République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mouloud Mammeri, Tizi-Ouzou
Faculté de Génie Electrique et d'Informatique

Département Informatique.

MEMOIRE DE MASTER
En Informatique
Option : Conduite d'un projet informatique

Thème :

Etude comparative entre deux méthodes de compression VLC et Huffman

Réalisé par :
M^{elle} : AMEUR Nacima

Dirigé par :
M^{r} : SADOU.S

Année universitaire : 2010/2011.

Résumé :

Avec l'évolution des technologies de communication plusieurs domaine de recherche ont été créé afin d'assurer la fiabilité de cette transmission. La compression des données est l'un de ces domaines, c'est un monde vaste de recherche à cause de son importance dans le gain de l'espace de stockage et du temps de transmission.

Dans ce travail, nous avons comparé deux méthodes de compression, une qui existe déjà, qui a montré son efficacité dans ce domaine et elle est utilisée beaucoup. Cette méthode (algorithme) est celle de Huffman. L'autre méthode (algorithme) est une méthode proposée.

Les deux algorithmes sont des algorithmes de compression sans perte, basés sur le codage à longueur variable et aussi sur le codage statistique c'est-à-dire les fréquences d'apparition des symboles.

Cette comparaison est faite pour évaluer le temps et le taux de compression afin de constater laquelle parmi les deux méthodes est la meilleure.

Remerciements

Je tiens à remercier le Dieu tout puissant, qui m'a aidé et bénis durant la réalisation de mon travail.

Je tiens à exprimer ma profonde gratitude et mes sincères remerciements à mon promoteur, Monsieur SADOU qui m'a fait l'honneur de diriger ce travail et ses précieux conseils furent d'un apport considérable.

Aussi je tiens à lui reconnaître le temps précieux qu'il m'a consacré.

Que les membres du jury trouvent ici mes remerciements les plus vifs pour avoir accepter d'honorer par leurs jugements mon travail.

Mes sincères sentiments vont à tous ceux qui, de près ou de loin, ont contribué à la réalisation de ce projet.

Dédicaces

Je dédie ce modeste travail à :

Mes très chers parents;

Ma sœur AKILA et mon frère MOUMOUH ;

Mes très chers oncles et très chères tantes ;

Mes grands parents ;

Tous mes précieux amies et amis ;

Toute personne qui a contribué à la réalisation de ce travail ;

Toute personne que j'estime.

Nacima.

Sommaire

----------Chapitre-II- :Les méthodes VLC ----------

---------- Chapitre V : Implémentation et réalisation----------

---------- Conclusion générale ----------

Liste des figures.

Chapitre I : généralités.

Chapitre II : Les méthodes VLC.

Chapitre III : La méthode de Huffman.

Chapitre IV : Analyse et conception.

Chapitre V : Implémentation et évaluation.

Liste des tableaux.

Introduction générale

En quelques années seulement l'humanité a connu une gigantesque évolution au niveau des technologies surtout ce qui concerne les technologies de communication. Cette évolution est due à l'invention des supports de communication comme les téléphones mobiles et les ordinateurs et elle est due aussi à la découverte des réseaux.

L'information transmit dans le réseau peut être de n'importe quel type pourvu que nous puissions en donner une représentation numérique : textes, images, sons, vidéos par exemple. La transmission de ces types de données est omniprésente dans la technologie, et spécialement dans les télécommunications. Il est donc nécessaire de s'appuyer sur de bonnes bases théoriques pour que cette transmission soit fiable, ce dernier terme se voyant donner plusieurs significations qui sont les objectifs de plusieurs recherches dans le domaine mathématique et celui de l'informatique, souvent développées indépendamment les uns des autres.

La théorie des codes traite la forme de l'information elle-même quand elle doit être transmise ou stockée. Pour transmettre l'information, les canaux de transmission peuvent être de tous types (réseaux de câbles ou d'ondes). L'information doit être reçue par son destinataire dans son intégralité, en sécurité et le plus rapidement possible.

Réduire la redondance de données qui se trouve dans une information est ce que nous appelons la compression, c'est-à-dire rendre le message le plus court possible, et tous ça afin d'optimiser de l'espace de stockage, et plus encore augmenter la rapidité de transmission car les plus grands soucis des développeurs sont : l'espace mémoire et le temps.

La compression de données a fait un vaste domaine de recherche et elle continu à être un objectif de plusieurs études voyant son importance. Dans ce but plusieurs algorithmes ont été développés. Nous en trouvant plusieurs classification de ces algorithmes, mais la plus importante est celle qui décompose les algorithmes en :

- Les algorithmes de compression sans perte de données : c'est-à-dire lors de la décompression du fichier compressé nous obtenons le même fichier d'origine.

- Les algorithmes de compression avec perte de données : contrairement au premier type ici nous obtenons lors de la décompression un fichier qui n'est pas le même que celui d'origine mais cela n'influence pas beaucoup sur son contenu.

Dans ce projet, nous avons fait une étude comparative entre deux algorithmes de compression sans perte de données, le premier est un algorithme proposé qui se base sur le codage à longueur variable et l'autre existant déjà qui est l'algorithme de Huffman.

Pour mener à bien notre étude, nous avons décomposé le travail en cinq chapitres qui sont :

- **Chapitre I :** dans ce chapitre nous avons présenté des généralités, des définitions ainsi que des concepts de base de la compression des données.

- **Chapitre II :** ce chapitre est consacré pour la présentation de la compression à base des codes VLC (Variable Length Coding) ainsi que quelques algorithmes qui utilisent ce type de codage.

- **Chapitre III :** toute au long de ce chapitre nous avons présenté l'algorithme de Huffman utilisé dans notre étude, son principe, son fonctionnement, ses variantes, ses avantages et ses inconvénients.

- **Chapitre IV :** dans ce chapitre nous avons présenté le principe et le fonctionnement de la méthode VLC proposée. En plus nous avons fait l'analyse et la conception des deux méthodes et nous avons fait sortir les principaux modules et algorithmes à développés.

- **Chapitre V :** ce chapitre présente l'environnement de développement et les logiciels utilisés dans nos applications ainsi que les résultats de la comparaison de ces dernières.

Enfin nous avons terminé par une conclusion.

Chapitre I

Généralités

I. Introduction :

L'apparition du CD audio au début des années 80 est à l'origine de ce que nous appelons communément aujourd'hui le "multimédia". Sur une petite surface de 12 cm de diamètre, nous pouvions stocker d'un coup près de 650 Mo de données alors que les disques durs de l'époque atteignaient péniblement des capacités de 500 Mo. Très vite, nous avons pris conscience que le CD pouvait servir non seulement à stocker du son, mais aussi toutes sortes de données informatiques et notamment des images et de la vidéo.

Lors de la transmission des données (flux vidéo, sonore…) sur un réseau, la bande passante allouée pour le transfert est très limitée (quelques centaines de Ko/s pour la vidéo à la demande, par exemple), ceci nécessite des supports de stockage alliant une puissance et performance pour satisfaire la forte demande de l'espace de stockage, C'est pour ces raisons que la compression de données est presque toujours utilisée, pour réduire autant que possible la quantité de données à stocker ou transmettre.

II. Définitions :

II.1. Le codage : [1]

La notion de code au sens large veut dire une règle permettant de convertir une information sous une forme différente de sa représentation initiale, c'est-à-dire changer la représentation de l'information.

Ainsi, nous pouvons diviser les « codes » en plusieurs grandes familles :

- les codes de communication (Morse, Baudot, Popham,...).
- les codes de représentation (UTF, ASCII, Base64,. . .).
- les codes de protection (Cryptographie ancienne, Stéganographie).
- les codes de compression ("Théorie du codage").
- les codes d'identification (IBAN, EAN, Code-barres,. . .).

II.2. La compression :[25]

La compression de données (codage de source) est peut être considérée comme une transformation qui fait correspondre de façon biunivoque un message comprimé Y=(y1 ,y2,….yn} à un message source S= {s1, s2,…….sm}. Cette transformation est appelée codage de source (Source Coding). Nous pouvons dire aussi que la compression des données est une opération de codage de source qui tire parti des particularités de la source réelle pour éliminer les informations redondantes La décompression est l'opération inverse de la compression.

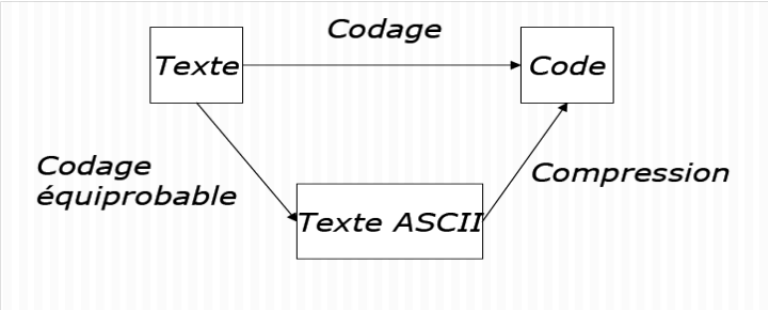


Figure1.1 Du codage à la compression.

II.3. La données brutes : [2]

Les termes données brutes (raw data) et données non codées (unencoded data) désignent les données avant qu'elles ne soient compressées et les termes données codées (encoded data) ou données compressées (compressed data) désignent les données après qu'elles aient été compressées.

II.4.Le taux de compression :[2]

Le terme taux de compression (compression ratio) est utilisé pour se référer au rapport entre la taille des données non compressées sur la taille des données compressées. Si nous pouvons compresser un fichier 10 fois, nous parlerons alors d'un taux de 10:1 ce qui signifie très logiquement que sa taille a été divisée par 10. Ce sera également un critère d'efficacité entre différents algorithmes : Un algorithme ayant un taux de 10:1 sera 5 fois plus efficace qu'un algorithme n'ayant qu'un taux de compression de 2:1.

$$taux\ de\ compression = \frac{taille\ \ des\ données\ non\ compressées}{taille\ des\ données\ compressées}$$

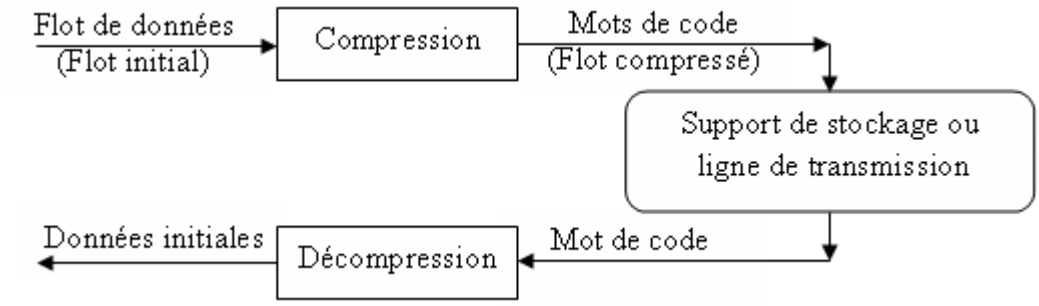


Figure1.2 :La compression et la décompression.

II.5. Théorie de l'information :

II.5.1. Quantité d'information – Incertitude :[3]

Nous définissons la notion de quantité d'information (ou incertitude) d'une observation x de la variable aléatoire X par : $I_X(x) = -\log(P(X = x))$

Comme nous pouvons s'y attendre, l'incertitude d'un événement certain est nulle, tandis que celle d'un événement de probabilité nulle est infinie. Entre ces deux cas limites, la valeur de la mesure obtenue dépend de la base du logarithme, qui détermine l'unité de mesure et correspond à l'alphabet utilisé.

En règle générale, nous mesurons la quantité d'information en bits. Nous utilisons donc le logarithme base 2. En effet, pour décrire un événement de probabilité 1/2, nous avons besoin de :
- $\log_2(1/2) = 1$ symbole binaire.

L'unité de la quantité d'information est le Shannon.

II.5.2.L'entropie :[3]

Nous appelons entropie de la variable aléatoire X l'espérance de la quantité d'information des observations de X :

$$H(X) = E_X\{I_X\} = E_X\{-\log(P(X))\}$$

Exemple :

Considérons une source d'information S qui émette des messages constitués de symboles de l'alphabet $\Lambda = \{\lambda_1, ..., \lambda n\}$, et de loi de probabilité P : $\{p1 = P(\lambda_1), ..., p_n = P(\lambda n)\}$. Son entropie est donnée par la formule1 :

$$H(s) = -\sum_{k=1}^{n} p_k - \log(p_k)$$

Où :

- $0 \leq H \leq \text{Log}_2(n)$
- H=0 lorsque P(Xi)=1
- $H=\text{Log}_2(n)$ lorsque P(Xi)=1/n, i=1..n

Shannon a de plus montré, d'une part, que l'entropie d'une source correspond à la longueur moyenne minimale de description de ses messages, et d'autre part qu'en choisissant convenablement son code, il était possible d'approcher cette valeur minimale d'aussi prés que l'on veut.

II.5.3. Longueur moyenne d'un code :[1]

Soit un codage des messages à partir d'un alphabet de d caractères tel qu'il existe une correspondance non ambiguë entre chaque message et son code. Si le message i est représenté par une séquence de li caractères, la longueur moyenne d'un code est donnée par la formule suivante :

$$\bar{L} = \sum_{i=1}^{N} pi\ li$$

II.5.4. La redondance : [3]

La compression de données se fonde sur la détection et l'élimination des redondances dans un flot de données. Sachant que l'information informatique (texte, audio, image, etc.) est logiquement représentée par une combinaison de 256 octets, appelée alphabet. Ceci implique que chaque flot de données de taille supérieure à celle de l'alphabet (nombre de symboles) présente nécessairement une redondance.

Exemple: Soient « A » l'alphabet et « F » le flot de données.

Si A= {0,1}, sa taille est de 2 symboles. Un flot de données F=1001…10, présente une redondance de symboles 0 et 1 dès que la taille de F dépasse 2. De ce fait, les flots de données de grande taille ne sont que le résultat de la redondance des symboles de l'alphabet qui est un ensemble borné.

III. La classification des algorithmes de compression :

III.1. Compression symétrique / asymétrique :[4]

- ***Compression symétrique :***

Dans le cas de la compression symétrique, la même méthode est utilisée pour compresser et décompresser l'information, il faut donc la même quantité de travail pour chacune de ces opérations. C'est ce type de compression qui est généralement utilisée dans les transmissions de données.

- ***Compression asymétrique :***

La compression asymétrique demande plus de travail pour l'une des deux opérations, la plupart des algorithmes requiert plus de temps de traitement pour la compression que pour la décompression. Des algorithmes plus rapides en compression qu'en décompression peuvent être nécessaires lorsque nous archivons des données auxquelles nous accédons peu souvent.

III.2. Compression physique / logique :[4]

Nous considérons généralement la compression comme un algorithme capable de comprimer des données dans un minimum de place (compression physique), mais nous pouvons également adopter une autre approche et considérer qu'en premier lieu un algorithme de compression a pour but de recoder les données dans une représentation différente plus compacte contenant la même information (compression logique).

La distinction entre compression physique et logique se base sur la façon dont les données sont compressées ou plus précisément comment est-ce que les données sont réarrangées.

- ***Compression physique :***

La compression physique est exécutée exclusivement sur les informations contenues dans les données. Cette méthode produit typiquement des résultats incompréhensibles qui apparemment n'ont aucun sens. Le résultat d'un bloc de données compressées est plus petit que l'original car l'algorithme de compression physique a retiré la redondance qui existait entre les données elles-mêmes.

- ***Compression logique :***

La compression logique est accomplie à travers le processus de substitution logique qui consiste à remplacer un symbole alphabétique, numérique ou binaire en un autre. Changer "United State of America" en "USA" est un bon exemple de substitution logique car "USA" est dérivé directement de l'information contenue dans la chaîne "United State of America" et garde la même signification.

III.3. Compression statistique / numérique :[4]

Nous pouvons encore distinguer les algorithmes qui travaillent au niveau statistique et ceux qui opèrent au niveau numérique.

- ***Compression statistique :*** dans ce cas la valeur des motifs ne compte pas. Ce sont les probabilités qui comptent, et le résultat est inchangé par substitution des motifs.

- ***Compression numérique :*** dans ce cas les valeurs des motifs influents sur la compression (par exemple JPEG), et les substitutions sont interdites.

III.4. Codage adaptatif, semi-adaptatif et non-adaptatif :[2]

-1- Codage non-adaptatif :

Certains codeurs statiques sont conçus pour compresser seulement des types spécifiques de données. Ces codages non-adaptatifs contiennent un dictionnaire statique de chaînes de caractères prédéfinies qui sont connues comme apparaissant à de grandes fréquences dans les données à encoder. Par exemple, un codeur non-adaptatif conçu spécifiquement pour compresser la langue française contiendra un dictionnaire avec des chaînes de caractères telles que "et", "mais", "de", "le", car ces chaînes apparaissent très fréquemment dans les textes en français.

-2- Codage adaptatif :

Un codeur adaptatif, à l'inverse n'intégrera pas de données relatives à la fréquence d'apparitions des données à compresser. Des compresseurs adaptatifs déterminent la dépendance des données en construisant leur dictionnaire à la volée. Ils n'ont pas de listes prédéfinies de chaînes de caractères par exemples mais les construisent dynamiquement à l'encodage.

La compression adaptative est capable de s'adapter à n'importe quelles données d'entrées et de retourner une sortie avec le taux de compression le meilleur possible. C'est une des principales différences avec les compressions non-adaptatives qui sont capable d'avoir des codages efficaces uniquement avec un type de données d'entrées très restreint pour lequel ils ont étés conçus.

-3- Codage semi-adaptatif :

Un mélange de ces deux méthodes d'encodage à l'aide de dictionnaires est la méthode d'encodage semi-adaptative. Un encodage semi-adaptatif fait un premier passage sur les données pour construire le dictionnaire et un second passage pour effectuer l'encodage. En utilisant cette méthode, un dictionnaire optimal est construit avant qu'un quelconque encodage soit effectué.

III.5.La compression sans / avec perte :[2]

III.5.1. Compression sans perte :

La compression est dite sans perte lorsqu'il n'y a aucune perte de données sur l'information d'origine. Il y a autant d'information après la compression qu'avant, elle est seulement réécrite d'une manière plus concise (c'est par exemple le cas de la compression « gzip » pour n'importe quel type de données ou bien le format « png » pour des images synthétiques destinées au Web). La compression sans perte est dite aussi compactage, non destructible, réversible, ou conservative.

L'information à compresser est vue comme la sortie d'une source de symboles qui produit des textes finis selon certaines règles. Le but est de réduire la taille moyenne des textes obtenus

après la compression tout en ayant la possibilité de retrouver exactement le message d'origine. Il n'existe pas de technique de compression de données sans perte universelle qui pourrait compresser n'importe quel fichier.

Il existe 3 types d'algorithmes de compression sans perte :

-1- Codage statistique :

Le but est de :

- Réduire le nombre de bits utilisés pour le codage des caractères fréquents.
- Augmenter ce nombre pour des caractères plus rares.

Exemple :

Certaines informations sont plus souvent présentes que d'autres dans les données que nous voulons compresser. Dans un fichier HTML par exemple, nous trouverons beaucoup de signes < , /, et >. On va chercher à coder les données se répétant souvent sur moins de bits, et les données moins fréquentes sur plus de bits. C'est ce principe qui est utilisé pour les algorithmes de Lempel-Ziv, Zip, Huffman …).

-2- Substitution de séquences :

Comprime les séquences de caractères identiques.

-3- Utilisation d'un dictionnaire :

Le but est de :

- Réduire le nombre de bits utilisés pour le codage des mots fréquents.
- Augmenter ce nombre pour des mots plus rares.

Le taux de compression des algorithmes sans perte est en moyenne de l'ordre de 40% pour des données de type texte. Par contre, ce taux est insuffisant pour les données de type multimédia. Il faut donc utiliser un nouveau type de compression pour résoudre se problème : la compression avec perte.

Les formats de fichier de compression sans perte sont connus grâce à l'extension ajoutée à la fin du nom de fichier « nomdefichier.zip » par exemple. Les formats les plus courants sont : 7z , ace, arc, arj, gzip, rar, zip, zoo …etc

III.5.2. Compression avec perte :

Son principe est basé sur l'étude précise de l'œil et de l'oreille humaine. Les signaux audio et vidéo contiennent une part importante de données que l'œil et l'oreille ne peuvent pas percevoir et une part importante de données redondantes.

Les objectifs de la compression avec pertes sont d'éliminer les données non pertinentes pour ne transmettre que ce qui est perceptible et, comme pour la compression sans perte, d'éliminer l'information redondante.

Ce type de compression engendre une dégradation indiscernable à l'œil (ou à l'oreille) ou suffisamment faible, en contrepartie d'un taux de compression très élevé.

Il existe des algorithmes de compression consacrés à des usages particuliers, dont en voici 3 :

- Compression du son (Audio MPEG, ADCPM ...).
- Compression des images fixes (JPEG,...).
- Compression des images animées (MPEG, ...).

III.5.3. Compression presque sans perte :

Les méthodes de compression sans perte significative sont un sous-ensemble des méthodes de compression avec perte, parfois distinguées de ces dernières. La compression sans perte significative peut être vue comme un intermédiaire entre la compression conservative et la compression non conservative, dans le sens où elle permet de conserver toute la signification des données d'origine, tout en éliminant une partie de leur information.

Dans le domaine de la compression d'image, la distinction est faite entre la compression sans perte et la compression sans perte significative. Une image compressée presque sans perte peut être décompressée pour obtenir les pixels de sa version non-compressée à l'identique. Elle ne peut en revanche pas être décompressée pour obtenir sa version non compressée intégralement à l'identique.

IV. Quelques techniques de compression :

IV.1. Quelques techniques de compression sans perte :

IV.1.1. L'algorithme de Huffman :[25]

Le codage de Huffman est un algorithme de compression de données sans perte. Le codage de Huffman utilise un code à longueur variable pour représenter un symbole de la source (par exemple un caractère dans un fichier). Le code est déterminé à partir d'une estimation des probabilités d'apparition des symboles de source, un code court étant associé aux symboles de source les plus fréquents. Les codes de Huffman sont des codes optimaux, au sens de la plus courte longueur.

Le principe du codage de Huffman repose sur la création d'un arbre composé de nœuds. Nous recherchons tout d'abord le nombre d'occurrences de chaque caractère. Chaque caractère constitue une des feuilles de l'arbre à laquelle on associe un poids valant son nombre d'occurrences. Puis l'arbre est créé suivant un principe simple : on associe à chaque fois les deux nœuds de plus faibles poids pour donner un nœud dont le poids équivaut à la somme des poids de ses fils jusqu'à n'en avoir plus qu'un, la racine. Nous associons ensuite par exemple le code 0 à la branche de gauche et le code 1 à la branche de droite. Nous consacrerons par la suite tout un chapitre pour détailler cet algorithme.

IV.1.2.L'algorithme RLE (Run-length Encoding):[2]

RLE est un algorithme de compression de données qui a été créé pour compresser n'importe quel type de données sans tenir compte de l'information qu'elle contient. Toutefois le contenu des données va affecter le taux de compression qu'il pourra atteindre. Bien que la plupart des algorithmes RLE ne puissent pas atteindre les forts taux de compression d'autres méthodes de compression plus avancées, RLE est à la fois simple à implémenter et rapide d'exécution ce qui fait de lui une bonne alternative entre utilisé un algorithme de compression plus complexe ou laisser l'image sans compression.

Le secret de la méthode RLE consiste à identifier et supprimer des redondances d'informations en les codant sous une forme plus compacte. RLE s'emploie à réduire la taille physique d'une répétition de chaîne de caractère. Cette chaîne répétée est appelée un passage (*run*) et est typiquement codée avec 2 bytes. Le premier byte représente le nombre de caractères dans le passage et est appelé le compteur de passage (*run count*). Il peut prendre une valeur comprise entre 0h et 128h ou 256h. Le second byte est la valeur du caractère dans le passage qui peut prendre la valeur 0h à FFh. Ce dernier byte est appelé la valeur du passage (*run value*).

Non compressé, le passage suivant comprenant 15 caractères devrait normalement prendre 15 bytes à stocker.

AAAAAAbbbXXXXXt

En utilisant le codage RLE, cette chaîne pourra être compressée en 4 paquets de 2 bytes.

6A3b5X1t

Le "6A3b5X1t" généré pour représenter la chaîne de caractère est appelé un paquet RLE (*RLE packet*). Ainsi, après le codage, la chaîne de 15 bytes de départ prendra seulement 8 bytes de données pour représenter la chaîne. Dans ce cas, RLE permet d'atteindre un taux de compression de 2:1.

De longs passages sont rares dans certains types de données. Par exemple le texte ASCII (comme celui contenu sur cette page) contient rarement de longs passages mais plutôt des passages de 1 ou 2 caractères identiques, rarement plus. Or il s'avère que pour coder un passage en RLE, nous avons besoin de 2 caractères, par conséquent, il s'avère que si nous codons des chaînes de caractères en RLE constituées de passages composés uniquement de 2 caractères nous n'aurons aucun gain et la taille du fichier après compression sera exactement la même. Pire, si les passages sont composés uniquement de 1 caractère, la taille du fichier après compression sera le double de celle de départ.

Si nous regardons ce qui se passe sur l'exemple suivant constitué principalement de caractères uniques :

Cannibalisation

Après le codage RLE :

1C1a2n1i1b1a1l1i1s1a1t1i1o1n

Nous pouvons déduire de tout cela que le codage RLE est simple et efficace mais que l'efficacité de la compression dépend fortement du type de données à encoder. Une image en noir et blanc constituée d'une grande partie de blanc s'encodera très facilement en raison de la grande quantité contiguë de données identiques. A l'inverse, une image photographique constituée de beaucoup de couleurs différentes s'encodera très difficilement.

IV.1.3. L'algorithme LZW (Lempel – Ziv – Welch) :[2]

Un des algorithmes de compression les plus utilisé en matière de compression d'informations en informatique est le LZW. Cette méthode d'encodage sans perte est trouvée dans plusieurs formats tels que GIF ou TIFF et fait également partie du standard de compression de modem V.42bis et de PostScript niveau 2.

A la fin des années septante, deux chercheurs israéliens, Jacob Ziv et Abraham Lempel, mettent au point une technique de compression encore plus sophistiquée, appelée compression LZ. Ils inauguraient la compression moderne, exploitant un dictionnaire mobile sur le même principe que la compression arithmétique. C'était, depuis Huffman, la première fois que l'on mettait sérieusement en doute l'efficacité des méthodes statistiques. Le principe est assez simple à comprendre: pour coder un mot, comme "anticonstitutionnellement", nous pouvons effectivement le transmettre, ou ne transmettre que sa position dans un dictionnaire contenant tous les mots possibles de la langue française. Dans le deuxième cas, il n'y a qu'un entier à transmettre. L'inconvénient est qu'il faut disposer d'un dictionnaire parfait. L'idée maîtresse de LZ et des algorithmes à dictionnaire est de créer ce dictionnaire pendant le processus de compression.

Deux procédés en dérivent, connus sous les noms de LZ77 et LZ78, établis d'après leurs années de création. Ils reposent sur l'enregistrement de séquences de caractères dans une sorte de dictionnaires de "phrases", construit à partir des données en entrée, au fur et à mesure de la lecture du fichier à compacter. Le texte qui reste à compresser est comparé au contenu du dictionnaire. Dès qu'il rencontre une nouvelle séquence, le moteur de compression vérifie sa présence dans le dictionnaire. Si elle ne s'y trouve pas, il l'ajoute dans le dictionnaire et place en sortie un symbole marqueur qui en identifie l'adresse dans une table réservée à cet usage. Si la table a déjà été enregistrée, le compresseur se contente de mettre la table à jour, en y reportant l'identifiant. LZ77 est généralement trouvé dans la compression de texte et de programmes d'archivage tels que compress, zoo, lha, pkzip, et arj. LZ78 quant à lui est plus facilement utilisé pour compresser des données binaires telles que dans des bitmaps.

IV.1.4. L'algorithme CCITT pour fax (International Telegraph and Telephone Consultative Committee):[2]

Le CCITT est une organisation de standardisation qui a développé une série de protocoles de transmissions pour la transmission d'images et de données en noir et blanc sur des lignes téléphoniques (fax). Ces protocoles sont officiellement connus en tant que standard CCITT T.4 et T.6 mais y sont plus souvent référé comme standard de compression CCITT Groupe 3 et Groupe 4 respectivement.

Les codes des groupes 3 et 4 sont des algorithmes de compressions spécifiquement désignés pour le codage d'images en noir et blanc. Tous les fax modernes supportent la norme de compression Groupe 3. Elle est rapide, a un bon taux de compression et contient des informations de corrections d'erreurs.

Le groupe 4 est une forme de compression plus efficace capable d'encoder un document à la même vitesse que le groupe 3 mais en prenant 2 fois moins de place malgré le fait qu'il soit un tout petit peu plus difficile à implémenter. Il à été conçu pour fonctionner sur un réseau local c'est pourquoi il ne contient pas de code de détection d'erreurs.

