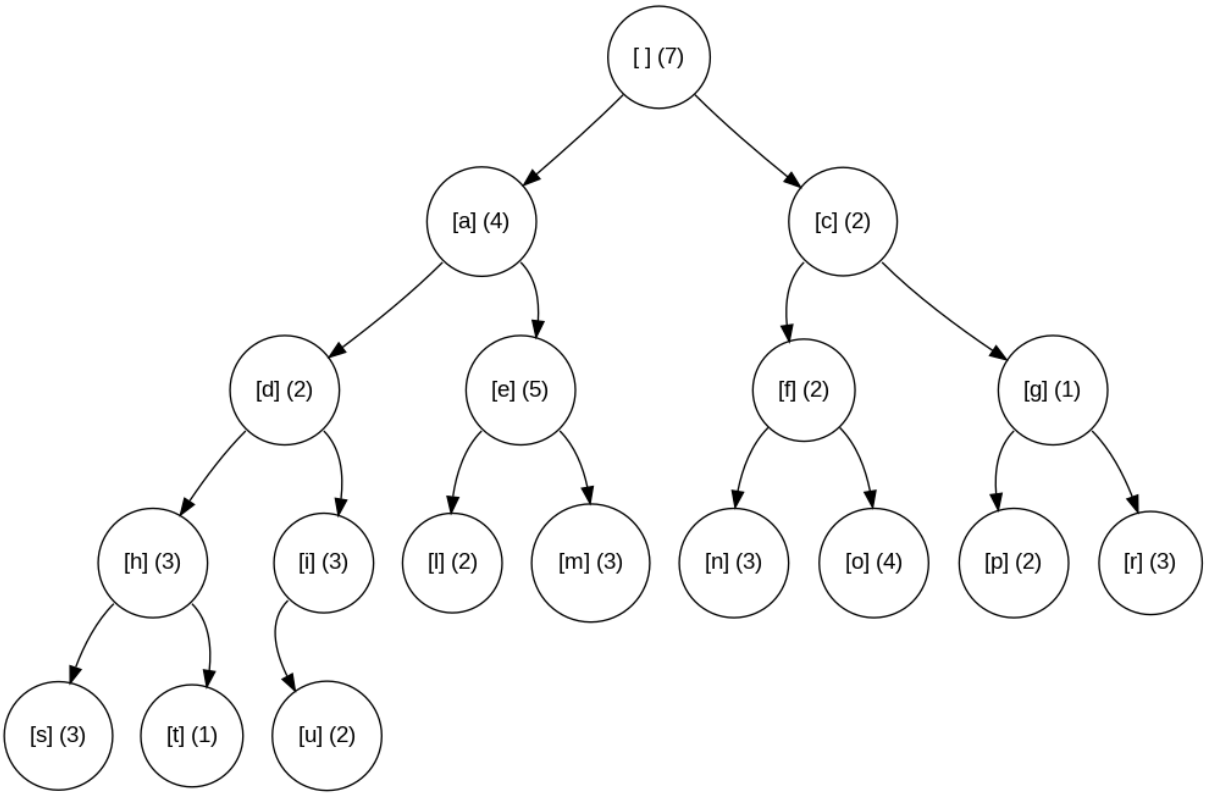

FIGURE 3.4 – Arbre minimier entrée au clavier

A noter que les résultats sont identiques à ceux fournis dans le texte du travail proposé, ce qui prouve le bon fonctionnement du programme.

Ensuite, les résultats obtenus à partir d'un texte de Shakespeare sont présentés.

```
$ ./codage < ./texts/shakespeare.txt

car : occ | long | bits
----+-----+------+--------
'       ' :   3 |   19 | 1111111111111111100
'
' :172420 |    5 | 01110
' :172420 |    5 | 01111
' ' :1041029 |    2 | 00
'!' :8431 |    9 | 111101110
'"' :  80 |   16 | 1111111111111100
'#' :   1 |   21 | 11111111111111111010
'$' :   2 |   21 | 11111111111111111011
'%' :   1 |   21 | 11111111111111111100
'&' :  65 |   16 | 1111111111111101
''' :13340 |    9 | 111101111
'(' : 318 |   14 | 11111111110110
')' : 318 |   14 | 11111111110111
'*' :  22 |   17 | 11111111111111110
',' :93032 |    6 | 110000
'-' :5508 |   10 | 1111110110
'.' :85754 |    6 | 110001
'/' :   6 |   19 | 1111111111111111101
'0' : 157 |   15 | 111111111111000
'1' : 350 |   14 | 11111111111000
'2' : 231 |   14 | 11111111111001
'3' : 165 |   15 | 111111111111001
'4' : 194 |   14 | 11111111111010
'5' : 113 |   15 | 111111111111010
'6' : 155 |   15 | 111111111111011
'7' :  84 |   16 | 1111111111111110
'8' : 148 |   15 | 111111111111100
'9' :  92 |   15 | 111111111111101
':' :4455 |   10 | 1111110111
';' :16404 |    9 | 111110000
'?' :11137 |    9 | 111110001
'A' :44480 |    7 | 1110000
'B' :13278 |    9 | 111110010
'C' :18747 |    8 | 11110000
'D' :13128 |    9 | 111110011
'E' :36114 |    7 | 1110001
'F' :11316 |    9 | 111110100
'G' :10425 |    9 | 111110101
'H' :16799 |    9 | 111110110
'I' :51812 |    7 | 1110010
'J' :1837 |   12 | 111111111010
'K' :5771 |   10 | 1111111000
'L' :21759 |    8 | 11110001
'M' :14504 |    9 | 111110111
```