Les algorithmes du CCITT sont non adaptatifs ce qui signifie qu'ils n'adaptent pas leurs algorithmes pour encoder chaque bitmap avec la meilleure efficacité possible. Les algorithmes CCITT sont des algorithmes statiques en référence à l'utilisation d'une table d'encodage de taille fixe pour l'ensemble d'un fichier. Le calcul des fréquences peut s'effectuer dynamiquement au fur et à mesure du traitement du fichier. Au démarrage du traitement toutes les probabilités seront de valeurs égales mais en même temps qu'un caractère sera encodé ou décrypté, sa probabilité d'apparition sera incrémentée.

De plus en pratique, cette méthode atteint des taux de compression supérieurs puisque dans un fichier, les probabilités d'occurrences évoluent sans cesse et qu'un modèle dynamique peut s'adapter aux variations de fréquences. Si la compression dynamique est un peu plus délicate à programmer que la compression statique, elle offre néanmoins nombre d'avantages.

VI.1.5. La compression arithmétique :[2]

La technique d'encodage arithmétique due à *P. Elsa* n'a pas, contrairement à celle de Huffman, pour restriction de ne devoir traduire les probabilités que par des nombres entiers de bits. Elle peut encore réduire l'espace occupé en encodant un caractère sur une fraction de bit. Pratiquement, les différences constatées entre le résultat atteint par Huffman et celui de code arithmétique restent assez minces.

Ce genre de cas se rencontre en particulier quand les probabilités d'apparition des divers symboles sont très déséquilibrées; imaginons en effet une source ne produisant que deux symboles, l'un avec une probabilité de 98%, l'autre avec la probabilité complémentaire: un codage selon Huffman ne peut attribuer qu'un bit à chaque symbole, indépendamment de la quantité d'information contenue dans le symbole incriminé, simplement parce qu'il n'est pas possible de transmettre des fractions de bit. C'est dans ce genre de cas que le codage arithmétique se montre plus favorable que les codages statistiques.

L'encodage arithmétique traite l'ensemble d'un message comme une seule entité. Il fonctionne par la représentation d'un nombre par un intervalle de nombres réels compris entre 0 et 1. A mesure que le message s'allonge, l'intervalle nécessaire pour le représenter diminue, et le nombre de bits qui servent à préciser cet intervalle s'accroît. Les symboles successifs du message réduisent cet intervalle en concordance avec la probabilité d'apparition du symbole.

VI.1.6. L'algorithme de Shanon- Fano :[2]

Dans la méthode connue sous le nom de "Shannon- Fano", l'idée est de répartir les symboles en deux groupes de valeur à peu près équivalente, cette valeur étant la somme, dans chaque groupe, des probabilités d'apparition des symboles qu'il contient. Le groupe de gauche est appelé 0, celui de droite 1 (ce choix est arbitraire).

Les groupes sont à nouveau subdivisés et nommés 0 ou 1 jusqu'à ce que la subdivision ne contienne plus qu'un symbole. L'arbre binaire ainsi obtenu est formé de segments ou branches et de feuilles. Chaque branche représente un bit d'informations (0 ou 1), Chaque feuille contient un caractère simple. Pour déterminer le code numérique d'un caractère donné, il faut partir du sommet de l'arbre et suivre les branches jusqu'à atteindre la feuille qui le représente. Les caractères les plus fréquents se trouvent le plus près du sommet et requièrent donc moins de bits dans leurs transcriptions compressées.

Un exemple de codage selon Shannon-Fano :

Nous allons effectuer la compression de notre message-témoin suivant « BANANES ET ANANAS » en utilisant l'algorithme de Shannon-Fano. Nous remarquons que, dans l'arbre de codage résultant, nous inscrivons un "0" en regard de la branche de gauche, et un "1" en regard de la branche de droite. Ce codage est sans effet sur l'efficacité du résultat, et est donc arbitraire. La subdivision définit deux symboles (qui sont en fait des groupes de symboles), "AN" et ES<space>BT. Cette subdivision est ainsi faite pour que les fréquences des deux subdivisions soient aussi équilibrées que possible. Voici la figure qui montre ces subdivisions :

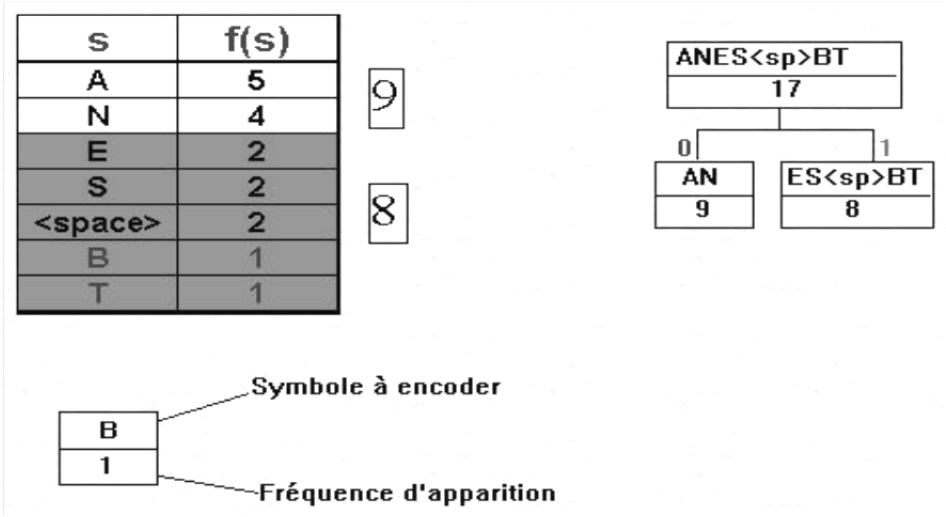
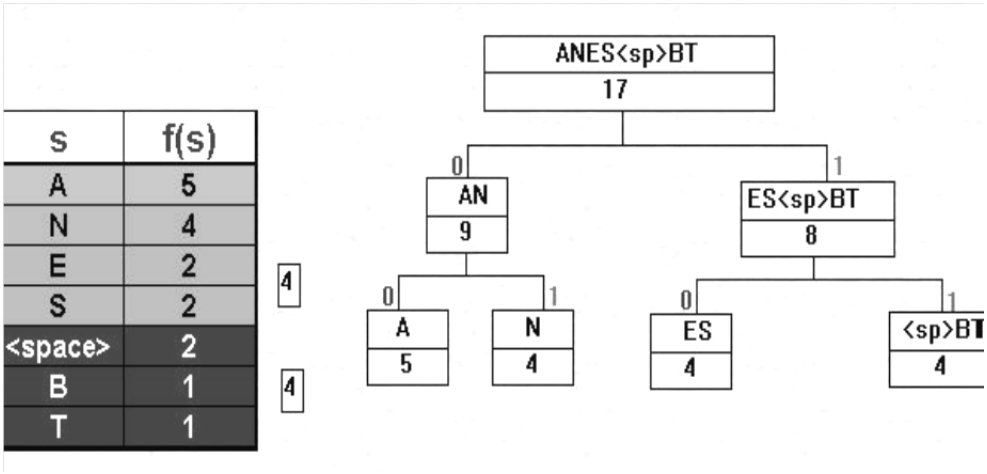


Figure1.3 : Les subdivisions de l'algorithme de Shanon-Fano.

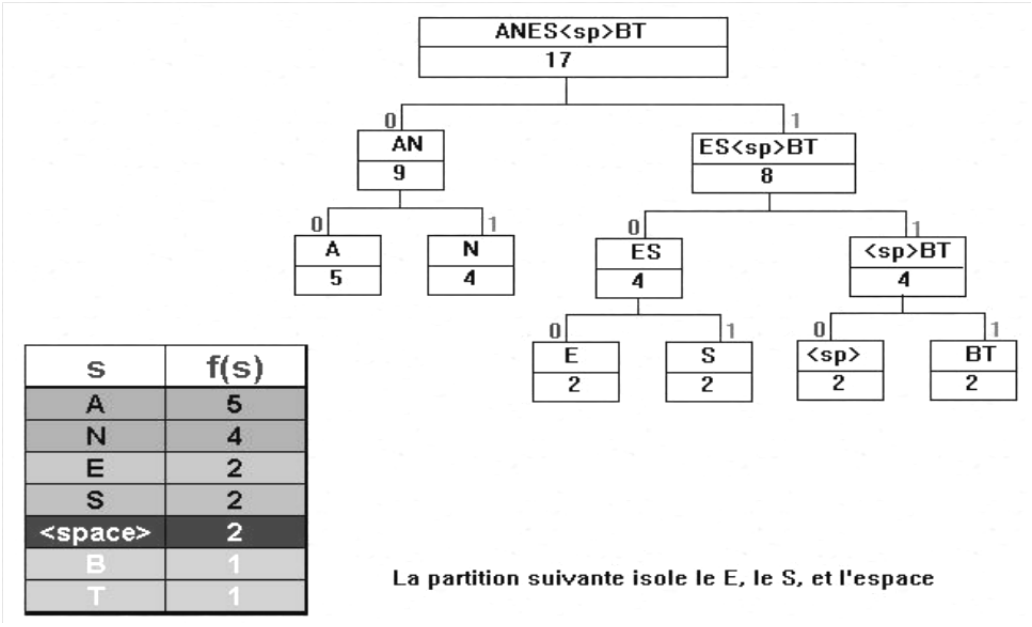
La deuxième étape va isoler les symboles A et N, et définir une nouvelle subdivision des symboles restants.



s	f(s)
A	5
N	4
E	2
S	2
<space>	2
B	1
T	1

Figure 1.4 : La 2ème étape de l'algorithme de Shanon-Fano.

La troisième partie :

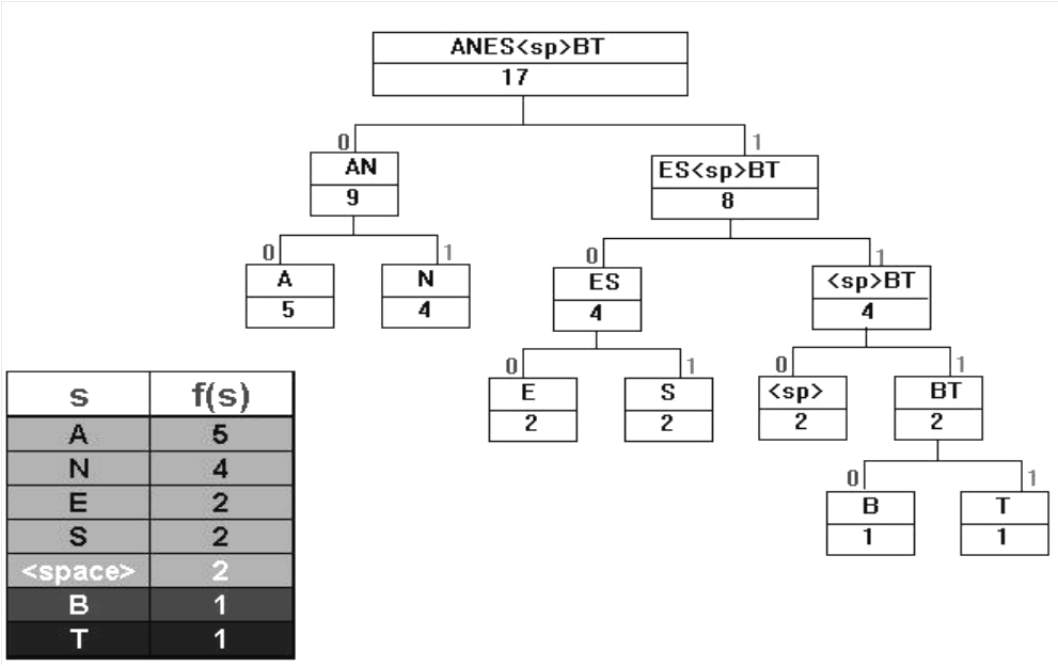


s	f(s)
A	5
N	4
E	2
S	2
<space>	2
B	1
T	1

Figure1.5 : La troisième étape de l'algorithme Shanon-Fano.

La troisième partition ne laisse plus que le couple BT non résolu.

Et la quatarième et dernière partie :



s	f(s)
A	5
N	4
E	2
S	2
<space>	2
B	1
T	1

Figure1.6 La quatarième étape de l'algorithme Shanon-Fano.

Pour connaître le code associé à chaque lettre, nous parcourons l'arbre final de haut en bas, et nous obtenons :

A	00
N	01
E	100
S	101
<space>	110
B	1110
T	1111

VI.2. Quelques techniques de compression avec perte :

Actuellement la compression avec perte est un sujet de recherche très prisé des laboratoires.
Plus la recherche avance, plus nous sommes capable d'avoir de forts taux de compression tout en maintenant une qualité tout à fait acceptable.

VI.2.1. Compression d'images :

Quelques définitions :[5]

Les espaces de couleurs :

Les couleurs sont déterminées en spécifiant quelques valeurs en général trois. Ces valeurs spécifient la quantité de chaque couleur fondamentale qui sont ensuite mixées ensemble pour former la couleur finale. On distingue plusieurs formats différents.

a- RGB (Red-Green-Blue):

C'est le format le plus connu. C'est un système additif dans lequel nous varions la quantité du Rouge (Red), du vert (Green) et du bleu (Blue) en les additionnant au noir pour produire de nouvelles couleurs. Les fichiers graphiques utilisant le système RGB représente chaque pixel comme étant un triplet de valeur chacune représentant la quantité d'une des trois couleurs de bases. Par exemple pour une image en "true color" le triplet (0, 0, 0) représente le noir alors que le (255, 255, 255) représente le blanc. Si les trois valeurs sont les même, ont a un ton gris. Nous pouvons parfois trouver des pourcentages de valeurs dans le triplet au lieu des valeurs proprement dites.

-b- CMY (Cyan-Magenta-Yellow):

C'est un système de valeurs soustractif utilisé par les imprimantes et les photographes pour un rendu final généralement sur des surfaces blanches. Ce système est l'inverse de RGB: nous soustrions de la couleur blanche une certaine quantité des couleurs de bases Cyan (Cyan), Magenta (Magenta) ou Jaune (Yellow) pour produire de nouvelles couleurs. En sachant toutefois que les couleurs CMY sont l'inverse de RGB donc en incrémentant la quantité du Jaune nous allons décrémenter la quantité de bleu. La représentation est la même que RGB en sachant que (0, 0 ,0) correspond à blanc et (255, 255, 255) correspond à noir

-c-HSV (Hue, Saturation, and Value):

Au lieu de mixer des couleurs, nous faisons plutôt varier le degré de propriétés de celles-ci. Le premier terme Hue (couleur) spécifie la couleur dans le sens commun du terme: rouge, vert, orange etc. Le deuxième Saturation (Saturation) aussi appelé chroma spécifie la quantité de blanc dans les couleurs. Une couleur saturée (100 pour-cent) ne contient pas de blanc. Par exemple un rouge avec 50% de saturation apparaît rose. La troisième valeur Value (Valeur), aussi appelée brightness (éclat, vivacité) est le degré de luminescence de la couleur ou "combien la couleur est claire".

-d- YUV (Y-signal, U-signal, and V-signal):

C'est un modèle un peu différent des autres modèles colorimétrique. C'est une transformation linéaire de RGB et est très utilisé dans le codage des couleurs pour les transmissions et en télévision. Y spécifie la luminance (information de luminosité) alors que les composantes U et V correspondent à la chrominance (information de couleur).

VI.2.1.1. La compression avec JPEG (Joint Photographic ExpertsGroup) :[6]

La compression JPEG consiste à effectuer une dégradation de l'image imperceptible à l'œil, de façon à offrir un taux de compression beaucoup plus intéressant que les autres méthodes, donc à permettre de réduire considérablement l'espace occupé par le fichier sur le disque ou la vitesse de son transfert sur un réseau. Pour ce faire, l'image est décomposée en blocs qui vont ensuite subir diverses opérations pour diminuer la taille de l'image initiale. Cette compression est donc une compression avec perte car elle subit une perte définitive d'information, même si il est possible de revenir à une image proche de l'image initiale avec certains procédés. Voici le procédé de compression et de décompression JPEG d'une image :

Découpage en blocs de 8×8 pixels et transformation DCT :

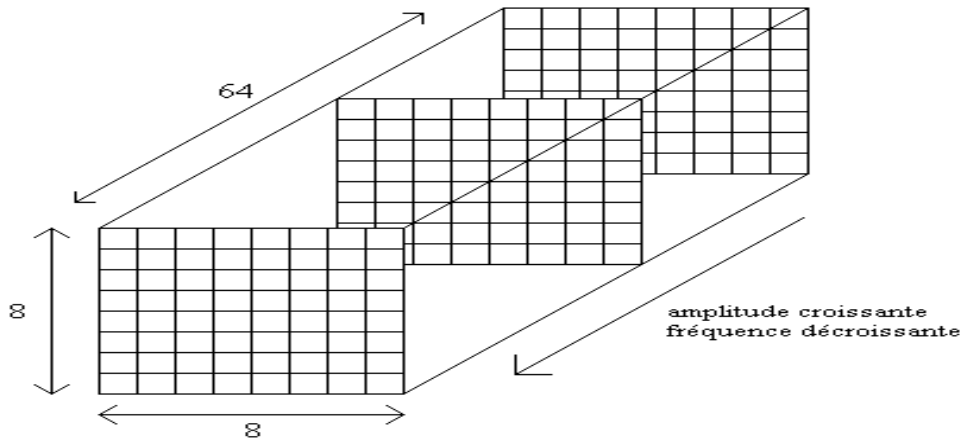
Une image informatique est constituée de points de couleur uniforme : ce sont les pixels. Ils sont caractérisés par leur position dans l'image, et par leur couleur. On peut représenter l'image par une matrice, qui a autant d'éléments que l'image a de pixels. Les coordonnées de ceux-ci sont données par leur position dans la matrice, et leur couleur par la valeur qu'ils ont dans la matrice.

Nous découpons ensuite cette matrice en sous matrice de 8×8 pixels. A chacune de ces matrices on a applique ensuite la transformation DCT (Discrete Cosine Transform, ou Transformée en Cosinus Discrète). Cette technique est similaire de à la transformée de Fourier : transformation en somme de sinus et de cosinus de différentes fréquences et amplitude, mais pour la DCT on ne prend en compte que les cosinus.

$$DCT(i,j) = \frac{1}{\sqrt{2N}} C(i)C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel\,(x,y) \cos\left[\frac{(2x+1)i\Pi}{2N}\right] \cos\left[\frac{(2y+1)i\Pi}{2N}\right]$$

$$C(x) = \begin{cases} \frac{1}{\sqrt{2}} & pour\ x = 0 \\ 1 & pour\ x > 0 \end{cases}$$

Chaque morceau d'image est donc représenté pas une somme de 64 fonctions cosinus de fréquence et d'amplitude aux quels on associe une matrice ce qui nous donne une matrice tridimensionnelle 8×8x64



Prenons pour simplifier une matrice 8×8 qui est la somme des 64 matrices que compote un bloc. Dans cette matrice est représenté les couleurs de chaque pixel de l'image (chaque coordonnée correspond à une couleur d'un pixel)

Matrice 1

$$\begin{bmatrix} 100 & 155 & 131 & 116 & 151 & 135 & 131 & 211 \\ 120 & 135 & 127 & 88 & 155 & 131 & 155 & 179 \\ 120 & 135 & 151 & 100 & 179 & 116 & 155 & 167 \\ 120 & 155 & 151 & 108 & 191 & 112 & 155 & 179 \\ 135 & 151 & 135 & 210 & 197 & 112 & 179 & 179 \\ 120 & 151 & 155 & 151 & 151 & 116 & 179 & 179 \\ 135 & 151 & 167 & 167 & 151 & 151 & 167 & 171 \\ 120 & 151 & 179 & 151 & 151 & 131 & 155 & 167 \end{bmatrix}$$

Matrice 2

$$\begin{bmatrix} 145 & -84 & 34 & -69 & 4 & -66 & -35 & 72 \\ -45 & -28 & 28 & 19 & 10 & -54 & 5 & 15 \\ 0 & -2 & -8 & -15 & -9 & 0 & 30 & -41 \\ 9 & -14 & 15 & -11 & 5 & 8 & -12 & -32 \\ 1 & 1 & 3 & -11 & 7 & -23 & -4 & 0 \\ 18 & 4 & -17 & -10 & 4 & -10 & 7 & -10 \\ -5 & 1 & -7 & -20 & 1 & -1 & -3 & 5 \\ 3 & 1 & 1 & 9 & 2 & 7 & 2 & -2 \end{bmatrix}$$

La transformation de la matrice1 par la DCT donne la matrice2. La transformation DCT est une transformation réversible, il n'y a donc pas de perte d'information à cette étape.

La quantification :

La quantification pondère l'importance des différents domaines de fréquences, sachant que la vision humaine est moins sensible aux hautes fréquences de détails. On va donc attribuer à chaque donnée un coefficient de poids qui correspond à son importance dans l'image. Cela revient à diviser les coefficients de Fourrier de l'image par ces coefficients de poids, et ne garder que ceux supérieurs à une certaine valeur. Cela va nous permettre d'atténuer les hautes fréquences, c'est à dire celles auxquelles l'œil humain est très peu sensible. Ces fréquences ont des amplitudes faibles, et elles sont encore plus atténuées par la quantification.

$$F*(u,v) = \left[\frac{F(u,v) + \frac{Q(u,v)}{2}}{Q(u,v)}\right]$$

$Q(i, j) = 1 + (1 + i + j) \times Fq$ avec Fq facteur de qualité et Q le pas de quantification.

Par exemple, pour $Fq = 5$ nous obtenons la matrice 3. Nous allons finalement prendre la partie entière de la division de chaque valeur de la matrice 2 par la valeur de la matrice 3 ayant la même position, ce qui nous donne la matrice 4.

$$\begin{bmatrix} 6 & 11 & 16 & 21 & 26 & 31 & 36 & 41 \\ 11 & 16 & 21 & 26 & 31 & 36 & 41 & 46 \\ 16 & 21 & 26 & 31 & 36 & 41 & 46 & 51 \\ 21 & 26 & 31 & 36 & 41 & 46 & 51 & 56 \\ 26 & 31 & 36 & 41 & 46 & 51 & 56 & 61 \\ 31 & 36 & 41 & 46 & 51 & 56 & 61 & 66 \\ 36 & 41 & 46 & 51 & 56 & 61 & 66 & 71 \\ 41 & 46 & 51 & 56 & 61 & 66 & 71 & 76 \end{bmatrix}$$

Matrice 3

$$\begin{bmatrix} 24 & -7 & 2 & -3 & 0 & -2 & 0 & 1 \\ -4 & -1 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Matrice 4

Nous pouvons remarquer que la matrice final (matrice 4) comporte un grand nombre de 0, c'est donc une valeur récurrente qui sera compressé pas un codage statique.

Voici une figure qui résume les deux opérations de compression et décompression JPEG :

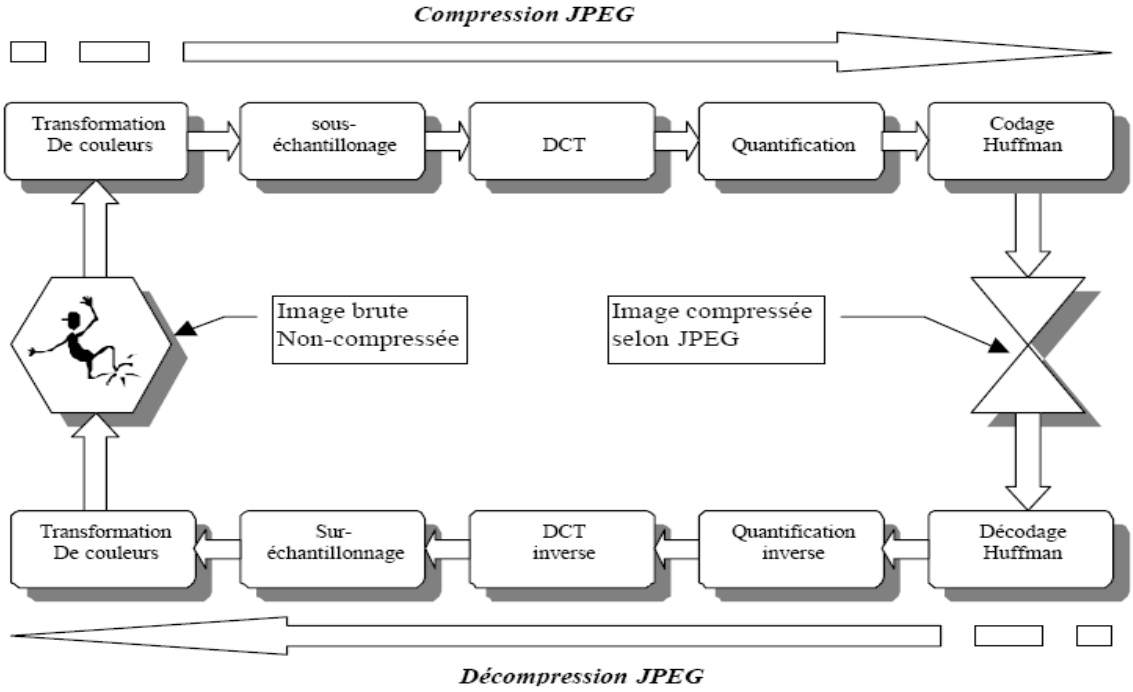


Figure1.7 : La compression et la décompression avec l'algorithme JPEG.

VI.2.1.2. La méthode RVQ :[5]

A l'origine, la méthode RVQ à été mise au point par la société Knowledge Adventure spécialisée dans les logiciels éducatifs interactifs. Non seulement elle exige une puissance de calcul considérable, mais encore elle nécessite une intervention humaine à chaque étape afin de choisir les meilleurs paramètres de compression. Si la compression est un processus lent, la décompression en revanche est rapide. Cette rapidité rend RVQ utilisable pour la décompression d'informations en temps réel, comme des données audio ou vidéo.

La méthode RVQ exploite la quantification pour choisir les informations à supprimer, mais elle effectue cette quantification sur les blocs eux-mêmes. Une image 640 x 480 découpée en blocs de 16 x 16 pixels est composée de 1200 blocs. Le but consiste à remplacer chaque bloc par un autre, comparable mais sélectionné parmi un sous-ensemble bien plus réduit de blocs. En fonction de leur similarité, les 1200 blocs de l'image d'origine sont donc répartit pour former 16 groupes au plus. Pour chaque groupe, un bloc de référence est alors calculé : il correspond à la moyenne de tous les blocs appartenant à ce groupe. L'étape de la quantification remplace finalement chacun des 1200 blocs originaux par son équivalent le plus approché, sélectionnée parmi les 16 blocs de référence, dont le choix est évidemment crucial.

Opération de type OU EXCLUSIF ou XOR :

La deuxième étape effectue une opération de type ou exclusif entre l'image d'origine et l'image obtenue en première étape. L'image résultante obéit à des lois simples : lorsque les bits sont identiques dans les deux images superposées, le résultat est un bit positionné à 0; Quand ils sont différents, le bit résultant est un 1. Ainsi lorsqu'une image est superposée sur elle-même avec cette opération XOR, le résultat obtenu est un écran de 0.

Le processus décrit dans la première étape est alors réitéré sur la nouvelle image. Le nombre de blocs de références n'est pas nécessairement de 16, comme lors de la première étape : seule l'intervention humaine peut en juger, de manière subjective, en fonction de la qualité d'image souhaitée.

Une fois la deuxième approximation calculée, l'image résultante est soustraite à son tour à la première avec une opération XOR. La raison est simple : XOR est une opération réversible (autrement dit, le second XOR annule les effets du premier). L'image obtenue alors est bien plus proche de l'originale que ne l'était la première approximation. Puis le processus est réitéré une troisième fois. Après ces trois itérations, l'image résultante, approximative, se rapproche beaucoup de l'image d'origine mais sa description prend un nombre de bits bien moindre.

La compression de type RVQ nécessite un temps considérable pour traiter une image. En revanche, la décompression est très rapide, puisqu'il suffit pour l'essentiel de récupérer des blocs et de les combiner à l'affichage.

VI.2.1.3. La compression fractale :[5]

La compression fractale est l'apanage de la société Iterated Systems. L'idée étant qu'une image est stockée sous la forme d'un ensemble de formules plutôt que d'un ensemble de points (ou bien de transformations d'ensembles de points). Ainsi l'apparence d'une image dépend uniquement du périphérique de sortie, car la compression fractale est indépendante de la résolution. Les principes mathématiques mis en œuvre pour la compression sont extrêmement complexes, mais à la manière des procédés JPEG et RVQ, la décompression est raisonnablement rapide.

A la manière des méthodes JPEG et RVQ, la compression fractale commence par découper l'image en blocs, mais chaque bloc est codé sous la forme d'une transformation mathématique calculée à partir d'autres régions de l'image originale (appelées régions de références). L'obtention d'une transformation optimale nécessite parfois une ou plusieurs modifications de la région de référence (par rotation ou par changement d'échelle notamment). De la sorte, les régions de références ainsi transformées correspondent autant que possible aux blocs en cours d'encodage. De façon évidente, la qualité de l'image décompressée dépend directement du temps passé à rechercher la région de référence la plus appropriée et la transformation la plus judicieuse.

VI.2.1.4. Compression par Ondelette :[5]

La transformation par Ondelettes est une technique inventée par Summus Ltd et qui consiste à décomposer une image en une multitude de sous-bandes, c'est à dire des images de résolution inférieure. Nous distinguons 4 étapes différentes pour procéder à la transformation:

1. Moyenner les pixels de l'image originale deux à deux suivant l'axe horizontal ; par exemple: ($H(x) = (X_n+X_n+1) / 2$).
2. Calculer l'erreur entre l'image originale et l'image sous-échantillonnée dans le sens horizontal; par exemple : ($G(x) = (X_n-X_n+1)/2$).
3. Pour chacune des deux images intermédiaires, moyenner les pixels deux à deux suivant l'axe vertical ; par exemple : ($H(y) = (Y_n+Y_n+1) / 2$).
4. Pour chacune des deux images intermédiaires, calculer l'erreur suivant l'axe vertical ; par exemple : ($G(y) = (Y_n-Y_n+1) / 2$)

Le résultat est une image d'approximation qui a une résolution divisée par deux et trois images de détails qui donnent les erreurs entre l'image originale et l'image d'approximation. Cette transformation est répétée autant de fois que nécessaire pour obtenir le nombre voulu de sous-bandes.

Il n'y a pas de pertes à ce stade de la transformation. Les pertes subviendront lors de la compression. Les étapes de compression sont les suivantes :

1. Transformations par ondelettes.
2. Quantification : les valeurs des images de détails inférieures à un certain niveau sont éliminées, en fonction de l'efficacité recherchée. C'est cette étape qui introduit des pertes.
3. Codage des valeurs restantes.

La transformation inverse par ondelettes reconstruit une image originale. La construction de l'image à partir des sous-bandes restitue l'image en mode progressif.

L'affichage de l'image peut s'effectuer en deux modes :

- Soit la taille de l'image augmente au fur et à mesure de la lecture du fichier compressé.
- Soit la résolution de l'image augmente au fur et à mesure de la lecture du fichier compressé.

VI.2.2.La compression de vidéos :[7]

Une vidéo est une série d'images placées les unes à la suite des autres. A titre d'exemple, les films diffusés au cinéma sont constitués de 24 images/seconde. Si une image a une taille d'environ 1mb, la taille non compressée d'une vidéo d'une minute est d'environ 1*24*60=1440mb. Une façon simple de compresser une vidéo est de compresser individuellement chaque image de la vidéo, ce qui est fait par le codec8 Mjpeg. Avec ce type de compression nous obtenons un taux de compression assez faible, comparé aux techniques de compression vidéo plus classiques dont les principes sont détaillés ci-dessous.

On peut constater qu'il existe une grande redondance temporelle entre les images successives d'une vidéo. L'idée est donc de décrire une image par une autre en décrivant les modifications par rapport à celle-ci. Avec ce type de technique nous atteignons des taux de compression supérieurs à 100 :1. La majorité des codecs actuels utilisent ce type de système.

La compression d'une séquence d'images fait appel aux meilleures techniques de compression pour une image isolée ainsi qu'à des techniques très élaborées pour l'aspect temporel : détection de mouvement et prédiction.

Groupes d'images :

Plus précisément, un compresseur MPEG emploie trois types de compression d'images qu'il alterne dans la vidéo : les images fournies en entrée sont compressées soit avec la méthode I, la méthode P ou la méthode B. Cela produit des données plus ou moins importantes dans le fichier de sortie. Par abus de langage, nous parlons d'images I, P ou B.

Image I :

Il s'agit d'une image qui a été compressée avec une variante de JPEG : couleurs en YCrCb avec sous-échantillonnage des plans Cr et Cb, DCT, quantification un peu plus violente que pour une image fixe, encodage zigzag classique, avec les coefficients DC0 des macros blocs mis à part et encodés en relatif les uns par rapport aux autres.

Image P

Une image P est compressée en comparant l'image fournie en entrée avec la précédente image de type P ou I du fichier. Le compresseur travaille avec des macros blocs 16x16 et indique seulement à quel autre bloc il ressemble dans l'image précédente et s'il y a un mouvement ou une erreur de prédiction (changements dans les coefficients DCT). Le compresseur peut également ignorer le macro bloc s'il ne change pas assez et au pire, s'il change trop, il le code comme dans une image I. Plus on accumule des images de type P, plus le résultat de la décompression sera imparfait. Il faut donc régulièrement insérer une image I.

Image B

Une image devant être compressée avec la méthode B est comparée à toutes les images de type I ou P précédentes ou suivantes, à la recherche de la plus ressemblante. Le principe est le même que pour les images P : vecteurs de déplacement et informations sur les changements dans la DCT du macro bloc. Les images B sont encore moins précises que les images P car elles s'appuient sur des images elles-mêmes imprécises.

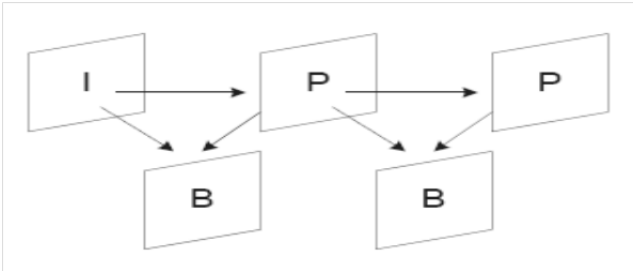


Figure1.8 : Images intra, prédites et bi-prédites.

VI.2.2.1. La compression avec MJPEG (Motion JPEG):[7]

Le format MJPEG consiste à stocker une succession d'images encodées en JPEG. Les images sont mises l'une après l'autre dans le fichier. Comme la compression JPEG n'examine pas les corrélations temporelles, on n'arrive pas à un bon taux de compression global. Les fichiers MJPEG sont en général de très grande taille. La norme MJPEG2000 fait appel à la compression JPEG2000 qui est meilleure. L'intérêt du format MJPEG n'est pas dans le taux de compression pour le stockage, mais dans la rapidité d'accès aux images. Le flux MJPEG autorise un accès immédiat à n'importe quelle image. Cela permet la recherche d'information et le montage vidéo. Au contraire, les formats MPEG et autres stockent les images dans un ordre spécifique : celui du décodage et non pas celui de la lecture, comme nous allons le voir ci-dessous.

Le MJPEG applique un algorithme de compression à chaque image de la vidéo en les compressant une à une en JPEG. Ce système permet d'accéder aléatoirement à n'importe quelle partie d'une vidéo car chaque image est traitée séparément. Le M-JPEG est principalement utilisé dans les studios de montage numérique.

VI.2.2.2. La compression avec MPEG (Motion Picture Expert Groups):[5]

MPEG est développé par un groupe d'experts qui a été chargé de définir ce format. Alors que JPEG est défini pour compresser des images fixes, MPEG est lui défini pour compresser et synchroniser des signaux audio et vidéo avec un flux de données. Il est dérivé de JPEG car il utilise le même algorithme de compression spatiale. Toutefois il n'est pas capable de sauver des images à ce format là (720x576) ; cela demande plus de bande passante que ses 1,86 Mb/sec ne le lui permette. C'est pourquoi MPEG est définit pour stocker des images au format de 352x240 pixels à 30 images secondes.

Les types de MPEG :[8]

-a- MPEG-1 :

MPEG-1 est basé sur les standards de télévision numérique (CCIR-601) utilisé aux USA. Cette méthode contient cinq parties : une partie système, une partie vidéo, une partie audio, une partie tests de conformité et une partie simulation logicielle. Le but des chercheurs qui ont travaillé sur cette première norme était de stocker et de reproduire de la vidéo avec le son associé sur des supports de stockage, en qualité magnétoscope (320 x 240), avec un débit maximum égal à 1,5 Mbits/s. Concrètement, MPEG-1 est devenu par la suite la norme de stockage de vidéos sur CD-ROM au format CD-I ou CD-vidéo. Lorsque la norme est sortie, la compression des séquences vidéo en MPEG-1 nécessitait des quantités de mémoire (16 Mo de RAM minimum) que les ordinateurs personnels de l'époque n'avaient pas encore. Aussi la compression vidéo MPEG-1 nécessitait de coûteux investissements.

-b- MPEG-2 :

Elle a été finalisée en 1996. Elle comporte neuf parties : les cinq parties semblables au MPEG-1, plus quatre autres parties composées d'extensions diverses. MPEG-2 a été développée pour la compression de la vidéo de qualité télévision à un débit de 4 à 6 Mbits/s. Un peu plus tard, lorsque MPEG-3 fut abandonné, MPEG-2 intégra la compression de la télévision à haute définition (TVHD). Aujourd'hui, MPEG-2 est aussi le format utilisé pour stocker les films sur DVD. Si sur le fond, le principe général de fonctionnement de MPEG-2 est très sensiblement identique à celui de
MPEG-1, il existe de nombreuses différences de formes entre les deux normes. Tout d'abord, MPEG-2 permet la compression d'images entrelacées là où MPEG-1 ne traite que les images en mode progressif, et ceci bien évidemment pour servir à la télévision numérique.

-c- MPEG-4 :

Elle a été finalisée dans sa version 2 en 1999. La finalisation définitive est prévue pour février 2002. Elle comporte sept parties, toujours sur le même schéma des cinq parties semblables au MPEG-1 et MPEG-2, plus deux autres parties composées d'extensions diverses. D'autres extensions sont encore attendues. Les objectifs de cette norme sont assez nombreux et touchent de vastes domaines. En fait, MPEG-4 a pour ambition de fusionner trois mondes : l'informatique, les télécommunications et la télévision. Cette norme s'appuie sur les succès futurs de la télévision numérique, des applications graphiques interactives et du multimédia (web), son but étant d'assurer une standardisation technologique à tous les niveaux : production, distribution et diffusion. Ses domaines d'application sont immenses : communication temps réel (visiophone), multimédia mobile, téléconférence, post-production (cinéma et télévision), stockage (DVD) et recherche d'informations basée sur le contenu, etc.

-d- MPEG-7 :

MPEG-7 a pour principal objectif de simplifier et d'optimiser la recherche de fichiers multimédia. Pour cela, MPEG-7 va spécifier une description standard de différents types d'informations multimédia. Cette description devra être associée au contenu lui-même pour permettre la recherche rapide et efficace des informations qui intéressent l'utilisateur. Cela peut s'appliquer aussi bien à de la vidéo qu'à du son, des images, des graphiques, des animations 3D, ou encore à la façon dont des objets vidéo sont combinés dans une séquence. Nous pouvons simplifier en disant que MPEG-7 est une "sur-couche" de MPEG-4 dans le sens où MPEG-4 fournit la manière de coder les séquences audiovisuelles tandis que MPEG-7 ajoute une couche d'informations décrivant ces séquences.

-e- MPEG-21 :

La dernière née des normes MPEG ajoute notamment à MPEG-7 une couche "sécurité" en proposant tout un éventail de solutions permettant des échanges de matériaux audiovisuels en toute sécurité (déclaration, identification, protection). MPEG-21 redéfinit également la structure interne des séquences audiovisuelles en optimisant les interactions possibles entre les objets audiovisuels, et entre les utilisateurs et ces mêmes objets.

VI.2.3. La compression audio :[9]

Elle est adaptée à la compression de sons de musique. Le principe :

- L'oreille humaine ne perçoit qu'environ 10% de l'information sonore du fait de limitations mécaniques.
- Nous parlons de compression psycho-acoustique (suppression d'une partie des données lors de la compression).
- Les signaux de fréquences voisines se masquent, un signal fort masque les signaux proches plus faibles.

La psycho acoustique : est l'étude des sensations auditives de l'homme. Elle rassemble et décrit les relations qui existent entre le phénomène acoustique physique, la perception que nous permet notre oreille et la description que nous en faisons.

La compression MP3 :[10]

Le MP3 (MPEG-1/2 Audio Layer 3) est un algorithme de Compression audio avec perte. Le taux de compression entre un son de CD audio et un son MP3 est de 1/12 pour un son stéréophonique échantillonné à 44.1 KHz.

Les grands principes de l'encodage MP3 sont :

- ✓ L'oreille humaine ne peut discerner que des sons compris entre 0.002kHz et 20kHz environ. Les fréquences qui ne sont pas dans cet intervalle ne sont pas stockées.
- ✓ La sensibilité de notre oreille suit une courbe en fonction de la fréquence ; cette sensibilité est maximale entre 2 et 5 kHz.
- ✓ L'effet de masque.
- ✓ Le joint Stéréo.
- ✓ Le codage d'Huffman.

Il s'agit donc d'analyser le signal dans le domaine des fréquences (analyse spectrométrique) afin de ne conserver que les sons audibles. Le système va décomposer le son en un ensemble de sons ayant chacun une fréquence donnée. Ce découpage se fait par la transformée de Fourier. Une fois cette étape effectuée, on applique les principes cités ci-dessus

pour éliminer les fréquences non perçues par l'oreille. Nous allons donc ne garder que les données auxquelles notre oreille est sensible.

L'effet de masque :

Un son précédé ou suivi par un autre son plus fort que lui sera en partie ou totalement masqué. Le niveau de masquage dépend du niveau du son masquant mais aussi de sa fréquence et de l'intervalle qui le sépare du son masqué. Appliqué à la compression audio, l'effet de masque permet de ne transmettre que les sons réellement détectés par l'oreille. Tous les sons masqués sont éliminés. Pour illustrer l'effet de masque voici un exemple : dans des parties calmes d'une musique, nous pouvons parfois entendre des sons très faibles comme la respiration d'un chanteur, mais pas dans les parties plus élevées.

Le réservoir de bytes :

Souvent, certains passages d'une musique peuvent ne pas être encodés sans altérer la qualité. Le mp3 utilise donc un petit réservoir de bytes qui agit en utilisant des passages qui peuvent être encodés à un taux inférieur au reste des données.

Le joint Stéréo :

Nous avons constaté qu'en dessous d'une fréquence donnée, l'oreille humaine est incapable de localiser l'origine du son. Dans beaucoup de chaînes Hi-Fi, le boomer(le haut-parleur qui a pour fonction de retranscrire les fréquences graves) est unique. Cependant, nous n'avons pas l'impression que le son vient de ce boomer mais plutôt des haut-parleurs satellites. Le format mp3 peut exploiter (en option) cette astuce en utilisant la méthode du joint stéréo, c'est-à-dire que certaines fréquences sont enregistrées en mono mais sont accompagnées d'informations complémentaires afin de restituer un minimum d'effet spatial.

V. Les atouts de la compression :[9]

- Gain en espace de stockage sur disque et en mémoire, par exemple :
  - Une image JPEG se compresse facilement à 90 %.
  - Un fichier HTML compressé en GZIP se comprime facilement à 50% de sa taille.
- Gain en temps sur les lectures/écritures.
  - Moins de données à lire/écrire physiquement sur disque.
  - Compression + écriture plus rapide qu'une écriture des données brutes.
  - Lecture + décompression plus rapide qu'une lecture des données brutes.

Exemple :

- Temps pour écrire une image de 14Mo: 5s (soit 2,8Mo/s).
- Temps pour compresser une image de 14Mo en JPEG: 3,4s, soit 6,5Mo/s, résultat: 503Ko (soit 4% de l'original).
- Temps pour écrire une image JPEG de 503Ko: 0,2s.
- Temps total compression + écriture: 3,6s, soit 1,4s de moins (gain de 24% en temps et 96% en volume stocké sur disque).

- Gain sur les temps de transmission
  - Temps de transmission à 56Kbps/s d'un fichier HTML de 30Ko: 5,4s.
  - Temps de compression du fichier HTML au format GZIP: 0,05s (soit 600Ko/s), taille compressée: 15Ko (gain de 50% en volume)
  - Temps de transmission de l'HTML compressé: 2,7s.

- Temps total compression/transmission/décompression: 2,8s, au lieu de 5,4s, soit un gain de près de 50% en temps.

- ❖ Moins de données à transmettre.
  - Compression + envoi / réception + décompression plus rapide qu'une transmission des données brutes.
- ❖ Les formats d'archive (comme ZIP ou Stuffit) réduisent le nombre de fichiers
  - Un seul fichier ZIP ou Stuffit peut contenir des centaines voire des milliers de fichier.
  - Ces fichiers peuvent être organisés hiérarchiquement
- ❖ Les fichiers compressés sont vérifiables (notion de checksum).

VI. Compression avec et sans pertes, comparaisons des diverses méthodes :[5]

VI.1. Comparaison des diverses méthodes de compression sans pertes :

	RLE	***LZW et LZ77***	***CCITT Groupe IV***
Avantages	Non destructif, Très simple à implémenter, Rapide d'exécution.	Non destructif pour images en 4, 8, 24 bits/pixel	Non destructive pour document 1 bits/pixel
Désavantages	Très faible compression	Faible compression	Pas de restitution de niveaux de gris
Format de fichier	BMP, PCX, RLE	GIF, TIF, PNG pour LZ77	TIF, IOCA, CALS
Images supportées	2, 4, 8, 24 bits/pixel	4, 8, ou 24 bits/pixel	Pages de textes. Dessins au trait en 1 bit/ pixel
Taux de compression	Typique = 2	Typique >= 2 Non programmable Efficacité fonction image	Typique = 20 Non programmable.
Compression/ décompression	Par logiciel Ultra-rapide Disponible sous Win (3.1, 95, NT), Dos.	Par logiciel. Ultra-rapide. Disponible sous Win (3.1, 95, NT) MacOs, Dos, OS/2	Par logiciel. Rapide. Disponible sous Win (3.1, 95, NT) MacOs, Unix

Tableau1.1 : Comparaison des méthodes de compression sans perte.

VI.2. Comparaison de diverses méthodes de compression avec pertes :

	JPEG	***Fractal***	***ondelettes***
Avantages	Compression normalisée ISO pour images en millions couleurs	Zoom sans effets de pixellisation. Taille finale ajustable	Excellente qualité Nombreux paramètres Réglables avant et après compression
Désavantages	Effet de mosaïque à fort taux de compression	Compression lente. Flou introduit à forts taux de compression	Flou introduit à forts taux de compression
Format de fichier	JFIF (avec extension JPG). TIF-JPEG	FIF (Fractal Image Format)	WI (Wavelet Image)
Images supportées	Photo numérique en 8, 24 bits/pixel	Photo numérique en 8, 24 bits/pixel	Photo numérique en 8, 12, 24 bits/pixel
Taux de compression	Typique = 20 Contrôlable par un facteur de qualité	Typique = 20 Contrôlable par un facteur de qualité ou ajustable de 10 à 100	Typique = 40 Contrôlable par un facteur de qualité ou ajustable de 1 à 250
Compression/ décompression	Par logiciel. Moyen à rapide Disponible sous Win (3.1, 95, NT) MacOs, Unix, OS/2, DOS	**Compression** Par logiciel ou carte hardware (FTC III-IV) Très long (minutes). Win (3.1, 95, NT). **Décompression** Par logiciel. Rapide. Mode progressif. Zoom fractal. Win (3.1, 95, NT), MacOs.	**Compression** Par logiciel. Rapide. Mode compressif. Filtrage contraste. Optimisation locale. Win (3.1, 95, NT), MacOs, Dos, OS/2, UNIX, DSP **Décompression** Par logiciel. Rapide. Win (3.1, 95, NT), MacOs, Dos, OS/2, UNIX, DSP, Plug-in Netscape, ActiveX MSexplorer

Tableau1.2 : Comparaison des méthodes de compression avec perte.

VII. Conclusion :

La compression de données a encore de beaux jours devant elle. Elle permet de résoudre des problèmes de volume, de stockage et de débit. De nombreux travaux de recherche sont en cours afin d'améliorer les performances des techniques de compression.

Dans ce chapitre nous avons vu quelques définitions qui sont en relation avec la compression des données, ensuite nous avons fait une classification des algorithmes de compression selon plusieurs critères, après nous avons donné quelques exemples d'algorithmes selon les types des fichiers ; texte, image, vidéo et son. Nous avons cité les atouts de la compression et en fin nous avons fait une comparaison entres les algorithmes que nous avons vu (sans perte et avec perte).

Chapitre II

Les méthodes VLC

I. Introduction :

Nous distinguons dans les systèmes de codage les systèmes de codage à longueur fixe où tous les mots ont la même longueur (par exemple: code ASCII) et les systèmes à longueur variable (alphabet morse) où les différents mots du code n'ont pas nécessairement la même longueur, en bits. Le passage d'un code à longueur fixe à un code de longueur variable constitue une technique de compression.

Dans de nombreux standards de compression, tel que JPEG, H263 ou le mode de base de H264, une étape de codage à longueur variable (CLV ou bien en anglais, VLC pour *variable length coding*) des données est effectuée. Les codes à longueur variable sont apparus dès les premiers travaux de Shannon sur la théorie des communications. Ceci est d'autant plus remarquable que l'étude des codes équivaut à celle d'un objet mathématique. Les techniques courantes de transmission de données "compressées" font souvent appel aux codages à longueur variable. Nous appelons aussi le codage VLC par le codage entropique.

II. Définitions :

1. Codage entropique (Codage à longueur variable VLC ou bien codage source) :[11]

La notion de codage entropique ou compression réversible d'une source correspond à un codage sans perte des symboles de la source avec pour objectif d'atteindre une limite théorique du gain de compression de Shannon caractérisée par l'entropie de la source. Le but du codage entropique est de remplacer les messages émis par une source S utilisant un alphabet N-aire $\{S_1, S_2, \ldots, S_N\}$ par des messages écrits dans l'alphabet binaire utilisé par les systèmes de stockage d'information sur ordinateur. Dans la suite nous supposons qu'un code associe à chaque symbole *si* (i=1, ... N) de la source un mot de code binaire *mi*, séquence de *li* bits.

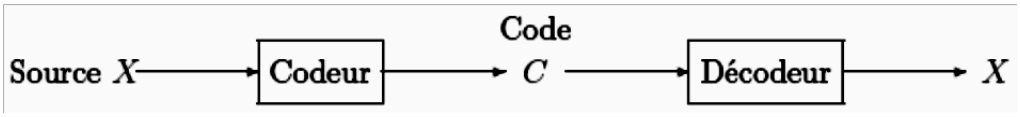


Figure2.1 : Le codage VLC.

2. Source discrète sans mémoire :[12]

Pour caractériser les sources, de nombreux termes sont empruntés à la description du langage courant. Une source dispose d'un **"alphabet"** constitué d'éléments ou symboles ou caractères $\{x_1, x_2, x_3, \ldots, x_k\}$ tel que K est la longueur de l'alphabet. Ces symboles sont associés pour constituer un message. Emettre un message revient à émettre une succession de symboles appartenant à une source. Chaque symbole x_k de l'alphabet a une probabilité d'utilisation p_k.

Pour simplifier le problème, une catégorie de sources est plus simple à modéliser : celle des sources pour lesquelles la probabilité d'émission d'un caractère est indépendante de ce qui a été émis avant ou sera émise après. C'est ce qui défini une source sans mémoire.

3. L'entropie d'une source simple :[12]

Soit une source S définie par son alphabet $\{S_1, S_2, \ldots\ldots, S_N\}$ de symboles régis par une loi de probabilité P : $\{P(S_1), P(S_2), \ldots\ldots P(S_N)\}$ .Une source sera dite simple (ou sans mémoire) si les symboles émis par la source S sont indépendants et de même loi. Une suite de N symboles émis aux instants 1, 2, ...*n* par S suit donc une loi de probabilité :

$$P(s1, s2, \ldots\ldots sn) = p(s1)\, p(s2). \ldots\ldots p(sn).$$

L'entropie d'ordre zéro H(S) d'une source simple S, de loi de probabilité P, est définie par l'expression :

$$H(S) = -\sum_{i=1}^{N} p(si) log2(p(si))$$

L'entropie dans ce cas a la propriété suivante:

- H(S) est maximale si tous les symboles $\{s1, s2, \ldots\ldots, s N\}$ de la source S sont équiprobables. Dans ce cas, l'entropie est égale à l'information associée à chaque message pris individuellement.

$$\forall\, i \in \{1,2, \ldots\ldots\ldots N\}, p(si) = \frac{1}{N} \leftrightarrow H(S) = log2(N)$$

Voici deux exemples de calcul d'entropie :

Nous avons :

1) A = $\{a_1, a_2, a_3, a_4\}$ et P (X) = { 1/4 , 1/4, 1/4, 1/4 }.

$$H(X) = \frac{1}{4} * 2 + \frac{1}{4} * 2 + \frac{1}{4} * 2 + \frac{1}{4} * 2 = \frac{8}{4} = 2$$

2) A = $\{a_1, a_2, a_3, a_4\}$ et P(X) = { 1/2, 1/4, 1/8, 1/8 }.

$$H(X) = \frac{1}{2} * 1 + \frac{1}{4} * 2 + \frac{1}{8} * 3 + \frac{1}{8} * 3 = \frac{7}{4} = 1.75$$

Exemple : Calcul de l'entropie d'une source binaire

Considérons le cas d'une source binaire S dont l'alphabet est {0,1} tel que P(1) = p et donc P(0)=1- p

$$0 < p \leq 1$$

$$H(S) = - p * log_2(p) - (1- p) \log_2(1 - p) = f(p)$$

La fonction f(p) est symétrique par rapport à p=0.5 et est maximale pour p=0.5

4. *Les caractéristiques du codage source :[11]*

- Nous considérons une source discrète, c'est-à-dire un dispositif qui fournit aléatoirement des séquences de symboles issus d'un ensemble discret fini. Une source peut être un texte, une image, ou plus généralement, tout signal numérique. Une source est modélisée par un ensemble de variables aléatoires, à valeur dans un alphabet de taille finie, Ω= {$x0$, ……x_N}. Ω est appelé l'ensemble des symboles de source.

- Une source est dite sans mémoire si la séquence de symboles générée par la source est une suite de variables indépendantes et identiquement distribuées.

- Un code de source C pour une variable aléatoire X de distribution de probabilité p, est une application de Ω vers l'ensemble des chaînes de symboles d'un alphabet D-aire A.

- Un code est dit non singulier si : $x_i \neq x_j \rightarrow C(x_i) \neq C(x_j)$ La non singularité assure le décodage d'un seul élément a la fois.

- L'ensemble des chaînes de symboles d'un alphabet D-aire A est noté A^+. En général, cet alphabet est binaire et nous avons D = 2, A = {0,1}. A^+ est alors l'ensemble des chaînes de caractères de taille finie formées de 0 et de 1, A^+ = { 0, 1, 00, 01, 10, 11, 000,….}. Un code associe à un symbole de source *x* un mot de code C(x). Ce mot de code est de longueur variable l(x), la longueur étant son nombre de bits. Ces codes sont appelés codes à longueur variable.

- L'espérance de la longueur d'un code C (ou longueur moyenne, selon la loi de probabilité de X) est donnée par:

$$L(C) = \sum_{x \in \Omega} p(x).\, l(x)$$

L(C) peut également se voir comme le taux de codage, c'est-à-dire le nombre moyen de bits codés par symbole de source.

- L'extension C^+ d'un code C est l'application de Ω^+ dans A^+, qui associe à une séquence de symboles de source la concaténation de ses mots de code:

$$C^+(x_0 x_1 \ldots\ldots x_N) = C(x_0)\, C(x_1) \ldots\ldots C(x_N)\ .$$

Cette définition est motivée par le fait que nous transmettons des séquences de symboles, et non des symboles isolés séparés par un symbole de séparation, ce qui serait inefficace.

- ***Régularité :*** Un code est dit régulier si tous les mots de code *mi* sont distincts. Tous les codes doivent au moins être réguliers pour permettre un décodage clair.

- ***Déchiffrabilité :*** Un code régulier est déchiffrable si pour toute suite *m*1 ,*m*2 , ...*mn* de mots de code il est possible de distinguer les *mi* sans ambiguïté et reconstruire ainsi les symboles *Si* correspondants.

- ***Codes instantanés :*** Nous disons d'un code qu'il est à décodage instantané s'il est possible de décoder les mots de code dès lors que tous les symboles qui en font partie ont été reçus.

❖ ***Extension d'un code :*** L'extension d'ordre n d'un code est le code formé par les séquences de n mots du code initial.

5. *Codage avec mots de longueur fixe :[12]*

Une manière simple de coder en binaire l'alphabet d'une source est d'attribuer à chaque symbole R bits. Il y a donc 2^R codes possibles et bien sûr nous avons la condition $2^R \geq K$ l'égalité étant possible lorsque le nombre K de symboles de la source est une puissance de 2. Dans le cas contraire nous aurons : $2^{R-1} < K < 2^R$. Cette dernière relation permet de déterminer le nombre R de bits nécessaires au codage de l'alphabet d'une source de K symboles:

$$R = Int[\ log_2(K)\ ] + 1$$

Nous avons :

$$\left.\begin{array}{l} H(X) \leq log2(K) \\ R \geq log_2(K) \end{array}\right\} \text{ et } \rightarrow R \geq H(X)$$

L'égalité a lieu lorsque tous les symboles de la source sont équiprobables et lorsque K est une puissance de 2.

- ✓ Un codage est dit d'autant plus efficace quand le nombre de codes possibles inutilisés est faible.

- ✓ L'efficacité dépend aussi de la quantité d'information moyenne de la source. L'efficacité η d'un codage sera ainsi définie par :

$$\eta = \frac{H(X)}{R}$$ elle est exprimée en %.

Exemple:

K= 24 ➔ $\log_2(K) = 4{,}585$ ➔ R = 5 ➔ $2^R = 32$ ➔ 8 codes non utilisés.
Si tous les symboles sont équiprobables : H(X) = 4,585 et η= 91,7 %.