FIGURE 3.5 – Output entrée texte p1.

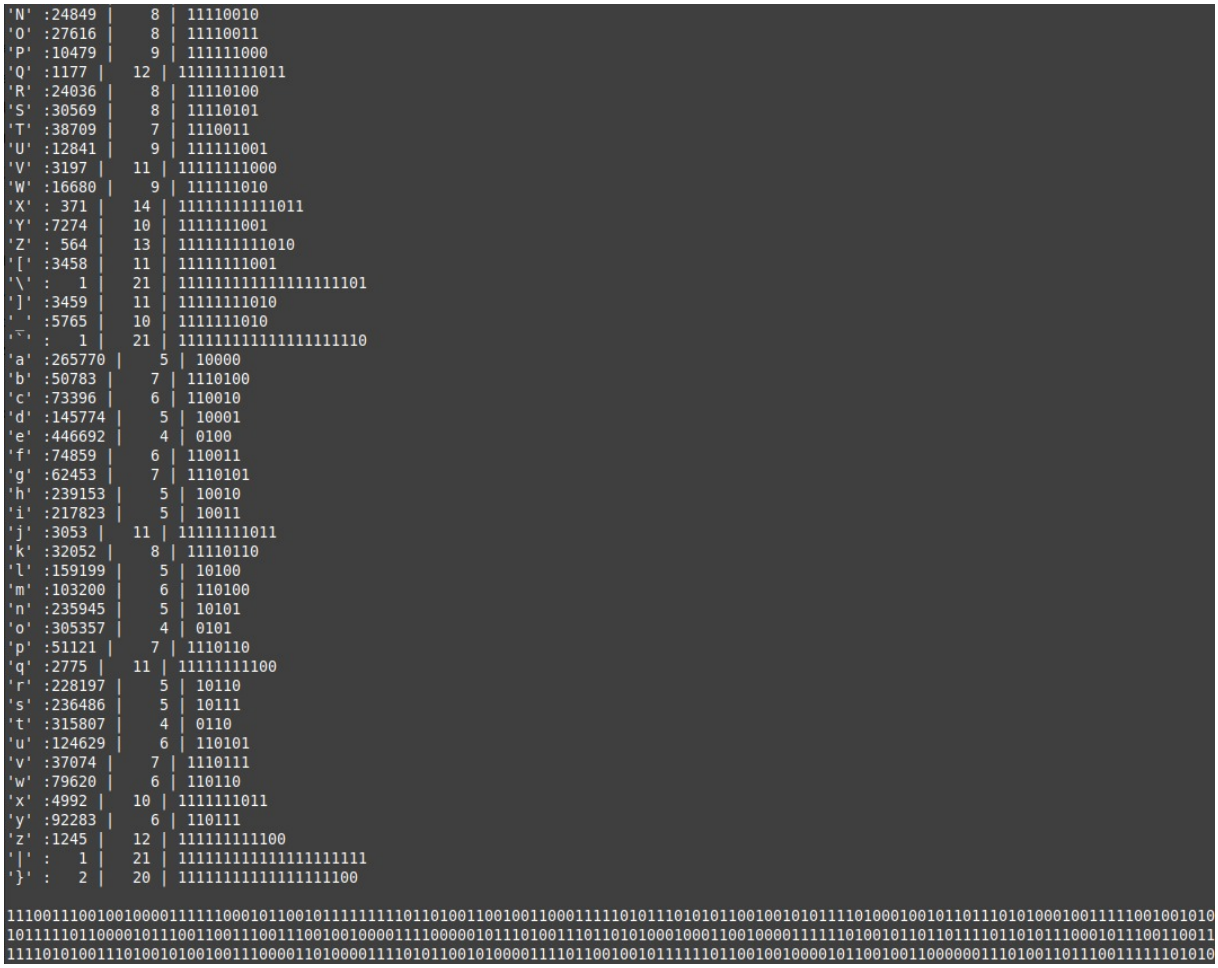

FIGURE 3.6 – Output entrée texte p2.