6. *Codage par blocs, extension de la source :[12]*

Pour améliorer l'efficacité du codage, nous pouvons transmettre et donc coder les symboles non pas individuellement mais par blocs de J symboles. Cette technique est appelée l'extension de la source.

Exemple:

avec { A , B } nous pouvons faire avec J = 2 les blocs { AA , AB , BA , BB }.

A partir d'une source dite primaire, nous fabriquons une source secondaire.

Source primaire de K symboles ➔ extension ➔ source secondaire de K^J symboles

Si nous utilisons N bits de codage par bloc :

$N \geq \log_2(K^J) \rightarrow N \geq J \log_2(K)$ et $N = Int[J \log_2(K)] + 1.$

Le nombre de bits par symbole de la source primaire est $R = N / J = Int[\log_2(K)] + 1/J$ (R n'est plus un entier).

L'efficacité de codage de la source primaire est égale à :

$$\eta_2 = \frac{H(X)}{Int\,[\,log2\,(K)] + \frac{1}{J}}$$

résultat à comparer à celui obtenu sans extension de source :

$$\eta_1 = \frac{H(X)}{Int\,[\,log2\,(K)] + 1}$$

et, à cause du terme en 1/J en dénominateur de η_2, nous avons $\eta_2 > \eta_1$. La technique d'extension de source peut ainsi améliorer l'efficacité du codage pour des mots de codes de longueur fixe.

Du point de vue de la source étendue N est le nombre de bits attribués à un de ses symboles et l'efficacité du codage est égale à :

$$\eta_2 = \frac{H(X)}{R} = \frac{J\,H(X)}{N}$$

Tout se passe comme si la source étendue avait une entropie de J.H(X).

Exemple:

K= 24 , J = 3 ➔ $K^J = 13824$ ➔ N = 14 ➔ $2^N = 16384$ ➔ R = 4,666.

Si tous les symboles sont équiprobables : H(X) = 4,585 et $\eta_2 = 98{,}25\ \%$.

7. Codes uniquement décodables :[13]

Le but du codage de source sans pertes est de comprimer les données de telle façon que nous puissions reconstruire parfaitement (sans pertes, sans erreur) la source au destinataire. Pour cela, il faut que le décodage ait lieu sans ambiguïté, c'est à dire qu'une séquence codée donnée doit être interprétable de façon unique comme une succession (concaténation) de mots de codes déterminés.

Un code permettant un tel décodage (sans ambiguïté) est qualifié d'uniquement décodable (u.d.). Formellement, cela signifie que si nous avons deux séquences de mots de code concaténés identiques :

$$C_1C_2 \cdots C_k = C'_1\, C'_2 \cdots C'_l$$

où les Ci et C'j appartiennent au code ζ, alors les mots de code sont un à un identiques :

$$k = l \text{ et } Ci = C'i\ ,\ i = 1, \ldots, k.$$

8. Codes instantanés et condition du préfixe :[13]

Un code instantané est caractérisé par la condition du préfixe : Aucun mot de code n'est le préfixe d'un autre mot de code (c'est-à-dire aucun Ci ne débute un Cj , j≠ i).

Il est facile de démontrer que la condition du préfixe caractérise bien un code instantané : cette condition est d'abord clairement suffisante, sinon un ci débuterait un autre Cj et il y aurait ambiguïté lorsque le décodeur lit Ci, de sorte qu'il ne pourrait pas conclure instantanément.

Réciproquement, avec la condition du préfixe le décodage est clairement instantané, car dès qu'un mot de code Ci est lu, il peut être décodé sans ambiguïté.

Un code instantané est aussi appelé code à préfixe (prefix code) dans la littérature. Pour résumer le vocabulaire vu jusqu'ici nous avons les ensembles emboîtés suivants :

$$\{codes\ VLC\} \supset \{\ codes\ inversibles\} \supset \{codes\ u.d\} \supset \{\ codes\ instantanés\}.$$

Voici un exemple qui montre le rôle des codes préfixés :

Dans cet exemple nous utilisons trois codages possibles pour une source de 4 symboles. Ces exemples sont dans le tableau ci-dessous:

Caractères	Probabilités	Code I	Code II	Code III
I	½	1	0	0
B	¼	00	10	01
F	1/8	01	110	011
O	1/8	10	111	111

Tableau2.1 : Exemple de codage préfixé.

Supposons que nous cherchions à transmettre le message « BOF » :

1. Avec le code I, le message envoyé est : 001001. C'est ce que voit le récepteur. Comment peut-il l'interpréter? De manière correcte bien sûr mais aussi 00 1 00 1 c'est à dire BIBI. Problème, le message n'est pas décodable de manière unique. Ceci est du au fait que le 1, code attribué à I est le début d'un autre code 10 attribué au O. Pour éviter cette situation, il ne faut pas qu'un code soit le "préfixe" d'un autre code. Les codes qui remplissent cette condition sont des codes préfixes.

2. Avec le code III, le message envoyé est 01111011. Au décodage nous pouvons voir 0 111….c'est à dire IO…Mais ici nous nous rendons compte du fait que ce qui suit c'est à dire soit 1, soit 10, soit 101 ne sont pas des codes et donc, nous pouvons revenir en arrière pour modifier l'interprétation soit 01 111 011 et retrouver le bon message. Le code n'est pas décodable de manière instantanée. Ceci est aussi dû au fait que le code utilisé n'est pas un code préfixe.

3. Le code II est lui un code préfixe et nous avons les deux propriétés souhaitées: décodable de manière unique et de manière instantanée.

III. Inégalité de Kraft-McMillan :

III.1. Définition :[13]

Pour trouver le meilleur code pour une source donnée, il faut minimiser le taux R sous la contrainte que le code soit uniquement décodables. Afin de réaliser cette optimisation, on caractérise d'abord le fait qu'un code soit uniquement décodable sur la distribution des longueurs:

$$\sum_{i=1}^{M} 2^{-li} \leq 1$$

Cette formule est appelée inégalité de Kraft-McMillan.

Preuve:

Pour un code uniquement décodable, toute séquence de l bits peut se décomposer d'au plus une façon comme concaténation de mots de codes $C_{i1}\, C_{i2} \cdots C_{ik}$ où $li_1 + li_2 + \cdots + li_k = l$. Cela vient directement de la définition d'un code uniquement décodable.

Le nombre total N_l (k) de concaténations possibles de k mots de codes donnant une séquence codée de longueur totale l bits ne peut donc pas dépasser le nombre total de séquences de l bits, qui est 2^l. (Sinon, il y aurait forcément ambiguïté.) Nous avons donc l'inégalité : N_l (k) $\leq 2^l$.

Mais par ailleurs, si nous développons la puissance kième (produit de k facteurs identiques)

$$(\sum_{i=1}^{M} x^l)^k = \sum_{i1,i2....ik} x^{li1+li2+\cdots+lik}$$

et si nous regroupons les termes de même puissance $l = l_{i1} + l_{i2} + \cdots + l_{ik}$ nous trouvons précisément N_l (k) puissances x^l pour chaque *l*. Ainsi :

$$(\sum_{i=1}^{M} x^{li})^k = \sum_{l} Nl(k)x^l$$

La somme au second membre va de $l = kl_{min}$ à $l = kl_{max}$ où l_{min} et l_{max} désignent les longueurs minimale et maximale d'un mot de code.

Nous concluons facilement en faisant $x = 1/2$ et $k \to \infty$: nous voyons par exemple que

$$\sum_{i=1}^{M} 2^{-li} \leq \sqrt[k]{k(lmax - lmin) + 1} \to 1$$

quand $k \to \infty$.

III.2. Algorithme de Kraft :[13]

Pour l'instant l'inégalité de Kraft-McMillan n'est qu'une condition nécessaire pour qu'un code soit uniquement décodable. Mais il y a une réciproque :

« *Si l'inégalité de Kraft-McMillan est vérifiée, alors il existe un code uniquement décodable, et même instantané, qui admette {l_1, l_2, . . . , l_M} comme distribution de longueurs ».*

Voici une preuve dont l'intérêt est de fournir un algorithme simple qui fournit un code instantané {C_1, . . . ,C_M} à partir d'une distribution {l_1, l_2, . . . , l_M} de longueurs vérifiant l'inégalité de Kraft-McMillan.

Preuve:

Nous se donnons une distribution de longueurs $l_1 \leq l_2 \leq \ldots \leq l_M$ vérifiant l'inégalité de Kraft-McMillan1. A chaque mot de code C_i nous associons le nombre $\overline{C_i} = 0,C_i \in [0, 1[$ dont les décimales de l'écriture en base 2 est formée des bits de Ci . Nous notons I_i l'intervalle $I_i = [\overline{C_i} ; \overline{C_{i+2}} + 2^{-li} [$.

Par exemple, C_i= 010 donne $\overline{C_i} = \overline{0,010} = 1/4$. et $I_i = [\overline{0,010}; \overline{0,011}[= [1/4 ; 3/8 [$ est l'ensemble des nombres de [0;1[dont les décimales en base 2 commencent par C_i .

Clairement, C_i détermine l'intervalle I_i, et réciproquement : il suffit de couper le développement binaire de la limite gauche de l'intervalle (de longueur Li) à $\log_2 \frac{1}{Li}$ bits pour retrouver C_i .

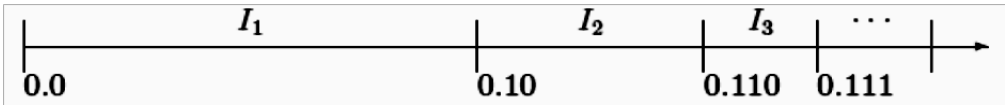
Nous pouvons alors traduire le fait qu'un code soit instantané sur les intervalles :

Aucun C_i ne débute un C_j si et seulement si $\overline{C_j} \notin I_i$. Autrement dit le code est instantané si et seulement si les I_i sont des intervalles disjoints.

L'inégalité de Kraft-McMillan revient précisément à dire que la somme des longueurs des I_i est ≤1 S'ils sont disjoints, nous pouvons donc les mettre bout-à-bout en restant dans le segment [0,1[.

Nous en déduisons facilement un algorithme de construction du code instantané :

Nous mettons bout-à-bout les intervalles I_i, classés par longueurs décroissantes (l_i croissantes) dans le segment [0,1[en partant de la gauche :



Nous commençons donc par $\overline{C_1} = \overline{0.0. \ldots 0}$, et nous posons $\overline{C_{i+1}}$ = extrémité droite de I_i à chaque étape. Cela revient, pour chaque étape, à calculer C_i+1 (addition binaire) puis à compléter avec des zéros si nécessaire pour obtenir C_{i+1} de longueur $l_{i+1} > l_i$. Les intervalles sont disjoints, le code est donc instantané.

Voici un exemple :

l_i	Code
1	0
2	10
3	110
4	1110
5	11101
5	11110
6	111110
6	111111

Tableau2.2 : Exemple de code instantané.

Avec cet algorithme nous détectons automatiquement si les l_i ne vérifient pas l'inégalité de Kraft-McMillan : on « dépasse » alors l'extrémité droite du segment [0,1[, et nous ne pouvons plus continuer. Par exemple :

l_i	**Code**
1	0
2	10
3	110
4	1110
5	11110
5	11111
6	Erreur

$$\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\frac{1}{32}+\frac{1}{32}+\frac{1}{64}>1$$

Une conséquence importante de la caractérisation par l'inégalité de Kraft- McMillan est que tout code uniquement décodable peut être remplacé par un code instantané de même distribution de longueurs et donc de même taux. On peut donc limiter la recherche du meilleur code à l'ensemble des codes instantanés. Donc du coup, on pourra toujours choisir un code optimal pour lequel le décodage est instantané, donc très simple à implanter.

IV. Recherche du code Optimal :[13]

D'après ce qui précède, pour trouver le meilleur code pour une source donnée, il suffit de minimiser le taux R sur la distribution de longueurs uniquement, avec la contrainte donnée par l'inégalité de Kraft-McMillan :

$$\min\{ R = \sum_i pi\ li \mid \sum_i 2^{-l_i} \leq 1 \}$$

Nous reconnaissons un problème classique qui se résout par la méthode des multiplicateurs de Lagrange.

Appliquant la méthode du Lagrangien ; celui-ci s'écrit

$$L = \sum_i pi\ li - \lambda \sum_i 2^{-l_i}$$

où λ¸ est le multiplicateur de Lagrange. Nous devons annuler les dérivées

$$\frac{\partial L}{\partial li} = pi - \lambda' 2^{-li} = 0$$

où ¸λ' est toujours constant (indépendant de i). Par conséquent, l'optimum est atteint lorsque les p_i et 2^{-li} sont proportionnels, pour une contrainte saturée $\sum_i 2^{-li} = 1$. La constante de proportionnalité vaut forcément ¸λ'= 1 ; nous trouvons donc que R est minimisé lorsque

$$l_i = \log_2 \frac{1}{pi'}$$

auquel cas nous découvrons avec stupéfaction que le taux minimal est l'entropie de la source :

$$H = \sum_{i=1}^{M} pi \; log2 \; \frac{1}{pi}$$

Seulement voilà, c'était trop beau : ce que nous venons de faire est faux, puisque ce résultat ne donne pas, en général, des longueurs l_i entières ! Il fallait tenir compte de cette contrainte supplémentaire, et la méthode du Lagrangien était donc inadaptée.

V. Arbre binaire de codes préfixés :[12] [7]

Un codage binaire peut être représenté de manière graphique par un arbre. Les arbres sont aussi des représentations commodes pour écrire les algorithmes de codage et de décodage.

Règles de construction d'arbre :

- ✓ Un déplacement à gauche correspond à un "0".
- ✓ Un déplacement à droite correspond à un "1".
- ✓ Chaque déplacement crée un nœud de l'arbre.
- ✓ Chaque nœud à un père (vers le haut) et peut avoir deux fils (vers le bas).
- ✓ Le lien entre deux nœuds est une branche.
- ✓ Un nœud qui n'a pas de fils est une feuille.

Donc nous déduisons cette définition : un code préfixe est un code dont les symboles codés sont des feuilles.

Les deux algorithmes (Shannon-Fano et Huffman) construisent chacun un codage optimal pour la compression mais de manière différente : nous aboutissons à des codes différents, mais au même taux de compression.

Pour construire ce code, les deux algorithmes utilisent un arbre binaire qui fabrique automatiquement un code préfixé. C'est tout simple : un arbre binaire est un arbre dont les nœuds sont soit réduits à des feuilles (pas de sous-branches), soit comptent exactement deux branches. Les branches sont marquées 0 et 1.

Voici un exemple d'arbre binaire, les nœuds sont les bulles, les feuilles contiennent les symboles:

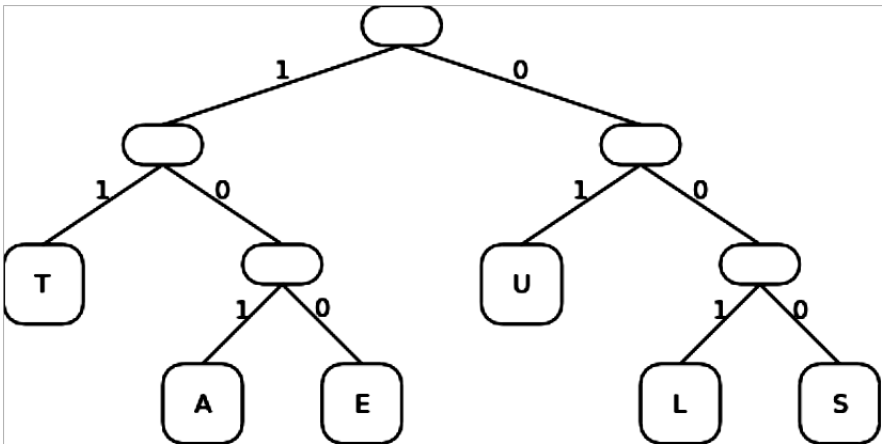


Figure2.2 : Exemple d'arbre binaire.

Pour garantir de manière sûre que c'est bien un codage préfixé, il suffit d'interdire que les nœuds portent des symboles : seules les feuilles peuvent être des symboles codés.

Pour obtenir le code d'un symbole, il suffit de suivre les branches en partant de la racine. Par exemple, L = 001.

Il faut que l'arbre binaire soit le même entre le codeur et le décodeur.

VI. Types de codage entropique :

VI.1. Les algorithmes de codage statistique :

Le principe des algorithmes de codage statistique est d'utiliser les probabilités d'occurrence de chaque symbole dans une séquence de symboles émanant de la source.

VI.1.1. Codage de Shanon - Fano :[14] [11]

Les étapes de ce codage sont :

- Ordonner les signes *Si* à encoder c'est-à-dire les probabilités d'apparition de chaque symbole sont placées dans un tableau trié par ordre décroissant.
- Diviser l'ensemble de ces signes ainsi ordonnés en Q=2 sous ensembles ordonnés, aussi équiprobables que possible c'est-à-dire le tableau est coupé en deux groupes de symboles S_0 et S_1 dont la somme des probabilités de chaque groupe avoisine 0.5.
- Attribuer à chaque sous ensemble un symbole *binaire* d'encodage distinct, le groupe S_0 est codé par un "0" et S_1 par un "1".
- Si un groupe S_i n'a qu'un seul élément, c'est une feuille terminale sinon répéter la même procédure à partir de l'étape 2 pour chaque sous ensemble jusqu'à ce que chaque signe possède un code *binaire* distinct.

Exemple : Application de l'algorithme de Shannon-Fano

Pour illustrer cet algorithme, nous allons coder la phrase suivante : "Le codage est indispensable"

Pour simplifier nous n'allons pas prendre en compte le symbole espace (blanc).

Cette phase est une source de 24 symboles

Tous ces symboles émanent de l'alphabet $A=\{E , S , A , D , I , N , L , B , G , P , T , O , C\}$. Cet Alphabet a N=13 symboles.

- ✓ ETAPE 1 :

Symbole	Nombre de fois	Probabilités
E	5	5/24
S	3	3/24
A	2	2/24
D	2	2/24
i	2	2/24
N	2	2/24
L	2	2/24
B	1	1/24
G	1	1/24
P	1	1/24
T	1	1/24
O	1	1/24
C	1	1/24

Tableau2.3 : Etape 1 de l'algorithme de Shannon-Fano.

- ✓ Application successive des étapes 2,3 et 4.

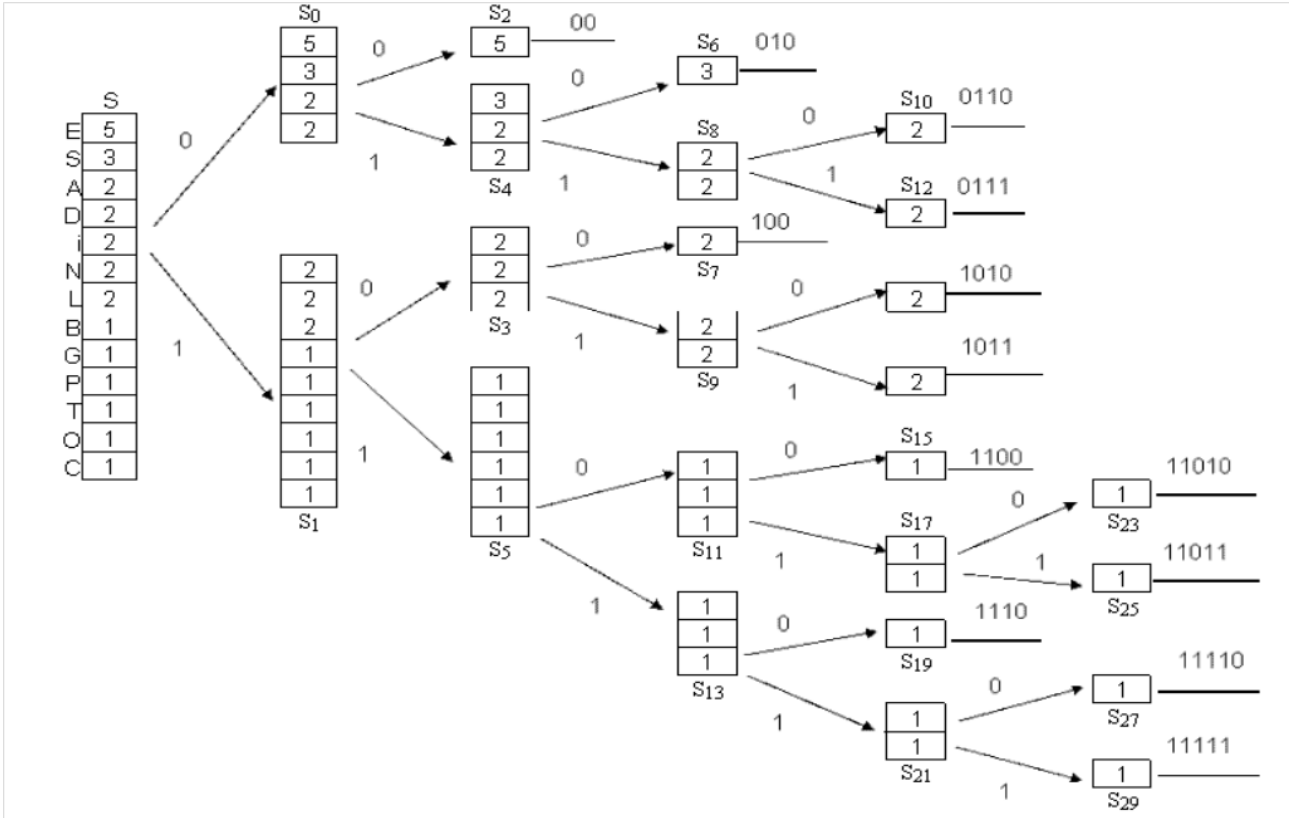


Figure2.3 : L'arbre de Shanon- Fano de l'exemple donné.

VI.1.2. Algorithme de Huffman :[7]

Dans l'algorithme de Huffman, nous procédons différemment pour un résultat comparable. Nous commençons par choisir les deux symboles qui ont le moins d'occurrences, nous leur donnons les codes 0 et 1 et nous les regroupons dans un arbre binaire auquel nous attribuons un nombre d'occurrences : la somme des deux symboles qu'il regroupe. Ensuite, nous continuons : nous choisissons deux symboles ou arbres qui ont le moins d'occurrences et nous les regroupons. Nous regroupons ainsi deux par deux des symboles ou des arbres en fonction du nombre d'occurrences qu'ils représentent.

Exemple de codage Huffman :

Soit la source S = "CITRONTRESCONTRIT".

- Nous commençons par l'ensemble : { C/2, I/2, T/4, R/3, O/2, N/2, E/1, S/1}.
- L'algorithme groupe les deux symboles ou arbres les moins représentés : E/1 et S/1 et crée un arbre pour eux : {E, S}/2. Dans cette notation, nous mettons la branche 0 à gauche, la branche 1 à droite.
- Nous continuons donc avec { C/2, I/2, T/4, R/3, O/2, N/2, {E, S}/2 }. Nous avons le choix pour grouper les moins représentés : le C et le I, le O avec le ES, etc. Le premier choix donne : {{C, I}/4, T/4, R/3, O/2, N/2, {E, S}/2}.
- Nous continuons et nous obtenons {{C, I}/4, T/4, R/3, {O, N}/4, {E, S}/2}.
- Un nouveau groupement du R et ES donne { {C, I}/4, T/4, {R, {E, S}}/5, {O, N}/4 }.
- Nous continuons : {{{C, I}, T}/8, {R, {E, S}}/5, {O, N}/4}
- Encore : {{{C, I}, T}/8, {{R, {E, S}}, {O, N}}/9 }.
- Et la dernière étape car il ne reste plus que deux sous-groupes : {{{C, I}, T}, {{R, {E, S}}, {O, N}}}/17.

A lire ainsi : nous obtenons donc les codes suivants :

C = 000, I = 101, T = 01, R = 100, O = 110, N = 111, E = 1010, S = 1011.

Le texte de la source S est codé = "00010101100110111011001010101100011011101100101 01" sur 49 bits.

Remarques finales sur ces deux algorithmes :

En principe, le codage de Huffman donne toujours le meilleur résultat possible en termes de compression. Avec ces algorithmes, les données compressées doivent être accompagnées de la table ou arbre de codage. La compression n'est rentable que si la représentation des deux : données et arbre est plus petite que les données d'origine. On peut imaginer que pour des données similaires, par exemple des articles de journaux en français, on puisse avoir la même table de codage basée non pas sur les statistiques d'apparition des lettres et non pas une table de codage par article. Nous y perdrions sur certains articles mais nous éviterions de stocker une table dans chaque article. Ces deux méthodes travaillent sur des données totalement aléatoires

VI.2. Le code Unaire (Unary code) : [20]

Le code à longueur variable le plus simple et le plus intuitif à lequel nous pouvons penser est le code unaire. Le code unaire pour entier positif N est constitué d'une suite de N bits à « 1 » suivi d'un bit à « 0 » ou bien le code inverse avec une suite de N bits à 《 0 》suivi d'un bit à « 1 》. La taille du code pour chaque entier N est (N+1) bits, le tableau suivant illustre le code unaire.

N	***Code***	***Code inverse***
0	0	1
1	10	01
2	110	001
3	1110	0001
4	11110	00001
5	111110	000001
…	…	…
I	11111…10	00000…01

Tableau2.4 : Exemple de codage unaire.

L'inconvénient de ce code est que sa longueur est exponentielle. Si en prend les caractères ASCII, le premier code est « 0 » et le 256éme est (111…10) c'est-à-dire 257 bits (33octets).

VI.3. Codage arithmétique :

D'autres systèmes de codage de source sans pertes ont été proposés pour prendre en compte les dépendances temporelles (d'un symbole à l'autre) de la source (avec mémoire). Ces systèmes de codage permettent de coder une source quelconque sans connaitre a priori ses statistiques, mais sont plus complexes à mettre en œuvre

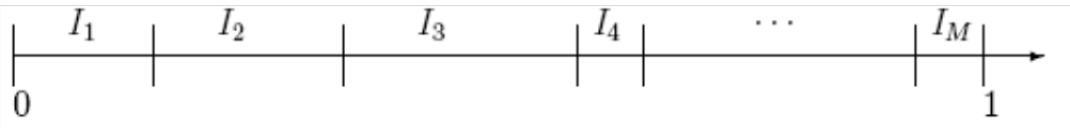
Le codage arithmétique est une extension itérative d'une technique de codage connue depuis les années 50, appelée codage d'Elias (ou de Shannon- Fano-Elias). Cette dernière technique a été simplement introduite comme moyen de construction d'un code VLC « scalaire », où nous codons chaque symbole de source l'un après l'autre.

VI.3.1.Codes d'Elias :[15]

La distribution de probabilité des symboles de source vérifie la relation :

$$\sum_{i=1}^{M} pi = 1$$

Nous pouvons donc construire une partition (un « découpage ») du segment [0, 1[(de longueur 1) en intervalles contigus $I_1, \ldots, I_M$, où chaque I_i est de longueur p_i. Dans la suite, chaque point du segment [0, 1[est un nombre qui sera représenté par son développement binaire (en base 2), par exemple :
0.110100100 . . ..



Peter Elias a proposé trois principaux algorithmes :

VI.3.1.1. Code Gamma γ(N) : [27]

Pour tout entier positif N, le processus de codage et de décodage est universel et se fait d'une manière très simple.

❖ **Le Codage**

La construction du code se fait selon l'algorithme suivant :

1. Chercher le plus grand nombre M tel que, $2^M \leq N < 2^{M+1}$, $N = 2^M + L$ avec la longueur de L est au plus M bits.

2. Coder le nombre M avec le code unaire, M《 0 》suivis par « 1 》ou M《 1 》suivis par « 0 ».

3. Concaténer les deux codes. M.L+

Exemple : Voici la liste des codes Gamma pour les 16 premiers entiers

$1 = 2^0 + 0 = 1$ (M=0, L=0)	$9 = 2^3 + 1 = 0001\ 001$
$2 = 2^1 + 0 = 01\ 0$ (M=1, L=0)	$10 = 2^3 + 2 = 0001\ 010$
$3 = 2^1 + 1 = 01\ 1$ (M=1, L=1)	$11 = 2^3 + 3 = 0001\ 010$
$4 = 2^2 + 0 = 001\ 00$	$12 = 2^3 + 4 = 0001\ 100$
$5 = 2^2 + 1 = 001\ 01$	$13 = 2^3 + 5 = 0001\ 101$
$6 = 2^2 + 2 = 001\ 10$	$14 = 2^3 + 6 = 0001\ 110$
$7 = 2^2 + 3 = 001\ 11$	$15 = 2^3 + 7 = 0001\ 111$
$8 = 2^2 + 0 = 0001\ 000$	$16 = 2^4 + 0 = 00001\ 0000$

Tableau2.5 : Exemple de codage Gamma.

❖ **Le Décodage**

Pour le processus de décompression il est aussi simple et se fait seulement en deux étapes :

1. Lire les zéros « 0 》depuis le code jusqu'à rencontrer le bit « 1 》, trouver le nombre de « 0 》et noter M.

2. Lire les M bits comme un entier L. et calculer $N = 2^M + L$

Il est facile de voir que ce code peut être employé pour coder des nombres entiers positifs même dans les cas où le plus grand nombre entier n'est pas connu à l'avance. En outre, ce code se développe lentement, ainsi c'est un bon candidat pour la compression des nombres entiers où les petits nombres entiers sont communs et le grand est rare.

VI.3.1.2. Code Delta δ(N) :

Elias a proposé un autre algorithme de construction d'un code VLC qui est le code δ (delta). Ce code peut se construire de deux manières déférentes :

- ✓ Par l'ajout des longueurs en binaire
- ✓ Ou par l'emploi de code Gamma précédent

Ainsi, le code Delta d"Elias, aussi pour les nombres entiers positifs, est légèrement plus complexe pour le processus de construction.

- ❖ **Le Codage**

Pour le première cas, Ajout de longueurs en binaire :

Dans son code gamma, Elias ajoute la longueur du code unaire (α). Dans son prochain code, δ (delta), il ajoute la longueur en binaire (β) Alors la construction de cet algorithme se déroule comme suit :

1. Écrire le nombre a codé N en binaire. Le bit (plus significatif) le plus à gauche sera un 1.
2. Compter les bits, enlever le bit le plus à gauche de N, et ajouter au début le compte en binaire, à ce qui reste du nombre N après que son bit le plus à gauche a été enlevé.
3. Soustraire 1 du compte de l'étape 2 et ajouter ce nombre de zéros au début code.

Exemple 1 : Quand ces étapes sont appliquées au nombre entier 17, voici les résultats :

Ecrire le nombre 17 en binaire : $17 = 10001_2$ (5bits).
Enlever le 1 le plus à gauche et ajouter $5 = 101_2$ nous aurons : 101|0001.
Trois bits étaient supplémentaires, ainsi nous ajoutons deux zéros pour obtenir le code de delta : 00|101|0001.

Pour le deuxième cas, Avec l'emploi de code Gamma : l'algorithme est fait selon les étapes suivantes :

1. Chercher le plus grand nombre M tel que, $2^M \leq N < 2^{M+1}$, $N=2^M+L$ avec la longueur de L est au plus M bits.
2. Coder le nombre M+1 avec le code gamma d'Elias.
3. Concaténer ensuite les deux codes M.L

Exemple 2:

N=17, le résultat est $17= 2^4+1$. Le code gamma de M+1=5 est 00101 puis concaténer avec L=0001 pour donner N=00101 0001.

Le tableau suivant illustre les codes Delta pour les 16 premiers entiers:

$1=2^0+0$ =1 (M=0, L=0)	$9=2^3+1$ = 00100 001
$2=2^1+0$ =010 0 (M=1, L=0)	$10=2^3+2$ =00100 010
$3=2^1+1$ =010 1 (M=1, L=1)	$11=2^3+3$ =00100 011
$4=2^2+0$ =011 00	$12=2^3+4$ =00100 100
$5=2^2+1$ =011 01	13=2+5 =00100 101
$6=2^2+2$ =011 10	$14=2^3+6$ =00100 110
$7=2^2+3$ =011 11	$15=2^3+7$ =00100 111
$8=2^3+0$ =00100 000	$16=2^4+0$ =00101 0000

Tableau2.5 : Exemple de codage Delta.

❖ **Le Décodage**

Le décodage est fait selon les étapes suivantes :

1. Lire la chaine de gauche à droite jusqu'à reconnaitre un code gamma qui représente M+1.
2. Lire les M bits suivants qui représentent le nombre L.
3. Calculer $N=2^M+L$.

VI.3.1.3. Code Omega :

Comme le codage gamma et le codage delta, le codage Omega permet de coder tous les entiers naturels, à l'exception de zéro, sans qu'il y ait besoin de connaitre au préalable l'intervalle des valeurs à coder (contrairement, par exemple, au codage binaire).

Pour cela, le codage Omega fait précéder la représentation binaire du nombre par sa longueur (en nombre de bits), comme le font les codages précédents. Comme l'intervalle des valeurs pouvant être prise par cette longueur n'est pas non plus connu à l'avance, elle doit être codée avec un code préfix. Contrairement au codage gamma qui utilise un codage unaire à cet effet, ou au codage delta qui utilise un codage gamma, le codage Omega utilise un codage Omega. Ce codage est donc un codage récursif.

❖ **Le Codage**

Le codage d'un nombre entier positif N est fait périodiquement dans les étapes suivantes :

1. Initialiser le code-ainsi-loin à 0.
2. Si le nombre à coder est 1, arrêter ; Autrement, ajouter au début la représentation binaire de N au code-ainsi-loin (à gauche de 0). Supposer que nous avons ajouté L bits au début.
3. Répéter l'étape 2, avec la représentation binaire de L−1 au lieu du N.

Exemple 1 : Le nombre entier 17 par exemple est donc codé par :

1) un 0 simple,

2) ajouté au début, la représentation binaire de 17 qui est 10001 sur 5 bits,

3) ajouté au début la représentation binaire de 5−1 qui est 100 sur 3 bits,

4) ajouté au début la représentation binaire de 3−1 qui est 10 sur 2 bits,

5) arrêté car 2−1 = 1 Le résultat est 10|100|10001|0.

Le tableau suivant énumère les 16 premiers codes d'Omega :

1= 0	9= 11 1001 0
2=10 0	10=11 1010 0
3= 11 0	11=11 1011 0
4= 10 100 0	12= 11 1100 0
5= 10 101 0	13= 11 1101 0
6= 10 110 0	14= 11 1110 0
7= 10 111 0	15= 11 1111 0
8= 11 1000 0	16= 10 100 10000 0

Tableau2.6 : Exemple de codage Omega.

- **Le décodage**

Le décodage est fait dans plusieurs étapes non récurrentes où chaque étape lit un groupe de bits du code. Un groupe qui commence par un zéro signale la fin du décodage.

1. Initialiser N à 1
2. Lire le prochain bit. S'il est un 0, arrêter. Autrement lire N bit de plus, affecter le groupe de N + 1 bits à N, et répéter cette étape.

Exemple 2: Décoder 10|100|10001|0.

Le décodeur initialise N = 1 et lit le premier bit. C'est un 1, ainsi il lit N = 1 bit de plus (0) et assigne N = 10_2 = 2. Il lit le prochain bit. C'est un 1, ainsi il lit N = 2 bits de plus (00) et assigne le groupe 100 à N. Il lit le prochain bit. C'est un 1, ainsi il lit quatre bits de plus (0001) et assigne le groupe 10001 à N. Le prochain bit lu est 0, indiquant la fin du décodage. Donc la valeur de N est N= 10001_2 = 17.

- **Taille des codes d'Elias**

Le tableau suivant montre la taille des différents codes (Gamma, Delta et Omega) :

Valeur	Code Gamma	Code Delta	Code Omega
1	1	1	1
2	3	4	3
3	3	4	3
4	5	5	6
5-7	5	5	6
8-15	7	8	7
16-31	9	9	11
32-63	11	10	12
64-127	13	11	13
1000	19	16	17
104	27	20	21
105	33	25	28

Tableau2.7 : Comparaison de la taille des codes Gamma, Delta et Omega.

VI.3.2. Le codage de Golomb :[27]

Le codage de Golomb est un codage entropique inventé par Solomon Wolf Golomb en 1966 et utilisé essentiellement en compression de données. Le code produit est un code préfixe.

Principe :

Le codage de Golomb d'un entier naturel N dépend d'un paramètre k et se fait en deux étapes :

1. le codage du quotient de la division euclidienne de N par k avec un codage unaire ;
2. le codage du reste de la même division avec un codage binaire tronqué.

Mathématiquement, pour coder un entier $N, N \in \mathbb{N}, N = q \times k + r$, nous codons d'abord $q = \lfloor N/k \rfloor$en unaire, puis $r = N - k \times q$en binaire tronqué.

Optimalité :

Le codage de Golomb est adapté pour des données dans lesquelles les valeurs les plus faibles sont plus probables que les autres (mais où les autres peuvent malgré tout apparaitre).

Exemples :

Représentation des premiers entiers naturels avec un codage de Golomb								
Décimal	**Binaire**	**Code de Golomb k = 1 (unaire)**	**Code de Golomb k = 2**	**Code de Golomb k = 3**	**Code de Golomb k = 4**	**Code de Golomb k = 5**	**Code de Golomb k = 10**	**Code de Golomb k = 16**
0	0000	0	0 0	0 0	0 00	0 00	0 000	0 0000
1	0001	10	0 1	0 10	0 01	0 01	0 001	0 0001
2	0010	110	10 0	0 11	0 10	0 10	0 010	0 0010
3	0011	1110	10 1	10 0	0 11	0 110	0 011	0 0011
4	0100	11110	110 0	10 10	10 00	0 111	0 100	0 0100
5	0101	111110	110 1	10 11	10 01	10 00	0 101	0 0101
6	0110	1111110	1110 0	110 0	10 10	10 01	0 1100	0 0110
7	0111	11111110	1110 1	110 10	10 11	10 10	0 1101	0 0111
8	1000	111111110	11110 0	110 11	110 00	10 110	0 1110	0 1000
9	1001	1111111110	11110 1	1110 0	110 01	10 111	0 1111	0 1001
10	1010	11111111110	111110	1110 10	110 10	110 00	10 000	0 1010

Tableau2.8 : Exemple de codage de Golomb.

Utilisations :

Le codage de Golomb est principalement utilisé dans sa variante dite codage de Rice qui peut être implémentée de façon plus efficace. Un codage de Rice est d'ailleurs équivalent à un codage de Golomb dont le paramètre est 2 élevé à la puissance du paramètre de Rice.

Itérations du codage arithmétique :[13]

L'idée du codage arithmétique est d'itérer la procédure d'Elias au fur et à mesure du temps, afin de coder plusieurs symboles de source successifs. Nous prendrons ainsi en compte les corrélations éventuelles des symboles de la source.

Commençons par le codage de deux symboles consécutifs (x_1, x_2). La distribution de probabilité conjointe de ce vecteur s'écrit :

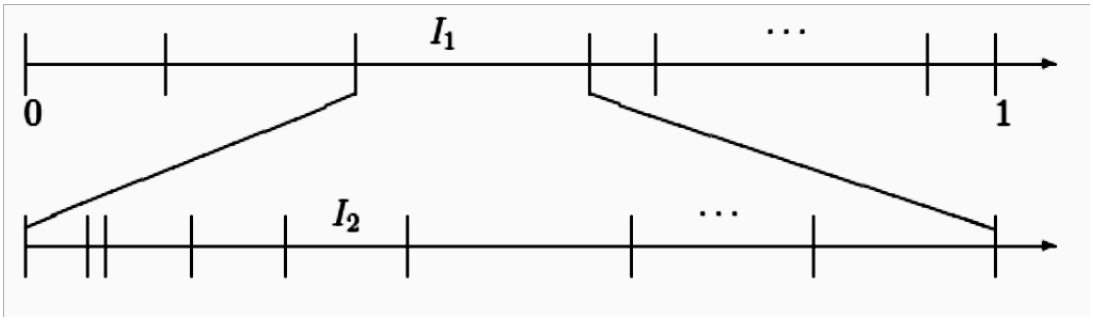
$$p(x_1, x_2) = p(x_1)p(x_2|x_1)$$

où $p(x_2|x_1)$ est la probabilité conditionnelle de x_2 sachant x_1.

Nous commençons par appliquer la procédure d'Elias pour le symbole x_1 ; nous obtenons une représentation codée dans un intervalle I_1 de longueur $p(x_1)$. Pour tenir compte de x_2, nous considérons non pas les probabilités $p(x_2)$, mais les probabilités conditionnelles $p(x_2|x_1)$ qui vérifient la relation :

$$\sum_{x2} p(x2|x1) = 1$$

Nous recommençons alors la procédure d'Elias, non plus sur le segment [0,1[, mais sur l'intervalle I_1 que nous repartitionnons en intervalles I_2 plus fins, de longueurs proportionnelles aux $p(x_2|x_1)$:



Puisque $l(I_1) = p(x_1)$, ces nouveaux intervalles I_2 sont en fait de longueur
$l(I_2) = p(x_1)p(x_2|x_1) = p(x_1,x2)$.

Arrivé jusque là, rien n'empêche de continuer et de coder le symbole suivant x_3, en redécoupant l'intervalle I_2 en sous-intervalles I_3 de longueurs proportionnelles à $p(x_3|x_1,x_2)$. Ces intervalles seront en fait de longueur :

$$p(x_1, x_2)p(x_3|x_1,x_2) = p(x_1,x_2,x_3)$$

et ainsi de suite. Après n itérations de l'algorithme d'Elias on obtient un intervalle de longueur :

$$p(x_1,x_2, \ldots ,x_n) = p(x_1)p(x_2|x_1)p(x_3|x_1,x_2) \cdots p(x_n|x_1, \ldots ,x_{n-1})$$

qui prend bien en compte la dépendance temporelle des symboles de source.

Adaptabilité et universalité :[13]

L'intérêt du codage arithmétique n'est pas tant de fournir un algorithme de codage optimal mais il permet surtout un codage adaptatif : plutôt que de supposer connu une fois pour toutes la distribution de probabilité conjointe de la source $p(x_1, \ldots, x_n)$ (par modélisation et/ou estimation sur toute la source), il est possible d'estimer au fur et à mesure les probabilités conditionnelles $p(x_n|x_1, \ldots, x_{n-1})$ dont nous avons besoin pour l'étape n. Ceci permet une plus grande flexibilité pour coder n'importe quel type de source, en s'adaptant à des variations éventuelles de statistiques.

Cette estimation est d'abord brute; elle s'affine au fur et à mesure du codage des données x_1, x_2, ...

Nous pouvons même aller plus loin : plutôt que de transmettre en données annexes ces estimations de probabilité du codeur au décodeur (qui en a également besoin), nous pouvons envisager de répéter au décodage (indépendamment du codeur) l'estimation voulue sur les données au fur et à mesure de leur décodage. Nous obtenons alors un algorithme de décodage universel qui est capable de coder (et de décoder sans information supplémentaire) tout type de source sans connaissance a priori de ses statistiques.

VII. Nécessité d'un codage vectoriel :[13]

Nous avons vu que le taux de codage d'un code de Fano-Shannon, et donc a fortiori du meilleur code (de Huffman) vérifie l'inégalité $H \leq R \leq H + 1$.Mais pouvons-nous améliorer cet encadrement? Autrement dit, pouvons-nous trouver une famille de codes pour lesquels, quelque soit la distribution de probabilité de la source, le taux ne s'éloigne pas de l'entropie plus qu'une quantité < 1 ?

La réponse est non, comme le montre l'exemple suivant d'une source binaire (M = 2) :

x_i	p_i	l_i	C_i
1	E	1	1
0	1- ε	1	0

Ce code (qui consiste à ne pas coder du tout) est optimal : nous ne pouvons évidemment pas faire mieux (pas de longueur nulle pour un code uniquement décodable) :
R = 1. Mais l'entropie vaut5

$$H = H2\,(\varepsilon) = \varepsilon\, log2\, \tfrac{1}{\varepsilon} + (1-\varepsilon) log2\, \tfrac{1}{1-\varepsilon} \to 0 \qquad quand\ \varepsilon \to 0$$

Ainsi il y a des sources pour lesquelles le taux optimal R approche H +1 d'aussi près que nous voulons.

Cet exemple d'une source binaire d'entropie faible est caractéristique : nous ne pouvons pas en général « bien coder » ce type de source à cause de la limitation imposée dès le départ dans notre présentation des codes VLC : à savoir que nous codons la source symbole par symbole (c'est du codage « scalaire »).

Il est donc nécessaire, dans de pareils cas, de passer en codage vectoriel, c'est à dire de prendre en compte, lors du codage, plusieurs symboles successifs en même temps. En tout état de cause, une telle approche prend en compte les corrélations éventuelles des symboles de source là où le codage scalaire est impuissant à le faire ; le codage vectoriel doit donc conduire à une performance améliorée.

Pour prendre en compte plusieurs symboles successifs en même temps, nous pouvons tout simplement coder la source par blocs de n symboles. C'est du codage vectoriel (en dimension n). Cela revient à remplacer la source initiale par une « super-source » dont les symboles sont des vecteurs de n symboles successifs de la source initiale. Cette « super-source » s'appelle « extension d'ordre n » de la source initiale.

VIII. Théorème de Shannon :[13]

Examinons de plus près l'intérêt de coder vectoriellement (disons en dimension n) une source X donnée. Appliquons donc un code VLC optimal (de Huffman) à l'extension d'ordre n de la source. En fait, un code de Fano- Shannon fait l'affaire pour ce qui nous occupe ici. D'après ci-dessus, nous avons l'encadrement :

$$H_n \leq R_n \leq H_{n+1}$$

où R_n et H_n sont respectivement le taux de codage et l'entropie de la source étendue à l'ordre n. Ces deux quantités s'expriment en bits codés par symbole de source ; or un symbole de source est ici un « super-symbole », c'est-à-dire un vecteur de n symboles de source initiale X.

Par conséquent, nous revenons aux mêmes unités (bits codés par symbole de source X) en divisant par n ; le taux de codage, exprimé en bits par symbole de source X, vaut

$$R = R_n / n$$

alors que l'entropie

$$\mathcal{H} = H_n / n$$

représente l'entropie d'ordre n de la source (en bits par symbole de source X).

La formule ci-dessus devient :

$$\mathcal{H} \leq R \leq \mathcal{H} + \frac{1}{n}$$

Nous voyons bien que la situation du cas scalaire (n = 1) s'est améliorée, à cause de la présence du $\frac{1}{n}$ plus nous montons en dimension, plus nous approchons de l'entropie.

Nous approchons plus de l'entropie, mais de quelle entropie ? Pour le savoir, évaluons H dans le cas simple d'une source « sans mémoire », c'est à dire une source X dont les symboles sont indépendants dans le temps (et identiquement distribués).

L'entropie d'une source sans mémoire à l'ordre n est indépendante de n, et vaut :

$$\mathcal{H} = \mathrm{H} = \sum_{i=1}^{M} pi \log 2 \ \frac{1}{pi}$$

- Le théorème de Shonon de 1948 c'est-à-dire le premier théorème de Shannon pour le codage de source sans pertes ; il existe d'autres théorèmes de Shannon pour le codage de source avec pertes, le codage de canal, et le codage de source et de canal combinés dit que « *Par codage vectoriel en dimension suffisamment élevée, le taux de codage peut être rendu aussi proche de l'entropie de la source que l'on veut* ».

L'entropie apparait donc non seulement comme une « limite » inférieure sur le taux de codage mais aussi comme une limite « théoriquement atteignable ».

Lorsque la source est « avec mémoire » nous avons $\mathcal{H}$ < H ; nous en concluons que le codage vectoriel permet de réduire le taux en prenant en compte les corrélations des symboles de source. Ceci dit, l'intérêt du codage vectoriel ne se limite pas à la prise en compte des corrélations des symboles de source, puisque nous avons vu que même pour une source sans mémoire, au augmente les performances (en s'approchant de H) lorsque n croît (grâce au terme $\frac{1}{n}$). Ainsi le codage vectoriel permet de réduire le taux même pour une source sans mémoire.

Le théorème de Shannon semble fermer définitivement la question du codage sans pertes en proposant de réaliser un codage vectoriel en grande dimension pour toute source. Du point de vue des performances, c'est optimal : nous approchons d'aussi près que nous voulons de l'entropie, et de tout façon nous ne pouvons pas descendre en dessous.

Cependant il y a un problème, pour coder une source M-aire en dimension n, il faut appliquer un code VLC à la source étendue pour laquelle nous avons M^n symboles possibles. Le code doit donc être de taille M^n, qui augmente exponentiellement avec n. Par exemple, si nous voulons garantir un taux s'approchant de l'entropie à moins d'un centième de bit pour une source binaire, il faut un code de taille

2^{100}= 1267650600228229401496703205376

Inutile de dire que c'est exagéré: il faudrait stocker la table du code au codeur et au décodeur, ou transmettre cette table du codeur au décodeur, ce qui nécessite un « *overhead* » (données annexes) qui prend une part non négligeable du débit global.

Résumé :

Le codage vectoriel en grande dimension est d'un grand intérêt théorique. Mais en pratique, pour des questions de complexité et de mémoire, nous somme amené à imaginer d'autres solutions pour prendre en compte des symboles de source successifs. Une de ces solutions est le codage par plage.

IX. Applications du codage entropique :[16]

La principale application du codage entropique est la compression de données. Si le codage de Huffman a rapidement laissé sa place aux méthodes par dictionnaire pour la compression de données génériques, il reste très utilisé en compression d'images, et est présent dans la norme JPEG. Le codage arithmétique s'est montré efficace seulement à partir du début des années 1990, et est utilisé aussi bien en compression de données génériques qu'en compression d'images (JPEG 2000) et vidéo (H.264).

X. simplification de la table de VLC :[16]

Une méthode permettant de réduire la complexité de décodage consiste à utiliser une table de mots de code de taille réduite. Nous présentons ici un algorithme de regroupement des mots de code de même longueur en classes. Ainsi, par exemple, les 16 mots de code de 8 bits utilisés pour le codage de la texture dans la norme H.263+

$$A_8 = \{00100000, 00100001 \ldots .00101110, 00101111\}$$

sont formés du préfixe 0010 et de tous les suffixes possible de 4 bits. A_8 peut ainsi être décrit de manière compacte par 0010$$$$, où $ représente soit 0, soit 1. A l'aide de cette simplification, dans le treillis présenté au paragraphe précédente, 16 branches parallèles entre les nœuds (a) et (a + 8) peuvent être remplacées par une branche unique. Le principe de l'algorithme proposé généralise la technique de réduction présentée précédemment.

XI. Exemple d'encodeur entropique (CAVLC) : [17]

XI.1. Définition :

Le Context-adaptive variable-length coding ou CAVLC est une forme de codeur entropique à longueur variable utilisé dans la norme vidéo H.264 ou MPEG-4 AVC. Il fait partie des techniques de compression sans perte, c'est-à-dire qu'à partir du code binaire généré par le codeur, il est possible de faire l'opération inverse et de retrouver les informations d'entrée sans aucune différence.

XI.2. Principe :

Le CAVLC est un codeur adaptatif à longueur variable basé sur le codage de Huffman. Il s'agit d'un codage souvent représenté en arbre dans lequel chaque symbole est assigné à un code plus ou moins long selon sa probabilité d'apparition. Un code sera long si sa probabilité est faible. Le défaut du codage de Huffman est que le nombre de bits qu'il attribue pour chaque symbole est entier et donc il est souvent loin du codage optimal. Par exemple, si nous considérons une probabilité de 0.99%, le meilleur codage selon l'entropie

$$H = -\log(P), \text{ où P est un réel compris entre 0 et 1 (la probabilité)}$$

est de 0.015 bits or le symbole assigné sera codé sur 1 bit (au minimum).

XI.3.Applications :

Dans la norme H.264, son rôle est d'encoder les coefficients des blocs résiduels de l'image à encoder après les étapes de transformée de Fourier discrète, de quantification et de ré-ordonnancement. Il s'agit de la dernière étape d'encodage qui permet de créer le flux binaire. L'opération inverse est utilisée par le décodeur.

La norme H.264 possède également un deuxième codeur entropique mais de type arithmétique CABAC (Context-based adaptive binary arithmetic coding) qui est plus efficace d'un point de compression par rapport au CAVLC (environ 10% de gain en compression) mais qui demande beaucoup plus de ressources pour le décodage.

XII. Conclusion :

Dans ce chapitre nous avons présenté quelques définitions concernant le codage à longueur variable (VLC) qui est appelé aussi le codage entropique car tous se joue autour de l'entropie d'une source donnée. Nous avons aussi donné quelques méthodes ou bien algorithme qui utilisent le principe du codage VLC ainsi que les travaux de Kraft, Shanon, Fano et Elias afin de trouver un code optimal et aussi les types de codage entropique.

Les méthodes VLC sont utilisées principalement dans le domaine de la compression de données. Dans le chapitre qui suit nous allons détailler un des algorithmes qui utilise le codage à longueur variable qui est l'algorithme de Huffman car nous avons besoin de cet algorithme pour faire notre étude comparative.

Chapitre III

L'algorithme de Huffman

I. Introduction :

Quand il s'agit de transmettre de l'information sur un canal non bruité, l'objectif prioritaire est de minimiser la taille de la représentation de l'information : c'est le problème de la compression de données (lors d'un canal bruité on ajoute alors de la redondance – code correcteur d'erreurs - pour pallier les erreurs de transmission). Le code de Huffman (1952) est un code de substitution de longueur variable optimal ; la longueur d'un texte codé est minimale. On observe ainsi des réductions de taille de l'ordre de 20 à 90%. Ce code est largement utilisé, souvent combiné avec d'autres méthodes de compression. Il s'agit d'une méthode classique de construction de code optimal (table de conversion/substitution de chaque caractère en une chaîne binaire) qui utilise habituellement des arbres binaires. C'est un codage entropique de type VLC (codage à longueur variable) qui consiste à coder les caractères qui apparaissent souvent dans un texte par un code binaire court, et ceux qui apparaissent plus rarement par un code plus long.

II. L'algorithme de Huffman :

II.1. Présentation de la méthode :[4][26]

Le codage de Huffman est un algorithme de compression des données basé sur les fréquences d'apparition des caractères apparaissant dans le document initial. Il a été développé par un étudiant de la MIT (Massachusetts Institute of Technology), David A. Huffman en 1952. Cette technique est largement utilisée car elle est très efficace et nous observons selon le type de données des taux de compression allant de 20% à 90% mais plus généralement entre 30% et 60%. Le principe de compression est utilisé dans le codage d'image TIFF (Tagged Image Format File) spécifié par Microsoft Corporation et Aldus Corporation. La méthode JPEG utilise aussi la compression de type Huffman pour coder les informations d'une image (Elle utilise d'ailleurs des tables prédéfinies).

Ce procédé fait partie des méthodes de compression de type dites statistiques. Cela repose sur le principe d'attribution de codes plus courts pour des valeurs fréquentes et de codes plus longs pour les valeurs moins fréquentes. Cela est plus efficace que la représentation actuelle qui consiste, quant à elle, à utiliser une longueur fixe pour chaque symbole (exemple : un octet par caractère, code ASCII).

Le codage de Huffman ressemble au code de Shannon – Fano dans le sens où il affecte à chaque caractère de la source u mot dont la longueur s'approche autant que possible du minimum théorique.

II.2. Principe de l'algorithme de Huffman :[4][26]

L'algorithme de Huffman utilise une table contenant les fréquences d'apparition de chaque caractère pour établir une manière optimale de les représenter par une chaîne binaire (cela reprend le principe du morse qui tend à minimiser le nombre de symboles à utiliser pour les lettres les plus fréquemment employées).

Nous pouvons décomposer la procédure en plusieurs parties :

- ✓ Tout d'abord, la création de la table de fréquence d'apparition des caractères dans le texte initial.
- ✓ Ensuite la création d'un arbre binaire (usuellement dénommé arbre de Huffman) suivant la table précédemment calculée (nous devrions parler plutôt de l'arborescence de Huffman).
- ✓ Enfin coder les symboles en représentation binaire optimale.

Avant de détailler les trois étapes essentielles de l'algorithme de Huffman nous allons définir d'abord quelques notions sur les arbres.

III. Les arbres :[18]

III.1. Définitions :

- ❖ **Les arbres** sont des structures non linéaires : chaque élément de l'arbre, bien que n'ayant qu'un prédécesseur, peut avoir plus d'un successeur. Dans la au dessous C1, C2 et C3 qui constituent les chapitres du livre sont tous successeurs du nœud livre.

- ❖ **Une feuille** est un nœud qui n'a pas de successeur (C3).

- ❖ **La racine** est un nœud qui n'a pas de prédécesseur (Livre).

- ❖ Les nœuds sont liés par une relation dite de parenté (symbolisée par un segment ou un arc). Livre est le père (parent) de C1, C2 et C3. Ces trois nœuds sont les fils de Livre. C1, C2, C3 sont des nœuds frères. C1, C2 et C3 sont des sous-arbres de Livre. C3 est un sous arbre sans descendance. Le sous-arbre de nœud C2 a trois sous-arbres correspondant aux sections S2.1, S2.2, et S2.3.

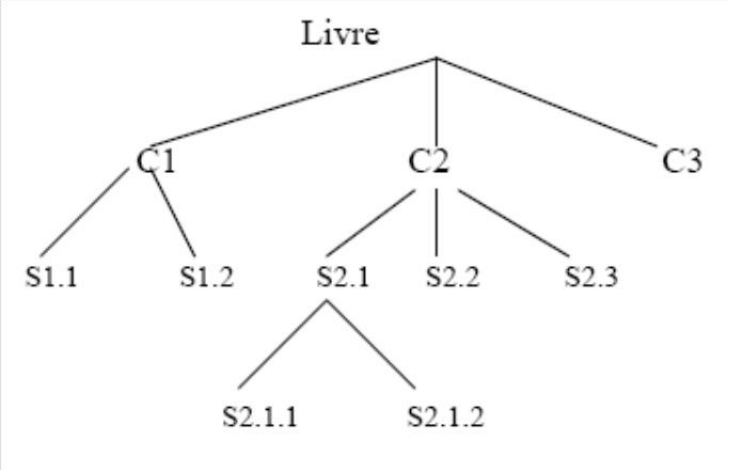


Figure3.1 : Exemple d'une structure d'un arbre.