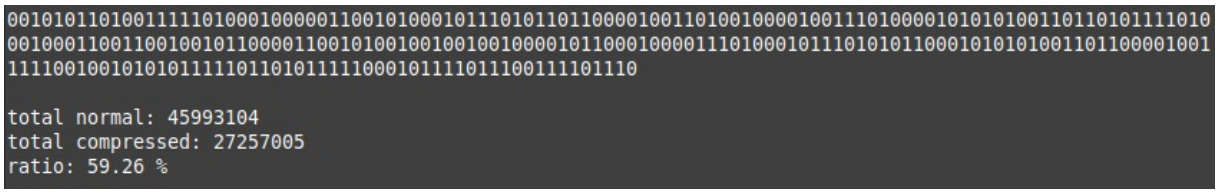

FIGURE 3.7 – Output entrée texte p3.

Il est intéressant d'observer la quantité de caractères qui sont réduits et compressés. Il est à noter que, bien évidemment, seule une partie de la représentation textuelle binaire a été présentée puisqu'il s'agit d'un fichier avec un très grand nombre de caractères.

Les arbres générés pour le fichier texte sont ensuite présentés.

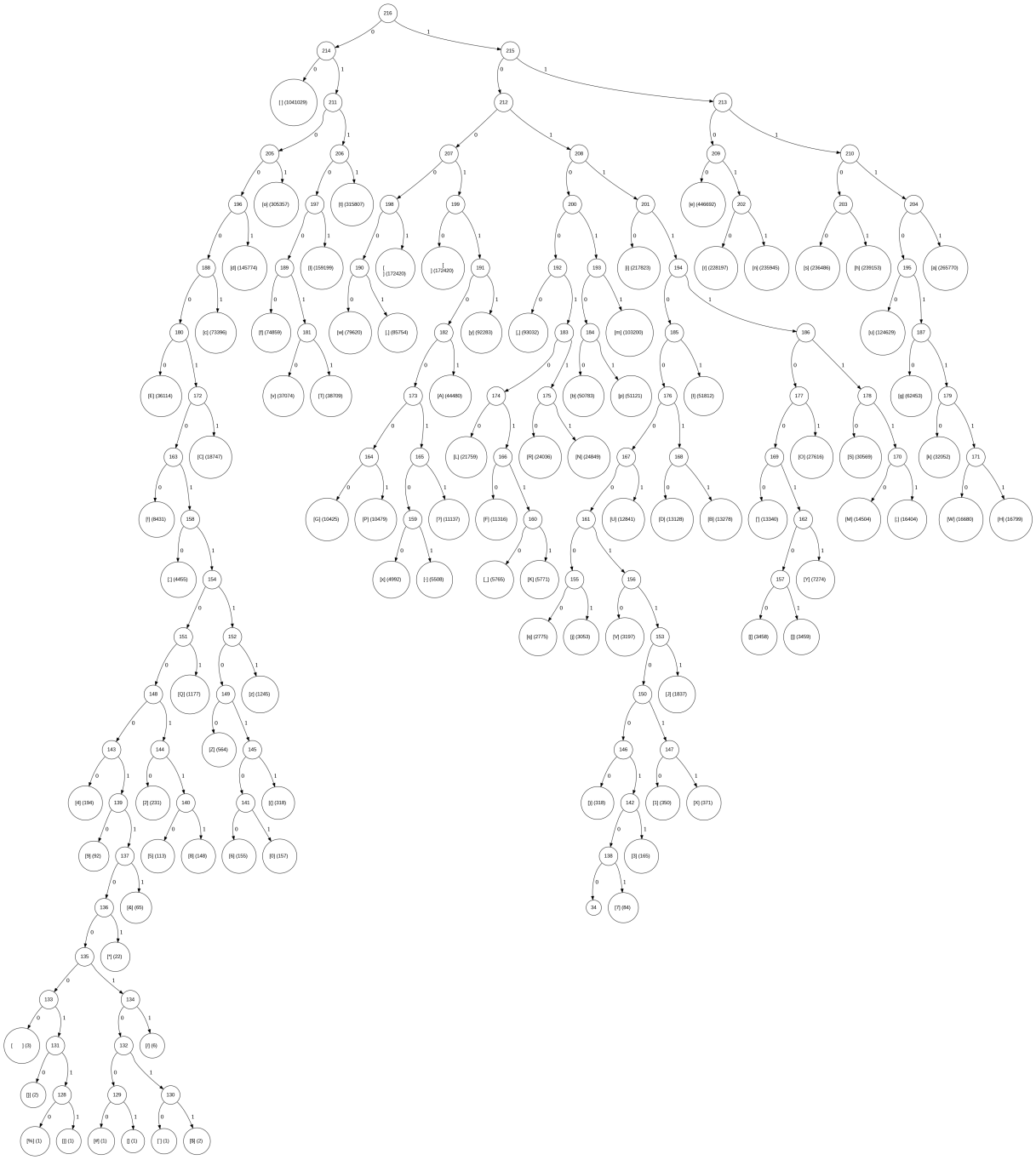

FIGURE 3.8 – Arbre Huffman entrée texte

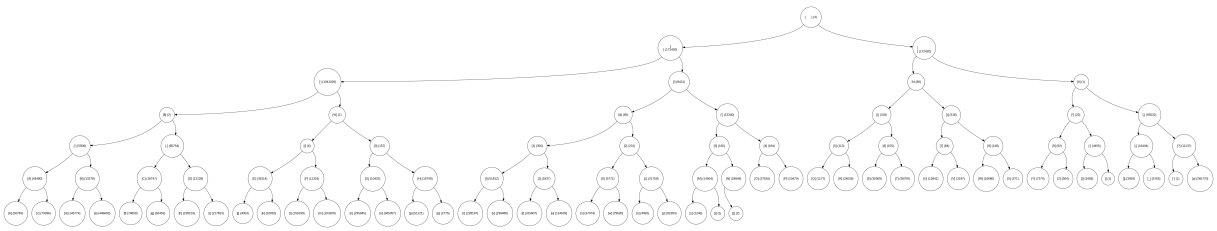

FIGURE 3.9 – Arbre minimier entrée texte

Notez qu'il s'agit d'arbres de taille significativement plus grande par rapport à ceux

présentés précédemment. C'était en fait prévu puisque le nombre de caractères est notamment plus élevé.

3.3 Programme 3 : huffman.exe

L'image ci-dessous montre la sortie de la compression d'un fichier texte.

FIGURE 3.10 – Output compression

Comme mentionné dans le travail de développement, la première ligne représente la quantité de caractères uniques trouvés dans le fichier texte. La seconde montre la correspondance entre le caractère et sa fréquence. Enfin, la représentation textuelle binaire du