III.2. Les types d'arbres :

Il existe plusieurs types d'arbres, les principaux sont les suivants :

- **Arbre binaire :** C'est un arbre dont un nœud n'accepte pas plus de deux nœuds, un nœud fils à droite et un fils à gauche (à part les feuilles). Dans ce type d'arbre nous avons:
  - Arbre binaire (non équilibré)
  - L'arbre AVL (Arbre binaire équilibré)

- **Arbre n-aires :** C'est un arbre dont chaque nœud peut accepter de 0 à n nœuds. Dans ce type d'arbre nous avons:
  - L'arbre de recherche (arbre non équilibré)
  - Le B-arbre
  - Le B-arbre*
  - Le B-Arbre+

III.3. Les opérations de base sur l'arbre:

La structure d'arbre n'est pas liée à des primitives de manipulations caractéristiques, comme Empiler et Dépiler. Nous distinguons les arbres quelconques sur lesquels les seules opérations usuelles de manipulation sont les parcours des arbres ordonnés sur lesquels il peut être intéressant de définir en plus, des primitives d'ajout et de retrait d'un élément tout en conservant l'ordre. Il est évidemment possible de faire des opérations de mise à jour (retrait ou ajout d'un élément) sur des arbres quelconques. Dans ce cas, il faut en général prendre en compte le contexte particulier de l'utilisation de l'arbre. Par exemple dans une arborescence de fichiers sur disque, nous ne pouvons détruire un fichier que s'il est une feuille de l'arbre.

III.4. Parcourt d'arbres :

Parcourir un arbre signifie visiter dans un certain ordre tous ses nœuds. Les parcours les plus classiques sont les parcours infixé, préfixé et postfixé. Ces parcours sont définis récursivement de la manière suivante :

1. ***Infixé (symétrique)*** *:* nous parcourons d'abord le premier sous-arbre en infixé, puis la racine, puis chacun des autres sous-arbres, en infixe.

2. ***Préfixe :*** nous parcourons d'abord la racine, puis chacun des sous-arbres en préfixe.

3. ***Postfixé*** *:* nous parcourons d'abord tous les sous-arbres en postfixe, puis nous parcourons la racine.

Exemple :

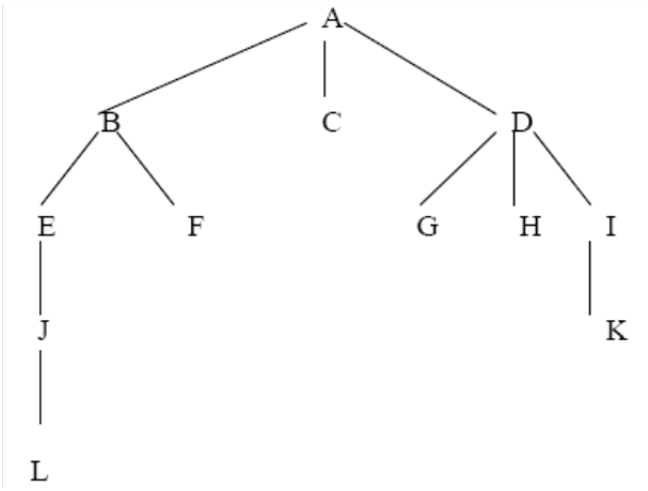


Figure3.2 : Exemple de parcourt d'un arbre.

Voici, l'ordre de visite des nœuds pour chacun des parcours précédents :

1. **infixe :** L J E B F A C G D H K I
2. **préfixe :** A B E J L F C D G H I K
3. **postfixe :** L J E F B C G H K I D A

Algorithmes de parcourt :

1) Prefixe *(nœud)*

Début

Traiter nœud
pour chaque fils f de nœud, s'il y en a
depuis le plus à gauche au plus à droite faire
Prefixe *(f)*

Fin

2) Infixe *(nœud) {*

si nœud est une feuille alors
traiter nœud
sinon {
infixe (fils le plus à gauche de nœud)
traiter nœud
pour chaque fils f de nœud, à l'exception du plus à gauche
depuis le plus à gauche jusqu'au plus à droite faire

infixe (f)
}
}

***3) Posfixe** (nœud) {*
Debut
Pour chaque fils f de nœud, s'il y en a
depuis le plus à gauche au plus à droite faire

***postfixe** (f) ;*
Traiter nœud ;
Fin

}

III.5. Les arbres binaires :

Les arbres binaires sont un cas particulier d'arbres. Ce sont des arbres où chaque nœud possède au plus un fils gauche et au plus un fils droit. Même si un nœud possède donc un seul fils, il peut être un fils gauche ou un fils droit. Un arbre binaire est beaucoup plus simple qu'un arbre quelconque.

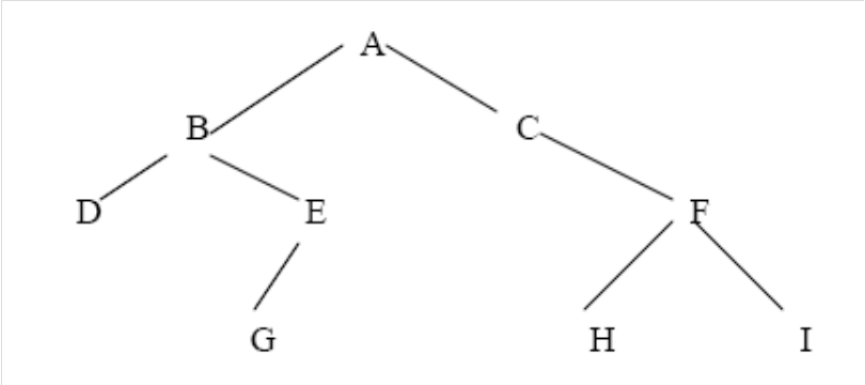


Figure 3.3 : Exemple d'arbre binaire.

IV. La procédure de compression de Huffman :[4]

IV.1. 1ère étape : Table de fréquence d'apparition :

Afin de construire cette table, il suffit simplement de dénombrer le nombre d'occurrences de chaque symbole « s » puis de calculer la fréquence « f_s » de chacun d'entre eux grâce à la formule suivante :

$$fs = \frac{nombre\ d'occurences de\ s}{nombre\ de\ symboles}$$

Ensuite nous trions le tableau en fonction de la fréquence d'apparition de façon croissante.

Exemple :

Supposons par exemple que nous ayons le texte suivant composé des symboles pris dans le code ASCII :

"L'algorithme de Huffman est une méthode qui permet de compresser les données".

Pour chaque symbole nous calculons tout d'abord le nombre d'occurrences de chacun puis sa fréquence d'apparition suivant la formule citée précédemment.

La table de fréquences T triée est (Afin de simplifier l'exemple, on a négligé la casse) :

Symbole	Nombre d'occurrence	Fréquence d'apparition
c	1	1/76=1.32%
g	1	1/76=1.32%
q	1	1/76=1.32%
‘	1	1/76=1.32%
a	2	2/76 = 2.63%
f	2	2/76 = 2.63%
i	2	2/76 = 2.63%
p	2	2/76 = 2.63%
u	3	3/76 =3.95%
h	3	3/76 =3.95%
l	3	3/76 =3.95%
d	4	4/76 =5.26%
n	4	4/76 =5.26%
o	4	4/76 =5.26%
r	4	4/76 =5.26%
t	4	4/76 =5.26%
m	5	5/76 =6.58%
s	5	5/76 =6.58%
	11	11/76 =14.47%
e	14	14/76 =18.42%

Tableau3.1 : Exemple de table de fréquences de Huffman.

IV.2. 2ème étape : Construction de l'arbre de Huffman :

L'arbre binaire de Huffman est la structure de données qui va nous permettre d'attribuer à chaque symbole une représentation binaire optimale. Afin de construire l'arbre, on utilise la table de fréquences précédemment construite que nous appelons T.

Étapes de la construction de l'arbre :

1ère étape : nous écrivons les symboles du code primaire dans l'ordre croissant.

2ème étape : nous regroupons les deux premiers éléments du code (i.e. les deux revenants le moins souvent) en additionnant leur fréquence.

3ème étape : avec ces deux éléments comme des branches, nous créons un nœud en associant la valeur 1 aux branches droites et la valeur 0 aux branches gauches.

4ème étape : nous donnons au nœud créé la somme des fréquences de ces deux fils.

5ème étape : nous sélectionnons, parmi tous les éléments et nœuds les deux dont la fréquence est la plus faible.

6ème étape : nous répètons le schéma de création d'un nouveau nœud et on lui attribue sa fréquence.

7ème étape : nous continuons jusqu'à obtenir la racine de l'arbre.

8ème et dernière étape : en commençant par la racine, nous remontons jusqu'aux éléments ; la suite des poids des branches leur donne le code secondaire.

Voici un algorithme qui sert à construire l'arbre de Huffman :

Algorithme : Construction de l'arbre

Données :

- T : la table de fréquence
- Q : Une file d'attente de nœuds de l'arbre binaire. Chaque feuille est étiquetée avec un symbole et son nombre d'occurrences. Chaque nœud interne est étiqueté avec la somme des occurrences des feuilles de sa sous arborescence.
- o : Une fonction qui à chaque nœud de l'arbre associe une valeur. Si le nœud est une feuille alors o renvoie le nombre d'occurrences du symbole, autrement o renvoie la somme des occurrences des feuilles de la sous-arborescence du nœud.

Résultat :

- A : L'arbre binaire résultant

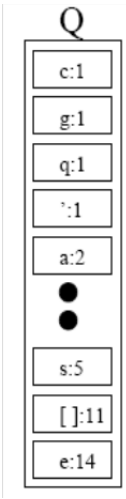
```
begin
    Initialisation de Q tel que :
    Q contienne les feuilles représentant les symboles de la table T

    tant que (Q non vide) faire
        Créer un nouveau nœud z dans A tel que :
        gauche(z) = x = extraire-min (Q)
        droite(z) = y = extraire-min (Q)
        o(z) = o(x) + o(y)
        Insérer (z, Q)
    fin
end
```

De façon informelle, nous utilisons une file d'attente Q dans laquelle nous plaçons les nœuds correspondants au couple [symbole : nombre d'occurrences du symbole] de tous les symboles. Ensuite nous extrayons de la file d'attente les 2 nœuds ayant la valeur minimale puis nous créons un nouveau nœud dans l'arbre de Huffman ayant pour fils les deux sélectionnés, nous rajoutons ensuite le nœud nouvellement crée dans la file d'attente, et nous réitérons jusqu'à ce que la file soit vide.

Voici l'arbre de Huffman de l'exemple précédent expliqué par des schéma en suivant les différentes étapes de sa construction :

Initialisation :



Etape 1 :

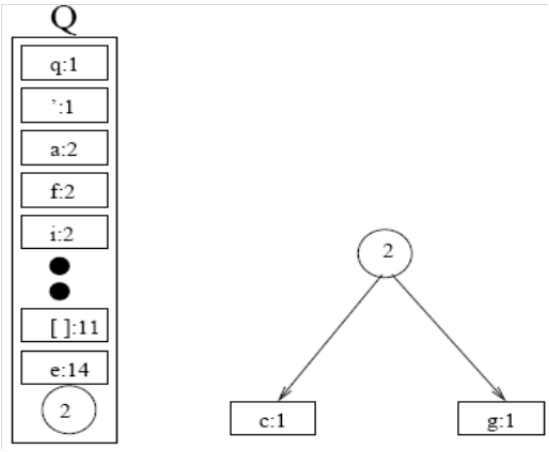


Figure3.4 : Etape-1- du codage de Huffman.

Etape 2 :

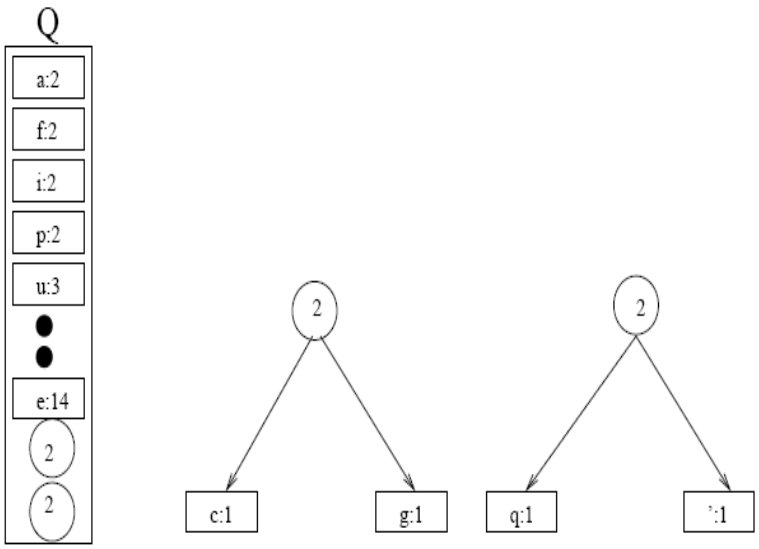


Figure3.5 : Etape-2- du codage de Huffman.

Etape finale :

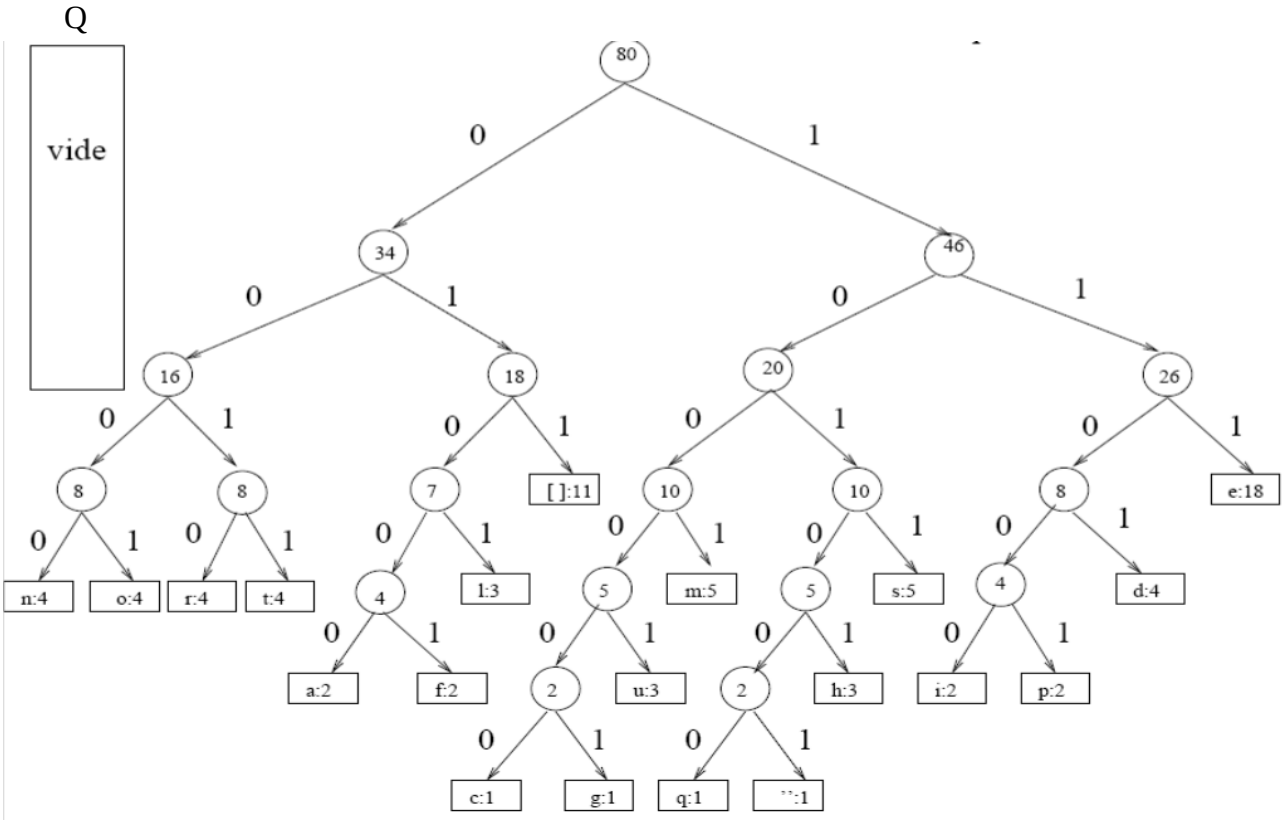


Figure3.6 : Etape finale du codage de Huffman.

L'arborescence de Huffman est finalement construite, nous étiquetons les arcs de la façon suivante :

- Les arcs reliant un nœud à son fils gauche sont étiquetés par '0'.
- Les arcs reliant un nœud à son fils droit sont étiquetés par '1'.

De cette manière, chaque feuille représentant un symbole peut être redéfinie par un nombre binaire correspondant au chemin entre la racine et la feuille de l'arborescence.

Ainsi, les symboles les plus utilisés ont une représentation binaire moins importante (en terme de taille) que les symboles les moins utilisés. Ceci permet de représenter chaque symbole de façon optimale et permet de réaliser une compression des données efficacement.

A partir de cet exemple, nous obtenons la table de correspondance suivante :

Symbole	Représentation binaire	Taille (en bits)	Gain (en bits)
C	100000	6	2
G	100001	6	2
Q	101000	6	2
‘	101001	6	2
a	01000	5	3
f	01001	5	3
i	11000	5	3
p	11001	5	3
u	10001	5	3
h	10101	5	3
l	0101	4	4
d	1101	4	4
n	0000	4	4
o	0001	4	4
r	0010	4	4
t	0011	4	4
m	1001	4	4
s	1011	4	4
	011	3	5
e	111	3	5

Tableau3.2 : Exemple de table de correspondance de Huffman.

IV.3. 3ème étape : Codification du message :

La phrase de l'exemple devient :

"L'algorithme de Huffman est une méthode qui permet de compresser les données"

l = 0101
’ = 101001
a = 01000
g = 100001
...
Message compressé = 010110100101000010110000 1 etc...

Le message compressé est une suite binaire de bits alors que le message original est une suite de symboles qui peut être ramené à une suite de bits.

IV.4. Compression / Décompression :

❖ *Compression :[19]*

Pour la compression des données, nous devons écrire un nouveau fichier contenant deux parties : l'entête et les données compressées. L'entête : il doit contenir le nom original du fichier, la taille originale et une table de correspondance qui permette de reconstituer le fichier original.

Pour compresser les données, nous allons donc lire le fichier original fragment par fragment (par exemple octet par octet) et on écrira le code Huffman correspondant dans le fichier de destination. Les codes de Huffman ne font pas toujours un multiple de 8 bits et par conséquent ils nécessitent d'être mis dans une zone tampon avant d'être écris dans le fichier. Nous pouvons par exemple utiliser un tableau de 8 caractères qui contienne soit '1' soit '0'.

❖ *Décompression :[18]*

Le décodage d'un message, codé avec l'algorithme de Huffman, est effectué en regardant à la chaîne de bits de gauche vers la droite jusqu'à`ce qu'une lettre soit décodé. Cela pourrait être fait en utilisant l'arbre de Huffman en commençant par la racine. Dépendant de la valeur du bit : gauche pour 0 ou droite pour 1, nous suivons les branches de l'arbre jusqu'à arriver à une feuille. La feuille contient le premier caractère du message. Nous recommençons ensuite ce processus pour les autres caractères du message.

V. Les variantes de l'algorithme de Huffman :[20][25]

Il existe quatre variantes de l'algorithme de Huffman, chacune d'elle définissant une méthode pour la création de l'arbre :

1. ***Statique :*** Dans ce cas il est nécessaire de disposer d'une table de Huffman prédéfinie. Cette table sera choisie avant la compression en fonction du type de données à compresser, par exemple « Table de Huffman pour textes français », « pour textes anglais », « pour codes sources HTML ». Ces tables ont été obtenues après analyse de quelques textes concernés afin d'obtenir les fréquences moyennes couramment utilisées.

   Ce type de codage assure une compression de compromis à partir d'une table de codage qui est établie une fois pour toutes à partir d'une analyse statistique portant sur un très grand nombre de messages. L'ensemble des tables définies sont repérées par un numéro et sont prédéfinies au niveau du codeur et du décodeur et le choix d'une table est effectué en fonction du type du message.

L'utilisation d'un jeu de tables de codage prédéfinies présente le grand avantage d'assurer une meilleure adaptation du codage aux différents messages, tout en ne nécessitant que très peu d'informations supplémentaires pour préciser la table utilisée, par exemple 4 bits pour un jeu de 16 tables.

2. ***Dynamique (semi-adaptatif) :*** Cette méthode est strictement identique à la méthode statique, mais crée dynamiquement l'arbre avant la compression. Pour cela, elle effectue 2 phases, dans la première il fait une analyse fréquentielle du texte, puis crée l'arbre et la table de codage et dans la deuxième il fait le codage.

   Dans cette méthode comme le décodeur ne dispose pas de la table de codage (ou l'arbre généré), cette dernière doit être jointe au message comprimé, ce qui nécessite la transmission d'informations supplémentaires. Pour que la méthode soit intéressante, il faut que la table de codage transmise avec le message ait une taille négligeable par rapport à ce dernier.
   Nous avons la relation suivante :

   $M = N - 1$ avec *N : le nombre total de feuilles de l'arbre.*
   M : le nombre de nœuds intermédiaires.

   Si nous prenons l'arbre construit lors de l'explication de la méthode de Huffman, nous avons $M = 19$ et $N = 20$.

   Nous pouvons constater que le nombre total de nœuds dans un arbre de Huffman est égal à : $N + (N - 1)$. Donc à chaque fois que le nombre de caractères augmente la taille de l'arbre augmente aussi.

   Dans le cas de petits fichiers le taux de compression peut être très mauvais, voire négatif. Il est donc préférable d'appliquer cette méthode sur des textes volumineux. Il serait possible de faciliter la transmission du texte codé en combinant les deux fichiers en un seul, par exemple un en-tête contenant la table et le corps contenant le texte encodé.

   Le fait que la table de codage doit accompagner le fichier peut être vu comme un avantage plutôt qu'un inconvénient, car il garanti que le fichier pourra toujours être décodé par la bonne table, même s'il a été archivé très longtemps auparavant. Mais il y a également un autre inconvénient c'est le temps d'encodage car il est nécessaire d'analyser le texte entièrement avant de pouvoir démarrer l'encodage, cet algorithme est donc inutilisable en « Temps réel » (envoi de fax, streaming…), il faut alors se tourner vers la méthode dynamique (ou adaptative).

3. ***Adaptatif :*** Le codage purement dynamique est indispensable dans le cas de traitements en temps-réel tel que l'envoi de Fax ou le streaming de données car nous ne pouvons pas attendre d'avoir analysé le texte entier avant d'envoyer les premiers résultats. Ce type de codage vise à effectuer la compression avec une table de codage qui est adaptée à l'état courant des fréquences d'apparition des caractères dans le message source.

Le principe de l'algorithme est de lire chaque caractère l'un après l'autre, et de construire dynamiquement l'arbre à chaque itération, c'est-à-dire à chaque fois que la source produit un nouveau symbole du message, le codeur utilise l'arbre de Huffman dont il dispose à l'étape précédente pour coder ce caractère. Le codeur doit ensuite mettre à jour la table des nombres d'occurrences des caractères en incrémentant d'une unité le compte du caractère. Un même caractère aura donc souvent plusieurs codes différents entre le début et la fin du texte.

Pour que cette méthode soit réalisable, il faut évidemment que le décodeur interprète chaque mot de code reçu avec le même arbre que celui qui a été utilisé par le codeur. Deux conditions doivent toujours être respectées :

- La fréquence de chaque nœud doit être égale à la somme des fréquences de ces deux fils.
- Il doit exister un ordre de sur les nœuds de l'arbre : les poids doivent être croissants, chaque nœud droit est > au nœud gauche.

Lorsque la fréquence d'un nœud doit être augmentée, l'arbre est mis à jour de façon à que la condition (2) soit respectée, sa fréquence est incrémentée, puis cette procédure est répétée récursivement jusqu'à atteindre la racine.

Cette méthode représente cependant le gros désavantage de devoir reconstruire l'arbre à chaque fois, ce qui implique un temps d'exécution énorme.

4. ***Huffman local :*** Dans beaucoup de cas, le modèle basé sur la fréquence relative des caractères ne représente la redondance que d'une façon partielle, car il peut exister des corrélations importantes entre caractères voisins dont il est possible de tirer parti pour comprimer plus efficacement le message source. Ces corrélations peuvent être prises en compte par le codage de Huffman en codant des groupes de caractères au lieu de caractères isolés.

Par exemple si l'alphabet de la source contient 128 caractères, l'arbre de Huffman correspondant comprend 255 nœuds (trouvé avec la relation précédente M = N -1, donc le nombre de nœuds est 128 + (128-1) =255). Maintenant, si nous tenons compte des corrélations entre deux caractères adjacents nous aurons 16384 paires de caractères donc 32767 nœuds dans l'arbre. Les opérations de codage et de décodage dans ce cas deviennent de plus en plus complexes.

Il est possible d'arriver à une solution pour prendre en compte les corrélations en utilisant des arbres différents selon le contexte local du message. Cette approche peut convenir par exemple au cas de message dont certaines parties ne contiennent que du texte, avec des lettres et des signes de ponctuation, alors que d'autres contiennent des données numériques, avec des chiffres et quelques signes habituellement associés aux chiffres tels que + ou =. Dans une telle situation, il est logique d'effectuer le codage localement, avec des arbres de Huffman différents pour les chiffres et pour les lettres, et avec un mécanisme qui permet de passer d'un arbre à l'autre lorsque le contexte local change.

Voici un exemple :

Caractère source	Probabilité p de production	Mot de code (Huffman global)	Mot de code (Huffman local)
A	0.2	10	1
B	0.1	011	01
C	0.05	01010	001
0	0.2	11	1
1	0.1	0100	01
2	0.05	01011	001
(espace)	0.3	00	0

Tableau3.3 : Comparaison entre le codage local et global de Huffman.

VI. Efficacité des différentes variantes du codage de Huffman :[25]

- L'encodage statique possède les avantages d'être rapide et de ne pas devoir transmettre d'arbre de décodage. En contrepartie, il faut choisir une table qui soit adaptée au fichier source (texte français, anglais, code source…), dans les autres cas les performances seraient dégradées (gain très faible voir négatif).

- L'encodage dynamique calcule dynamiquement les fréquences du texte, mais l'arbre de décodage doit être transmis ce qui diminue le gain, surtout dans le cas de fichiers peu volumineux, la taille finale (données compressées + dictionnaire) pouvant être supérieure au fichier initial. De plus, cet algorithme nécessite d'effectuer deux passes, il n'est donc pas utilisable sur les flux (en temps réel).

- L'encodage adaptatif ne nécessite aucune transmission de données supplémentaires et peut travailler sur des flux, mais est peu performant en terme de rapidité car l'arbre est mis à jour très souvent pendant les phases de codage et de décodage.

- En théorie, l'algorithme de Huffman est optimal dans le sens qu'il utilise moins d'espace pour effectuer un codage. Mais cela n'est vrai que si l'on connaît les vraies fréquences d'apparition des lettres, et cette fréquence est indépendante du contexte de la lettre dans le message. En pratique, la fréquence d'apparition d'une lettre change en fonction du contexte du message.

- Le deuxième facteur qui peut affecter l'efficacité de la compression du codage de Huffman est la fréquence relative des lettres. Certaines fréquences ne donnent aucun gain par rapport au codage de longueur fixe, alors que d'autres vont générer un gain substantiel. Toutefois, on peut dire, qu'en général, le codage de Huffman performe mieux quand il y a une grande variation dans les fréquences d'apparition des lettres.

- Le codage que construit l'algorithme de Huffman est un codage préfixe : aucun code n'est un préfixe d'un autre.

- La taille de l'arbre binaire de Huffman ainsi que celle de l'en- tête pour transmettre des données sont significativement plus importantes. Il faut aussi considérer le temps de traitement qui est plus long.

VII. Limitation du codage de Huffman :[21]

Nous pouvons montrer que pour une source X, d'entropie H(X) la longueur moyenne L d'un mot de code obtenu par codage de Huffman vérifie:

$$H(X) \leq L < H(X) + 1$$

Cette relation, qui montre que le codage de Huffman s'approche effectivement de l'entropie de la source et donc de l'optimum, peut s'avérer en fait assez peu intéressante dans le cas où l'entropie de la source est faible, et où un surcoût de 1 bit devient important. De plus le codage de Huffman impose d'utiliser un nombre entier de bit pour un symbole source, ce qui peut s'avérer peu efficace. Une solution à ce problème est de travailler sur des blocs de n symboles. Nous montrons alors que nous pouvons s'approcher de façon plus fine de l'entropie:

$$H(X) \leq L < H(X) + \frac{1}{n}$$

Mais le processus d'estimation des probabilités devient plus complexe et coûteux. De plus, le codage de Huffman n'est pas adapté dans le cas d'une source dont les propriétés statistiques évoluent au cours du temps, puisque les probabilités des symboles sont alors erronées. La solution consistant à ré-estimer à chaque itération les probabilités des symboles est impraticable du fait de sa complexité.

L'algorithme de Huffman est encore très utilisé de nos jours. La norme non conservative JPEG, après avoir éliminé un certain type d'informations, compresse les données restantes à l'aide de cet algorithme. Il est également utilisé pour la transmission de fax, mais également par la plupart des compresseurs comme second étage de compression par exemple après avoir été comprimé par une autre technique, les données sont surcomprimées à l'aide de l'algorithme de Huffman.

VIII. Performances de l'algorithme de Huffman :[4]

L'intérêt de la compression est de pouvoir réduire au maximum la taille des informations originales. Cette mesure peut-être effectuée par le taux de compression qui est défini par la formule suivante :

$$\rho = 1 - \frac{taille\_compressé}{taill\_normale}$$

Par ailleurs, l'utilisation de l'entropie permet de connaître le nombre minimum de bits nécessaires au codage d'un fichier. Nous rappelons que la formule de l'entropie est la suivante :

$$E = -\sum_{N}^{1}(Pk * log2(Pk))$$

Où :

P_k est la fréquence d'apparition du k-ième symbole parmi les n possibles.

Les performances sont dépendantes de la fréquence d'apparition des symboles dans le message original, par conséquent elles dépendent également du type de fichier que nous désirons compresser. En effet, les fichiers de type exécutable par exemple contiennent, de façon générale, moins de redondances qu'un fichier de type BMP (bitmap pour les images). De cette remarque, nous en déduisons que le taux de compression sera en moyenne meilleur pour les fichiers où les redondances sont fortes.

Des études ont déjà été réalisées afin de déterminer les taux de compression, voici un aperçu de ce que ces dernières nous montrent :

Type de fichier	Taux de compression
Texte	49.5%
Bitmap (image)	50%
Wave (audio)	50%
Exécutable	20%

Tableau3.4 : Taux de compression des différents types de fichier avec Huffman.

IX. Conclusion :

Dans ce chapitre nous avons présenté un des algorithmes les plus connus qui est l'algorithme de Huffman. Nous avons donné son principe, ses différentes étapes ainsi que quelques caractéristiques, ses variations, ses avantages et ses inconvénients.

L'algorithme de Huffman est un procédé largement répandu et qui se révèle être un algorithme performant en moyenne. Il est le plus représentatif des algorithmes de compression dit de type statistique : il en existe néanmoins d'autres tels que le VLC (Variable Length Code) dénommé souvent codage entropique.

Excepté tous ces avantages, Huffman présente l'inconvénient d'être relativement ancien et d'autres algorithmes plus récent reposant sur d'autres principes tels que les algorithmes de compression à dictionnaire (exemple : LZW) arrivent a des taux de compression supérieurs en moyenne à ceux que proposent l'encodage de Huffman.

Chapitre IV

Analyse et conception

I. Introduction :

Dans ce chapitre nous allons présenter dans une première partie la nouvelle méthode de compression à longueur variable proposée et ses trois variantes. Nous détaillerons par la suite les deux variantes utilisées dans notre application. Nous présenterons l'architecture de notre application en expliquant les deux processus de codage et de décodage.

Dans une deuxième partie, nous présenterons les différents modules de l'application qui fait la compression avec l'algorithme de Huffman en expliquant aussi les deux processus de codage et du décodage.

En plus des deux principaux processus de codage et de décodage nos deux applications doivent calculer le taux et le temps de compression et de décompression de ces algorithmes.

II. Présentation générale de la méthode proposée :

Dans cette partie nous allons faire un aperçu global sur la méthode de compression à longueur variable proposée dans ce projet, en plus de ses différents composants (modules) et le rôle de chacun d'eux.

Cette méthode peut être incluse dans les méthodes de compression sans perte, car lors de la décompression elle reproduira exactement le même fichier d'origine sans rien supprimer. Nous pouvons déduire que cette méthode sera plus efficace dans le cas des fichiers ou chaque information est utile comme les fichiers textes contrairement aux fichiers ou la suppression de quelques informations n'influence pas beaucoup comme les fichiers images ou bien vidéos.

II.1. Principe de la méthode proposée :

II.1.1. Le codage :

Le codage fixe est facile à générer. Pour un code fixe sur N bits nous aurons 2^N Symboles S= {S_0, S_1, … S_{2n-1}}. Le codage VLC correspondant à cet ensemble de symboles est :

$$(S_i) = preffix_i \,.\, suffix_i \,/\, i \in [0, 2^n - 1]$$

(Si) : Codage du symbole Si.

- **La signification des deux champs *preffix et suffix* :**

  Le code VLC proposé est constitué d'une partie fixe *(preffix)* et une partie variable *(suffix)*.

- ✓ $\boldsymbol{preffix}$ **:** ce champ représente deux (02) informations :

1) Il indique la position (P) du bit le plus significatif dans le code fixe de Si. Sa valeur va de 0 à N-1 c'est-à-dire :

$$Val\ (preffix)\ = P \in [0, N-1]$$

Cas particulier : Pour $Si=0 \rightarrow Preffix=0$

Exemple 1 *:* $S4=(4)10=(\ 00\ldots\ldots100\ )_2$
$\leftarrow$ N bits $\rightarrow$

La valeur du champ $preffix$ est la position du bit le plus significatif de la droite à gauche. $Val\ preffix = P=2$

2) Il indique la longueur du champ $suffix$

La longueur de la partie préfix*e :* C'est la taille en nombre de bits du champ $preffix$. Le nombre de bits, n, nécessaire pour représenter un nombre X est :

$$n = 1+\log_2 X \approx \log_2(X+1)\ldots..(1)$$

Exemple : le nombre de bits pour représenter la valeur X=8 est :

$$\text{Log2 } (8)\ + 1\ = \frac{\log(8)}{\log(2)} + 1 \approx 1 + 3 = 4 \text{ bits}$$

Pour calculer la longueur du champ $preffix$, on doit trouver la taille de la plus grande valeur du $preffix$ appartenant à l'intervalle [0, $N-1$] qui est $X=N-1$.

Appliquant la règle (1) :

$$L(preffix) = L\ (N-1) = log2\ \big((N-1)+1\big) = log2(N) bits$$

($preffix$) *:* Longueur du $preffix$ en nombre de bits.

- ✓ $\boldsymbol{suffix}$ **:** c'est la partie restante du code fixe Si qui suit le bit significatif.

1. Val (suffix i) = Si $-2^{preffix_i} \rightarrow S_i = 2^{preffix_i} + suffix_i$
2. $(suffix_i) = 1 \quad si\ preffix \quad i=0$
   $preffix \quad sinon$

Exemple 2 *:*

N=8 bits, le nombre de symboles est $2^N = 2^8 = 256$.

En reprend le même code S4 de l'exemple 1 et nous retrouvons la valeur du $suffix$.

*S_4 = (00……1**00**$)_2$, le suffix = **'00'** de là nous avons :*
← *N bits* →

- $val\ (suffix)$ =00
- $L\ (suffix)$ =$preffix$=2 $bits$

De l'exemple 1 et l'exemple 2, nous pouvons conclure le code VLC correspondant au symbole
S4 = $(00…100)_2$ → C(S4) = $preffix_4 * suffix_4$ = 010 **00** =01000
$preffix\ suffix$

***Exemple 3** :*
N=4 bits, Si=1 jusqu'à 15 l'ensemble des symboles.

Compression de la chaîne: CH=$S_1S_1S_1S_2S_3\ S_3$.

La représentation binaire de CH est 0001 0001 0001 0010 0011 0011, la taille de CH est: L(CH)=4*6=24 bits. La compression consiste à remplacer chaque code de taille fixe 4 bits par son code VLC correspondant suivant les étapes :

✓ Calculer les probabilités des symboles et les triées selon l'ordre décroissant :

Indice	Symboles (Si)	Probabilité P(Si)
0	S1	0.5
1	S3	0.33
2	S2	0.16

✓ Construire le code VLC :
Au lieu de coder les symboles eux-mêmes, nous procédons au codage des indices des symboles.

$C(Si)$=$Preffix\ i.Suffix\ i$
N=4 $bits$→$L\ (preffix\ i)$ = log2 (4) = 2 $bits$

Indice	Indice binaire	Position du bit poids fort	Code C (Si)	
			Preffix	Suffix
0	0000	0	00	0
1	0001	0	00	1
2	0010	1	01	0

✓ Remplacer chaque code fixe par son code préfixé.

(CH) = CHc = 000 000 000 010 001 001.

CHc : chaine compressée.
L(CHc) = 3x6 =18 bits.
La moyenne de bits par symbole :

$$\sum_{i=1}^{N} \mathrm{P(Si)\ L(Ci)} = 0.5 \times 3 + 0.33 \times 3 + 0.16 \times 3 = 2.99$$

$$\sum_{i=1}^{N} P(Si)L(Ci) \approx 3\ bits\ /symbole$$

II.1.2. Le décodage :

Pour le processus de décodage, nous devons connaitre la taille du code initial (N) et la table des probabilités des différents symboles puis procéder à la décompression suivant les étapes :

- ✓ Lire $\mathrm{Log_2}$ (N) premier bits (pr$effix$)
- ✓ Si $preffix$=0, $lire\ 1\ bit$
  Sinon lire $preffix$ bits (*suffix*)
- ✓ Calculer $indice$=$2^{preffix}$+$Suffix$.
- ✓ S_i= Table [indice] et le représenter sur N bits

II.1.3. Longueur du code :

La longueur de chaque code VLC assigné aux codes fixe sur N bits est calculée suivant la relation :

$$L(C_i) = L\ (preffix_i\ .\ suffix_i)$$
$$= L\ (preffix_i) + L(suffix_i)$$
$$= \log_2\ (N) + preffix_i$$

Nous avons : $L(Suffix_i) = \begin{cases} 1\ si & preffix_i = 0 \\ preffix_i & sinon \end{cases}$

Donc :

$$L\ (Ci) = \begin{cases} log2(N) + 1, i = 0 \\ log2\ (N) + preffix\ i,\ \forall\ i \in [1, 2^N - 1] \end{cases}$$

$$L(Ci) = Cte + preffix_i\ ,\quad avec\quad Cte = log_2(N)$$

Et sachant que : $preffix_i \in [\ 0,\ N-1] \rightarrow$ Le nombre de taille du codes est N différentes tailles, alors l'équation précédente devient :

$$L(Ci) = \begin{cases} log_2(N) + 1, & i = 0 \\ log_2(N) + preffix_i, & preffix_i \in\ [\ 1, \mathrm{N} - 1] \end{cases}$$

Remarque : pour un ensemble de symboles sur N bits, nous avons les résultats suivants :

- 2^N différents codes VLC.
- $L(C)i = \log_2(N) + preffix_i$, $preffix \in [1, N-1]$, il ya N tailles différentes.
- Le nombre de code pour chaque longueur est $2^{preffix_i}$

Exemple 4 :

Supposons n=8 bits, il y a 2^8 =256 symboles. Construisons le code VLC correspondant.

$C(Si) = preffix_i . suffix_i$,$\forall\ i \in [0, 2^8-1]$

Calculer le nombre de bits de la partie préfixe :

$L(preffix\ i) = \log_2(n) = \log_2(8) = \log(8) / \log(2) = 3\ bits.$

Le tableau suivant illustre les différents codes pour la valeur N=8 :

Indice	**symbole Si**	**bit poids fort**	**Code Ci**		**L(Ci) $Log_2(8)$+ p**	**Nombre de code** $[2L(suffix)]$
			preffix	**Suffix**		
0 -1	0000000X	0	000	*X*	4	2
2-3	0000001X	1	001	*X*	4	2^1 =2
4-7	000001XX	2	010	*XX*	5	2^2 =4
8-15	00001XXX	3	011	*XXX*	6	2^3 =8
16-31	0001XXXX	4	100	*XXXX*	7	2^4 =16
32-63	001XXXXX	5	101	*XXXXX*	8	2^5 =32
64-127	01XXXXXX	6	110	*XXXXXX*	9	2^6 = 64
128-255	1XXXXXXX	7	111	*XXXXXXX*	10	2^7 =128

Tableau4.1 : Code VLC pour N=8.

II.2. Les avantages de la méthode proposée :

- ✓ ***La simplicité :*** la génération du code est très simple et ne nécessite pas de modèles mathématiques compliqués, est aussi instantanément déduit du code fixe correspondant.

- ✓ ***La flexibilité*** : le code est paramétrable. Plusieurs variétés du code peuvent être obtenues selon les paramètres : N, *preffix, suffix.*

II.3. Les variantes de la méthode :

Nous avons trois variantes de compression de cette méthode qui sont : la compression standard, la compression adaptative et la compression récursive. Dans notre application, nous avons implémenté les deux premières c'est-à-dire standard et adaptative.

II.3.1. La compression standard :

Cette variante est réalisée en suivant les étapes de codage expliquées en haut c'est-à-dire que si par exemple nous avons la taille des mots à lire est égale à 8 donc N= 8, nous allons calculer la taille de la partie préfixe comme suit :

$$taille(preffix) = \frac{\log(N)}{\log(2)}$$

Dans notre cas cette taille est égale à 3, donc la partie préfixe est représentée sur 3 bits. Ensuite nous calculerons la partie suffixe avec les informations fournies par la partie préfixe et aussi par l'indice à compresser. Dans cette compression nous avons une partie qui a une longueur fixe c'est la partie *preffix* et une partie qui a une longueur variable c'est la partie *suffix.*

II.3.2. La compression adaptative :

Cette variante de compression est similaire à celle de compression standard sauf dans une chose, c'est le calcul de la taille de la partie *preffix.* Ici nous calculons la taille du *preffix* avec la formule suivante :

$$L(preffix) = log_2\,(log_2(nbre))$$

Tel que *nbre* : c'est le nombre total des différents caractères du fichier à compresser.

Exemple :

Soit N=8 bits, le nombre de symboles (caractères) possibles à coder avec N est $2^N = 2^8 = 256$ symboles.

Et soit n=16 symboles dont nous avons besoin seulement de coder.

Longueur du préfixe :

- dans le 1er cas : préfixe = $\log_2(N) = \log_2(8) = 3$
- dans le 2ème cas : préfixe = $\log_2(\log_2(16)) = \log_2(4) = 2$

Donc la longueur du préfixe sera 2 au lieu de 3 et nous gagnerons ainsi 1 bit de la taille de préfixe.

II.3.3. La compression récursive :

Un autre code peut être obtenu par le codage de la partie suffixe lorsque celle-ci dépasse une certaine longueur. Dans ce cas c'est la partie préfix qui indique cette taille [*L(suffix) = preffix*], la partie préfix sert de champ de contrôle.

- ✓ **Codage**

Le codage récursif suit les étapes :

- Choisir les valeurs de la partie préfixe (longueur de la partie suffixe) que nous souhaite appliquer le codage récursif

- Appliquer le codage standard. Trouver $C_i = P_i * S_i$

- Si $P_i \in$ [Valeurs choisies] Alors Coder la partie

suffixe S_i .trouver $S_i = P_j * S_j$. Le code résultat est : $C_i = P_i * P_j * S_j$

Sinon

Le code résultat est : $C_i = P_i * S_i$

- ✓ **Décodage**

Le décodage suit les étapes :

- Lire la partie préfix P_i.

- Si $P_i \in$ [Valeurs choisies] Alors :

Lire P_j
Lire S_j (P_j bits)
Calculer $X=2^{P_i}+2^{P_j}+S_j$

Sinon

Calculer $X=2^{P_i}+S_i$

II.4. Analyse et conception de la méthode proposée :

Pour l'analyse et la conception nous avons adopté une architecture flexible, basée sur une approche modulaire.

II.4.1. Définitions :[22]

a- Module : les modules sont des entités indépendantes intégrées dans une architecture pour produire une application.

b- Système : C'est l'ensemble des modules utilisés, ainsi que les relations qu'ils entretiennent entre eux.

L'intérêt de ce type de conception est de concentrer les connaissances liées à une entité logique à l'intérieur d'un module qui est seul habilité à exploiter ces connaissances. L'une des conséquences immédiates est que lorsqu'une maintenance est à effectuer sur une entité logique, celle-ci ne doit concerner qu'un seul module, ce qui simplifie la maintenance.

La conception consiste à élaborer à partir de la spécification du problème une solution informatique.

Stratégie générale : son principe c'est de décomposer le problème en sous-problèmes et identifier les problèmes pertinents. Nous avons deux façons de décomposer :

- Selon les fonctionnalités, traitements (conception fonctionnelle).
- Selon les données à manipuler (conception objet)

Peu importe la méthode de décomposition choisie, la conception peut être :

1) Descendante (top-down) :

L'approche descendante commence par décomposer le problème initial en sous-problèmes puis chaque sous-problème en de nouveaux sous-problèmes et ainsi de suite jusqu'aux problèmes que nous pouvons résoudre par des opérations primitives (ou des fonctions simples).

Exemples:

- Le travail d'une matière brute pour créer un objet plus élaboré : usinage d'une pièce.
- L'ébauche d'un dessin, avant le détail.
- La conception d'un produit en suivant un cahier des charges (décrivant un objectif dans sa globalité), entrant au fur et à mesure dans le détail des exigences : élaboration de circuits intégrés, conception logicielle.

2) Ascendante (bottom-up) :

L'approche ascendante construit des opérations primitives que nous assemblons pour obtenir des opérations plus complexes et ainsi de suite jusqu'à une opération globale qui résout le problème initial.

Exemples :

- L'assemblage de pièces : construction d'une maison, montage d'un kit, briques de Lego.
- En économie, le bottom-up est utilisé pour nommer les processus au sein d'une organisation qui prennent leur origine en bas de l'échelle.
- Dans le cadre de la recherche (terme notamment employé en informatique), l'approche bottom-up tente de faire dévoiler des théories de la pratique.

II.4.2. Analyse et conception de notre travail :

Dans notre application nous allons donner une grande importance au concept du module qui est considéré comme les tâches essentielles réalisées par l'application, la visualisation de ces tâches donne un aperçu général du rôle de l'application. Cette décomposition du problème en modules va nous faciliter le travail et elle nous permet de fixer nos objectifs.

Le rôle de notre application est de réaliser les deux variantes de compression de cette méthode proposée qui sont : la compression standard et la compression adaptative qui ont le même principe sauf dans le calcul de la longueur de la partie *preffix*.

D'après la description donnée au dessus nous déduisons que nous allons utiliser la conception descendante qui consiste à décomposer le problème en sous problèmes.

II.4.2.1. Modules principaux de l'application (le noyau) :

Avant de donner les différents modules qui composent ce système, il est nécessaire de donner les grandes lignes aux quelles ils feront référence, cela afin d'obéir à l'objectif de l'application à créer. Pour ce faire notre application doit alors offrir la possibilité de gérer les différentes tâches et demandes de l'utilisateur, qu'elle doit accomplir à savoir la compression standard, la décompression standard, la compression adaptative, la décompression adaptative, le calcule et affichage des résultats et performances de compression / décompression et éventuellement leurs sauvegarde. Puisque la différence entre les deux compressions est de calculer la longueur de la partie *preffix* donc nous pouvons décomposer le noyau de l'application en trois modules qui sont:

1. Module de compression.
2. Module de décompression.
3. Module de calcul des performances.

La figure qui suit nous montre cette décomposition :

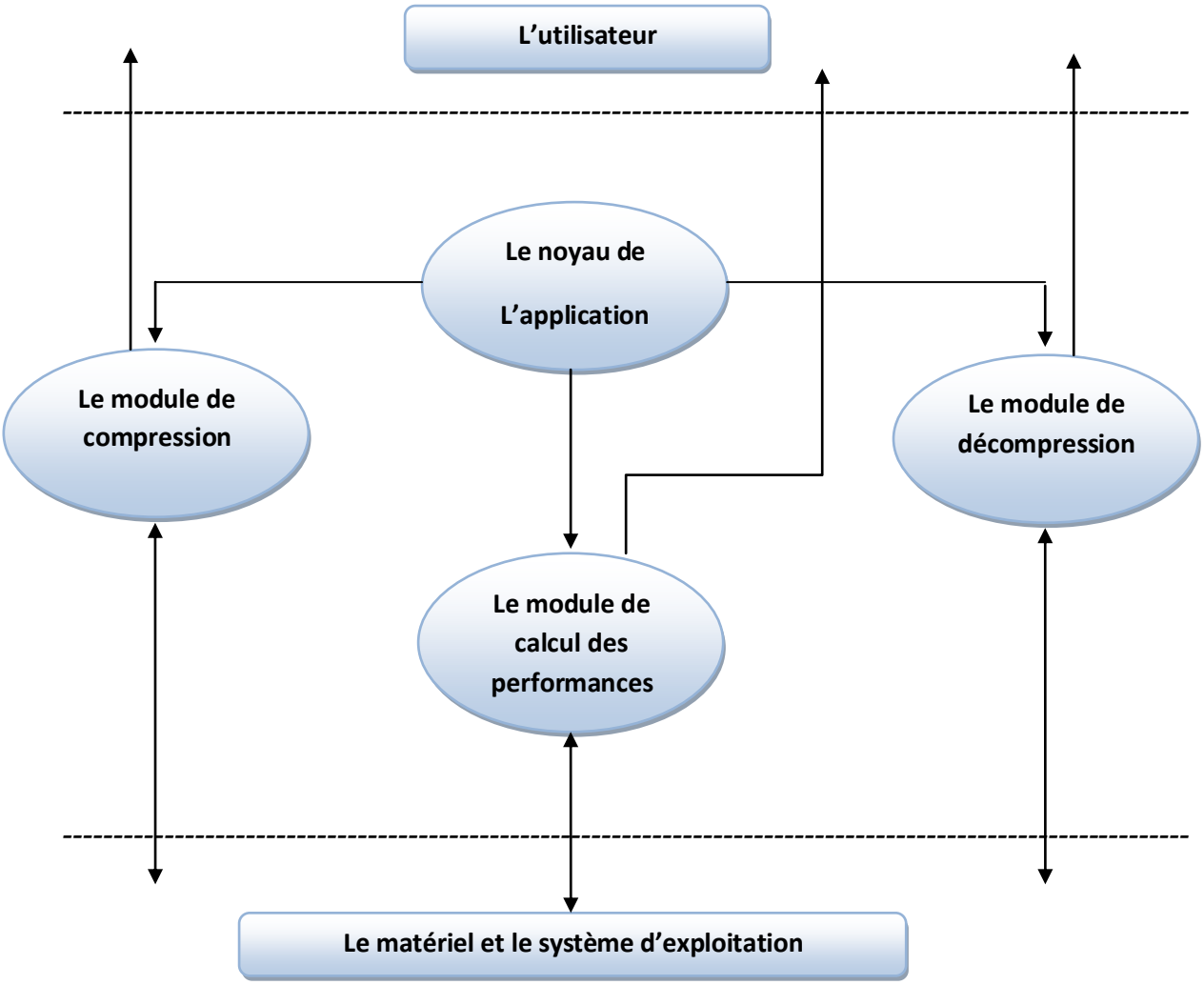


Figure4.1 : Décomposition du module principal de l'application.

II.4.2.2. Module de compression :

Avant de lancer la compression nous devons d'abord faire le choix du fichier à compresser, le système enregistre le chemin de ce fichier. Au lancement de la compression, la fonction de compression calcul premièrement le nombre d'occurrences des caractères dans le fichier, c'est-à-dire calculer les fréquences des caractères ensuite elle fait appel à la fonction d'ordonnancement pour ordonner la table des fréquences en ordre décroissant, après elle fait la table inverse c'est-à-dire elle met chaque indice de la table triée à la case qui correspond au symbole qui se trouve à cet indice.

Ensuite elle appelle la fonction qui va nous retourner la taille de la partie *preffix*, cette dernière dépond du type de compression choisi qui peut être soit standard soit adaptative. L'avant dernière étape est celle de codage, et au lieu de coder les symboles eux-mêmes nous avons codé les indices qu'ils leurs correspondent. Enfin, elle fait l'affectation, chaque symbole elle le remplace par son code.

Après cette description du module de compression, nous pouvons déduire que ce module peut être décomposé en sous modules qui sont les suivants :

1. Calcul des fréquences.
2. Tri et calcul de la table inverse.
3. Calcul de taille du *preffix*.
4. Codage des symboles.
5. Affectation des codes.

La figure ci-dessous nous montre le module de compression et ses sous modules :

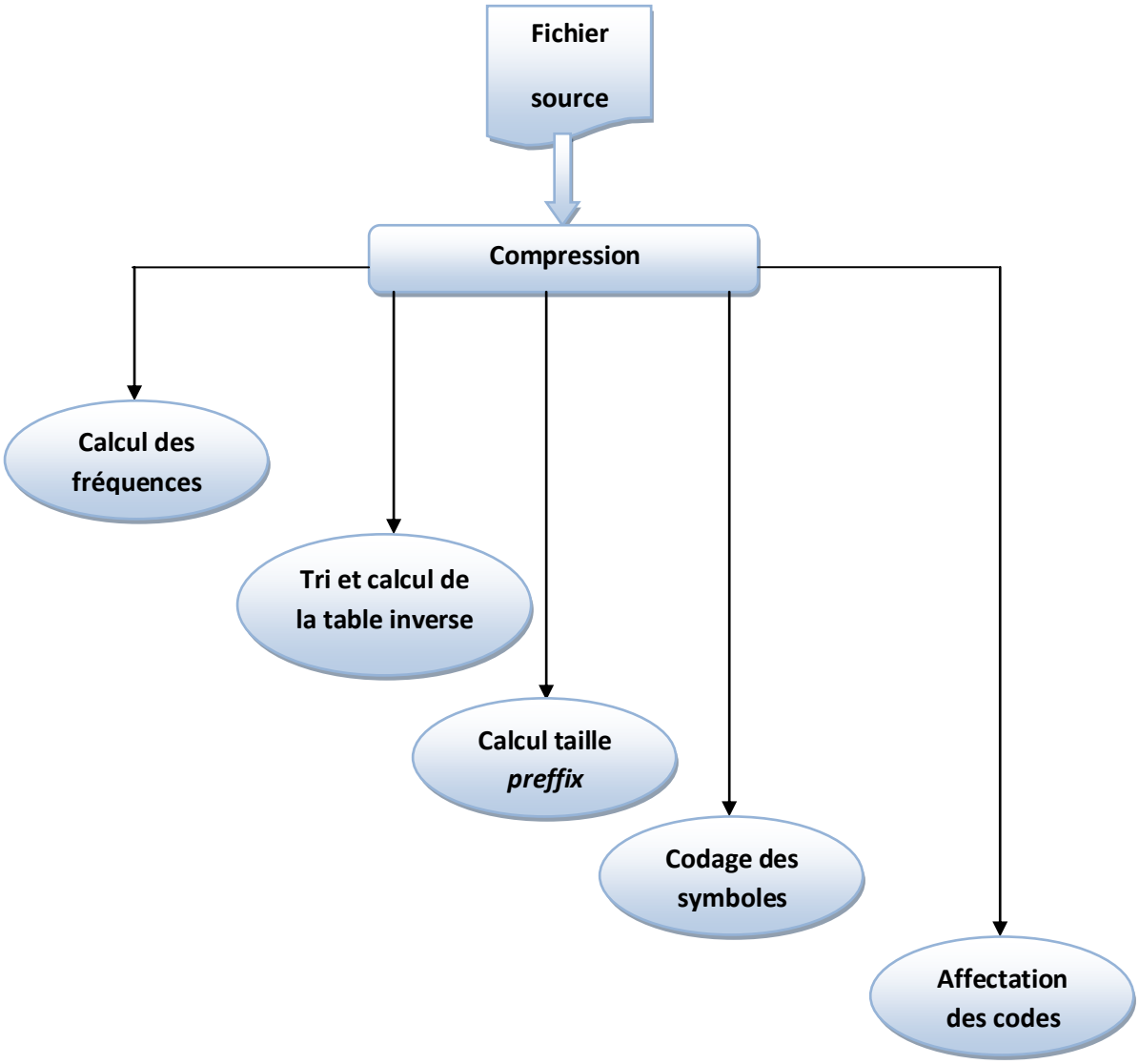


Figure4.2 : Schéma général du module de compression.

Le principe utilisé pour dans cette compression offre la possibilité d'affecter le code de la plus petite longueur pour le symbole le plus fréquent (plus grande probabilité) et le plus grand code pour le symbole le moins fréquent, et si la compression utilisée est celle adaptative le code résultant sera plus petit. Aussi l'utilisateur doit choisir la longueur des mots à lire lors du lancement de la compression.

II.4.2.3. Le module de décompression :

Avant de lancer la décompression, nous devons d'abord choisir le fichier à décompresser et le système va enregistrer le chemin du fichier. Au lancement de la décompression, la fonction va d'abord lire les informations accompagnant le fichier compressé tel que la taille du fichier, le type de compression et la table triée. Ensuite elle va calculer la taille du *preffix* suivant le type de compression, puis elle décode les caractères (données) et enfin elle va rechercher les symboles correspondants.