texte reçu. L'image ci-dessous est destinée à mieux faire comprendre le fonctionnement réel de cette représentation - puisqu'une correspondance est faite avec la table ASCII.

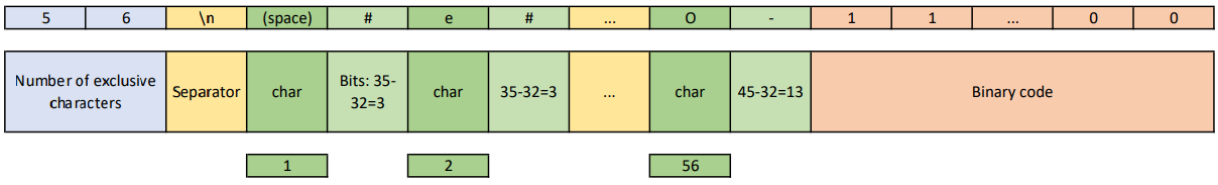

FIGURE 3.11 – Représentation de l'entête d'Huffman

4. Perspectives

L'une des limites observées dans le travail était le fait que nous ne travaillions pas avec un "vrai" binaire. Cela est devenu clair en comparant la taille occupée en mémoire par les fichiers compressés et non compressés. Il a été prouvé, d'une part, qu'il existe bien un taux de compression et donc une réduction de la quantité d'informations stockées. Par contre, l'espace occupé en mémoire augmente, ce qui n'est pas du tout intuitif. Ce problème pourrait être résolu si un binaire était vraiment utilisé dans son sens naturel, c'est-à-dire celui qui est lié à l'allocation de mémoire.

5. Références bibliographiques

[1] D.M. Ritchie et B.W. Kernighan. *The C programming language*. Bell Laboratories, 1988.

[2] David A. Huffman. « A Method for the Construction of Minimum-Redundancy Codes ». In : *Proceedings of the IRE* 40.9 (1952), p. 1098-1101. doi : 10.1109/JRPROC.1952.273898.