D'après cette description nous pouvons déduire que le module de décompression est décomposé en sous modules suivants :

1. Lecture des informations.
2. Calculer la taille du *preffix.*
3. Décodage des données.
4. Recherche des symboles.

La figure ci-dessous nous montre le module de décompression et ses sous modules :

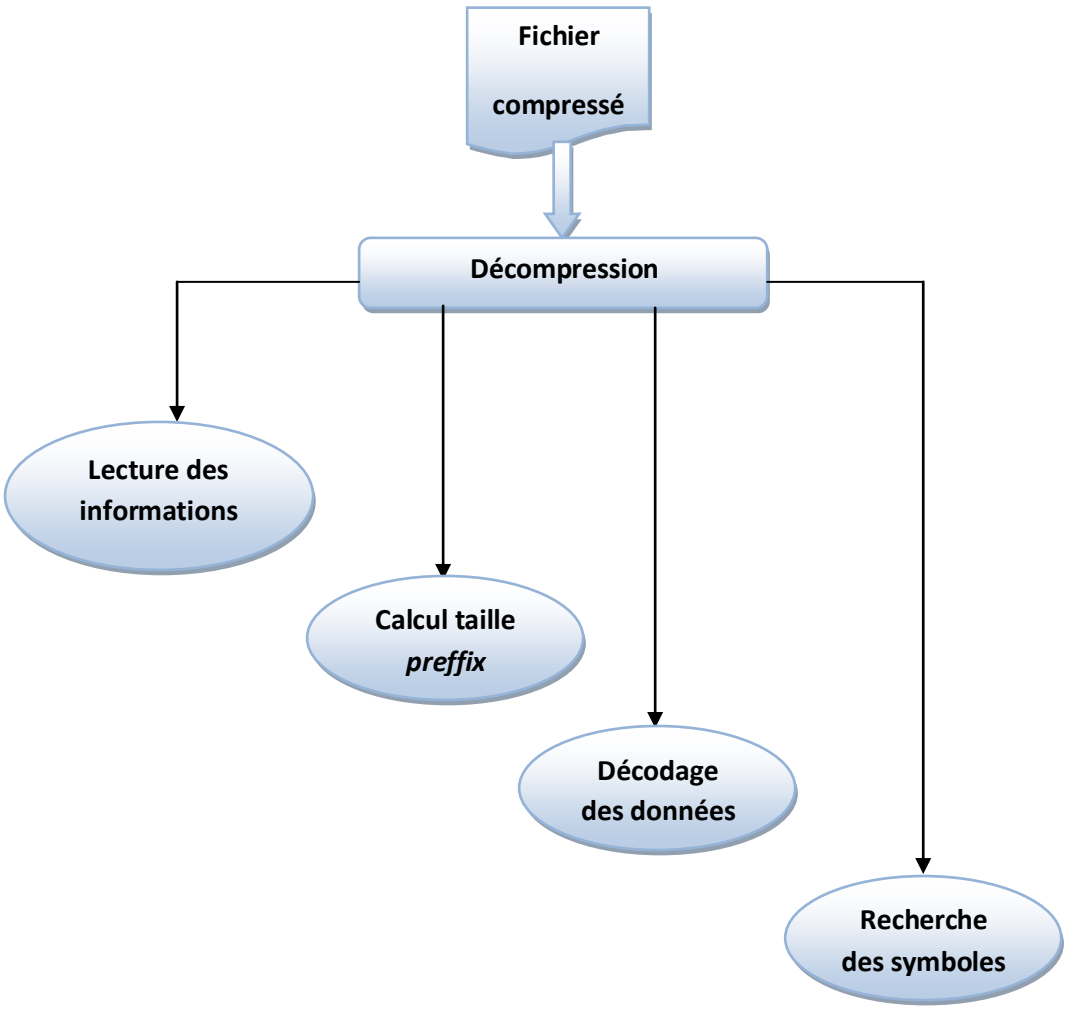


Figure4.3 : Schéma général du module de décompression.

Le principe utilisé pour la décompression permet donc de décompresser un fichier compressé et de retrouver le même fichier source sans aucune perte de données.

II.4.2.4. Module de calcul des performances :

Lors des deux opérations de compression et décompression, il ya un module qui est chargé de calculer les performances de cette opération, à savoir le taux de compression et le temps de compression pour l'opération de compression et seulement le temps de décompression pour l'opération de décompression. A la fin de l'opération le système affiche le résultat de ces calculs. Avec ces résultats nous pouvons évaluer cette nouvelle méthode.

a- Calcul des performances pour la compression :

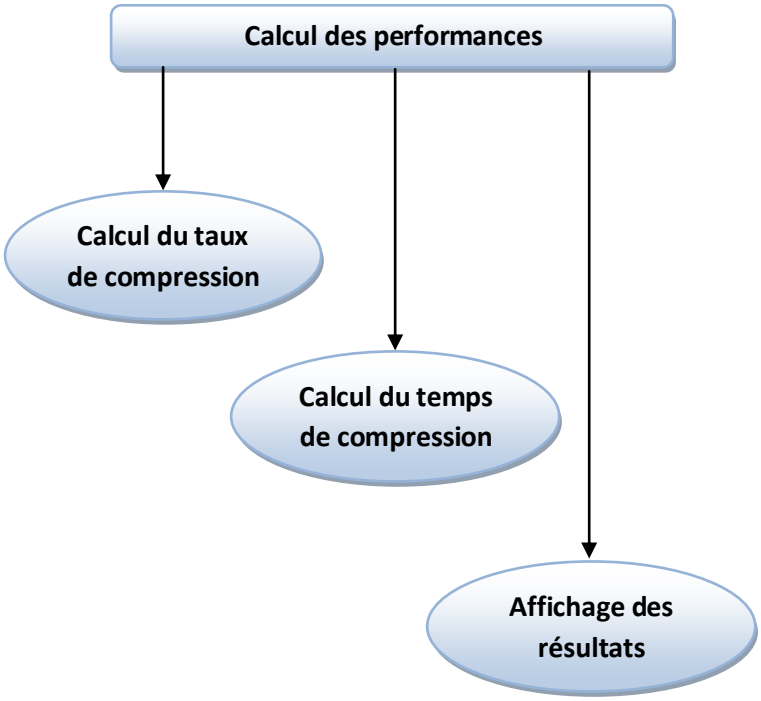


Figure4.4 : Schéma général du calcul des performances pour la compression.

b- Calcul des performances pour la décompression :

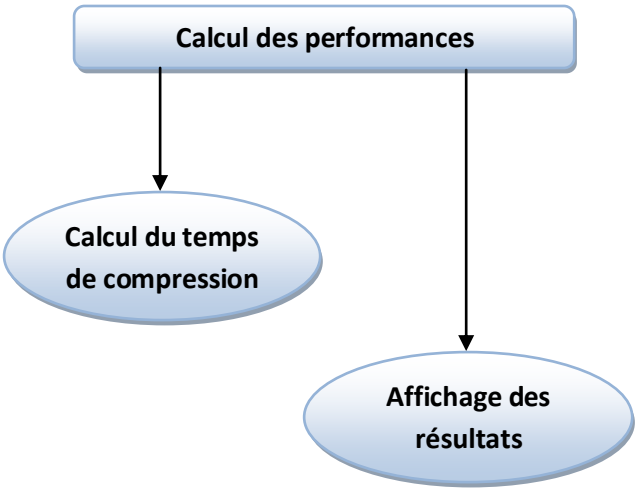


Figure4.5 : Schéma général du calcul des performances pour la décompression.

❖ ***Le temps de compression :***

Pour calculer le temps de compression, le module de calcul des performances utilise l'horloge système au début de la compression pour lui donner le temps de début de la compression. A la fin de la compression il sollicite une autre fois cette horloge pour lui donner le temps de fin de compression. Et pour obtenir le temps exact de compression il fait la soustraction du temps fin de la compression par le temps début de la compression, comme suit :

$$Temps\ de\ compression = temp\ de\ fin - temps\ de\ début$$

❖ ***Le taux de compression :***

Pour calculer le taux de compression, le module de calcul et affichage des performances doit récupérer avant la compression la taille du fichier d'origine (source) celui à compresser et le fichier compressé (cible), puis il calcule le taux comme l'indique l'équation suivante:

$$\text{Taux} = \frac{(\text{taille originale} - \text{taille finale}) * 100}{\text{taille originale}}$$

III. Analyse et conception de la méthode de Huffman :

La compression avec la méthode de Huffman a été présentée en détail dans le chapitre consacré pour cette méthode et son principe de base est l'utilisation des arbres binaires. Dans notre application nous avons utilisé deux variantes de cette méthode en même temps à savoir celle dynamique et celle locale.

Pour ce qui concerne le Huffman dynamique, il est illustré par le calcul des occurrences des caractères pour chaque fichier à compressé contrairement à celui statique où les tables des fréquences sont définies pour chaque type de fichier, donc le codeur contient déjà ces tables. Et pour ce qui concerne le Huffman local, il est illustré par l'offre du choix de longueur de mot à lire à partir du fichier à compressé (source) plus de chance de trouver des redondances du code.

III.1. Module principaux de l'application :

D'une manière générale, nous pouvons dire que le noyau de cette application est constitué des mêmes modules principaux que ceux de la méthode proposée, c'est-à-dire que la méthode de Huffman doit permettre à l'utilisateur de faire la compression de Huffman, la décompression de Huffman et le calcul et l'affichage des performances de cette méthode. Donc nous pouvons dire que cette méthode est décomposée en trois modules principaux :

1. La compression de Huffman.
2. La décompression de Huffman.
3. Le calcul et l'affichage des performances.

La figure suivante nous montre cette décomposition :

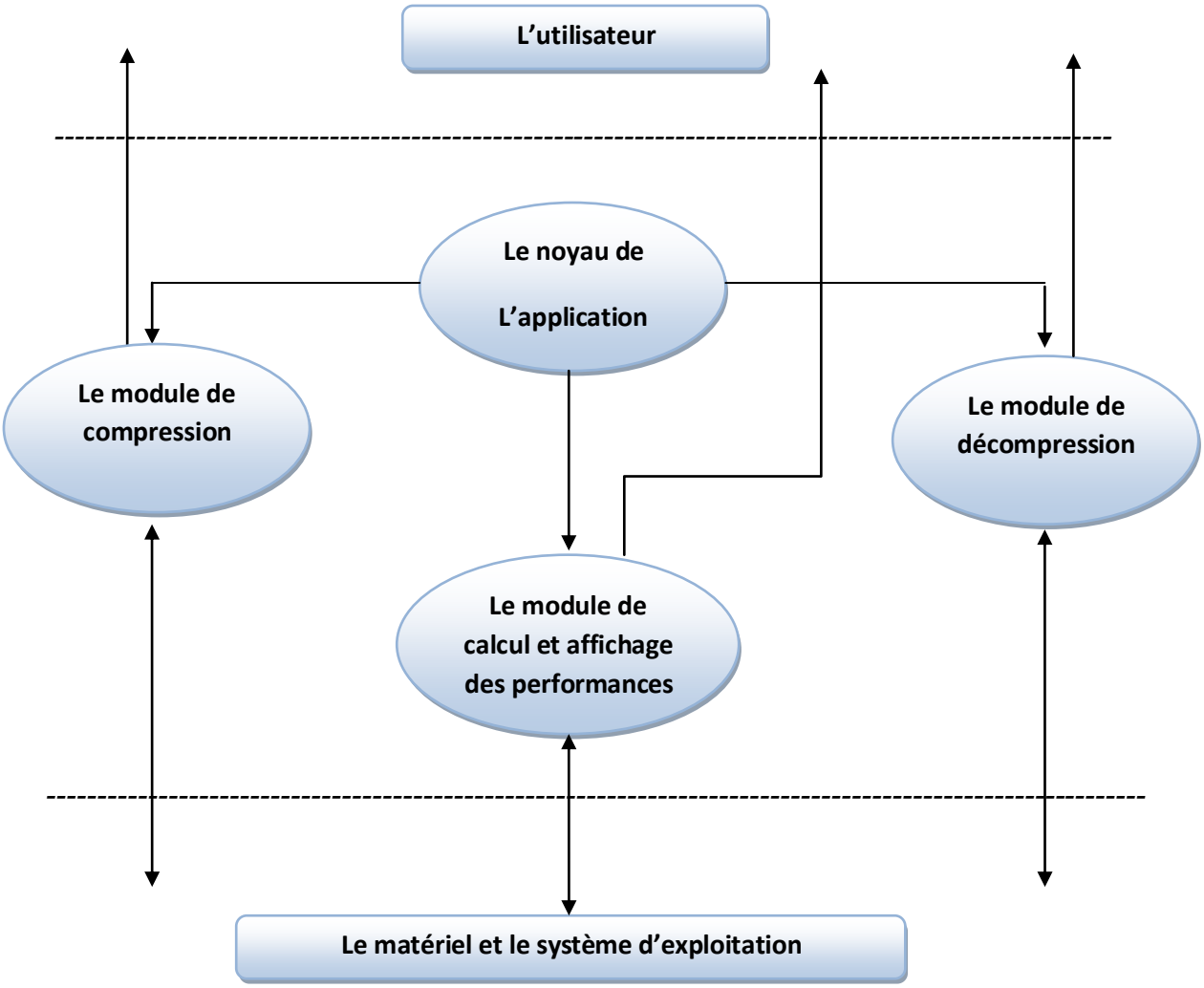


Figure4.6 : Décomposition du module principal de l'application.

III.2. Le module de compression de Huffman :

Avant de lancer l'opération de compression l'utilisateur doit d'abord choisir le fichier à compressé et le système va enregistrer son chemin. Au lancement de la compression, ce module va lire au premier lieu le fichier et établir la table des fréquences (occurrences) des caractères du fichier. Ensuite, il fait appel à la fonction qui va construire l'arbre de Huffman en suivant la table des fréquences obtenue auparavant. Après, il appelle la fonction qui va s'occuper de faire la correspondance entre les symboles et leur codes obtenus à partir de l'arbre de Huffman. En dernier lieu le module va faire l'affectation des codes aux symboles.

Donc, nous pouvons déduire que ce module peut être décomposé en sous modules suivants :

1. Calcul des fréquences.
2. Construction de l'arbre de Huffman.
3. Etablissement de la table de correspondance.
4. Affectation des codes aux symboles.

La figure ci-dessus donne le schéma général de cette décomposition :

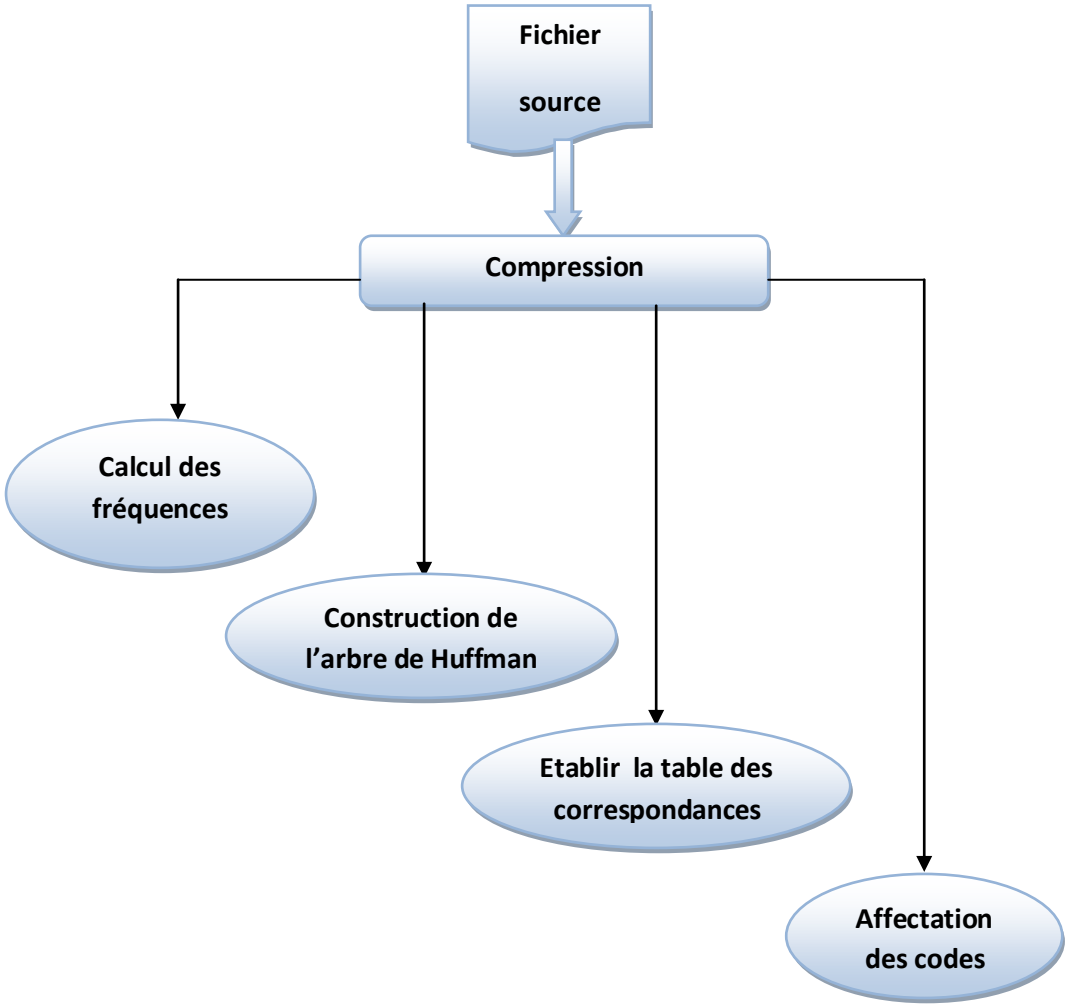


Figure4.7 : Schéma général du module de compression.

Avec le principe de l'arbre de Huffman le module va affecter les codes les plus petits aux caractères les plus fréquents et ceux qui sont grands aux caractères les ont des fréquences petites. En plus de ça, l'utilisateur a le choix de la longueur des mots à lire à partir du fichier source, c'est le Huffman local.

III.2. Le module de décompression :

Après le choix du fichier à décompressé, celui-ci doit être évidemment compressé avant avec la méthode de Huffman le module de décompression va lire en premier lieu l'en-tête accompagnant le fichier compressé qui contient les informations qui vont nous aider lors de la décompression, tel que la taille du fichier, l'arbre de Huffman, le nom original du fichier. Après la récupération des informations utiles pour la décompression, le module va décoder les données et ensuite il va rechercher les caractères correspondant données décodées. Donc le module de décompression est décomposé en ces sous modules :

1. Lecture de l'en-tête.
2. Décodage des données.
3. Recherche des symboles.

Voici la figure qui montre la décomposition du module de décompression :

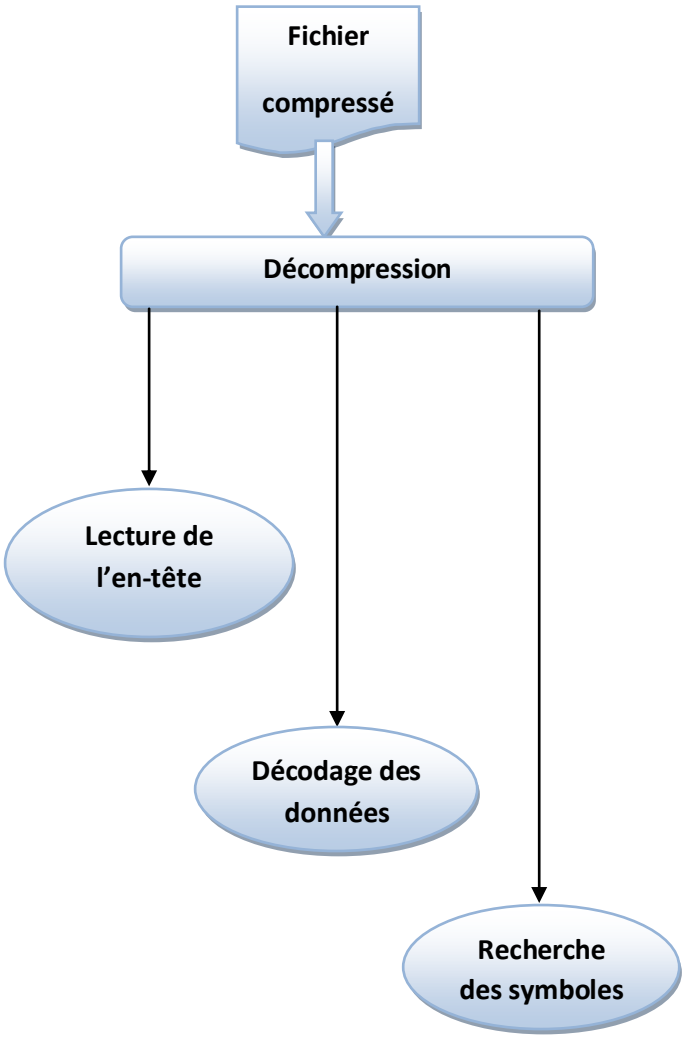


Figure4.8 : Schéma général du module de décompression.

III.3. Le module de calcul et affichage des performances :

Ce module est similaire à celui de la méthode proposée, c'est-à-dire qu'il calcul le temps et le taux de compression pour la compression, et il calcul le temps de décompression pour la décompression et puis il les affiche. Les résultats obtenus par ce module sont utilisés pour évaluer les performances de cette méthode. Nous avons les mêmes schémas que ceux de la méthode proposée.

IV. Conclusion :

Dans ce chapitre nous avons présenté la partie d'analyse et de conception de notre projet, cette étape est essentielle car elle nous aide à organiser notre application et faire sortir les grandes lignes, et aussi pour fixer les objectifs à atteindre. Ici nous avons fait l'analyse et la conception des deux méthodes à utiliser dans notre étude. Nous avons considéré chaque méthode comme étant un problème (noyau) et nous l'avons décomposé en sous problèmes (modules) et chaque module en sou modules, tout ça est accompagné par des schémas. Dans le chapitre qui suit nous allons faire l'implémentation des deux applications en se basant sur cette conception.

Chapitre V

Implémentation et évaluation

I. Introduction :

Dans ce chapitre nous allons présenter dans une première partie l'étape de l'implémentation et réalisation de notre travail, qui est la mise en œuvre des deux méthodes dont nous avons besoin ; celle proposée et celle de Huffman. Dans la deuxième partie nous allons faire des tests. Ces test sont fait afin d'évaluer ces méthodes et ils sont fait à base de plusieurs critères. A chaque fin d'un test, nous allons comparer les résultats et déduire la meilleure méthode parmi les deux. C'est critères sont :

- Taux de compression.
- Temps de compression.
- Type de fichier.
- Taille du mot à lire.

II. Environnement de développement :

Nous avons choisi pour la réalisation de notre travail le système d'exploitation Microsoft Windows XP Professionnel. Ce choix est fait car les systèmes d'exploitation Windows sont très utilisés grâce à leurs simplicités, efficacités, convivialités et fiabilités. L'ordinateur sur lequel nous avons fait ces évaluations est doté d'un processeur de fréquence 2.2 GHz et d'une mémoire de 2G.

Pour ce qui concerne la programmation, nous avons choisi le langage java et Eclipse GALILEO comme environnement de développement.

II.1. Description du langage JAVA :

Java est un langage de programmation récent (les premières versions datent de 1995) développé par Sun Micro Systems. Il est inspiré des langages C et C++. Comme C++, Java fait partie de la « grande famille » des langages orientés objets. Il répond donc aux trois principes fondamentaux de l'approche orientée objet (POO) : l'encapsulation, le polymorphisme et l'héritage. Java a rapidement intéressé les développeurs pour quatre raisons principales :

- C'est un langage orienté objet dérivé du C, mais plus simple à utiliser.

- Il est doté, en standard, de bibliothèques de classes très riches comprenant la gestion des interfaces graphiques (pour tester d'une manière conviviale les programmes), la programmation multithread (multitâche), la gestion des exceptions, les accès aux fichiers et au réseau.

- Il est doté en standard d'un mécanisme de gestions des erreurs (les exceptions) très utile et très performant.

- Il est multi plates-formes : les programmes tournent sans modification sur tous les environnements où Java existe (Windows, Unix et Mac).

- Une documentation de projet peut être générée, pour cela, chaque classe et méthode peuvent être commentées selon les conventions « JavaDoc ».

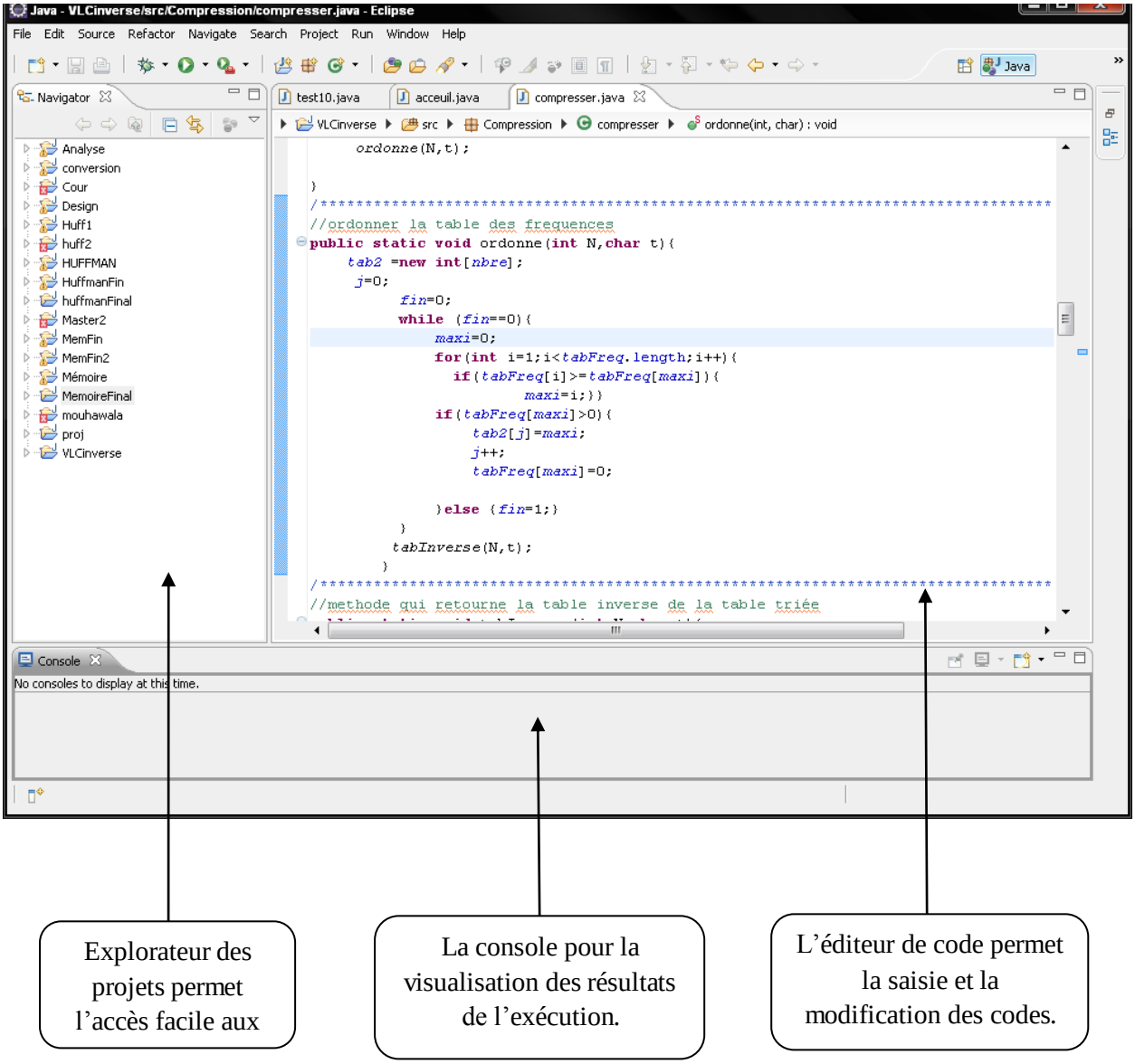
II.2. Description d'Eclipse (Galileo) :

Eclipse est un environnement de développement intégré libre extensible, universel et polyvalent, permettant de créer des projets de développement mettant en œuvre n'importe quel langage de programmation. Eclipse IDE est principalement écrit en Java (à l'aide de la bibliothèque graphique SWT d'IBM), et ce langage, grâce à des bibliothèques spécifiques, est également utilisé pour écrire des extensions. La spécificité d'Eclipse IDE vient du fait de son architecture totalement développée autour de la notion de plugin (en conformité avec la norme OSGi), toutes les fonctionnalités de cet atelier logiciel sont développées en tant que plug-in.

La base de développement d'Eclipse est l'Eclipse Platform, composée de :

- Platform Runtime démarrant la plateforme et gérant les plug-ins.
- SWT la bibliothèque graphique de base de l'EDI.
- Face, une bibliothèque graphique de plus haut niveau basée sur SWT.
- Eclipse Workbench, la dernière couche graphique permettant de manipuler des composants, tels que des vues, des éditeurs et des perspectives.

A la création d'un nouveau projet, Eclipse crée deux dossiers l'un **(scr)** pour contenir les code sources **(.java)** des classes l'autre (**bin)** pour contenir les codes des classes après compilation **(.class)** chaque projet contient la fonction principale **Main ().**



Explorateur des projets permet l'accès facile aux

La console pour la visualisation des résultats de l'exécution.

L'éditeur de code permet la saisie et la modification des codes.

Figure5.1 : L'interface de l'environnement Eclipse Galileo.

II.3. L'architecture des applications :

Les deux applications de notre projet sont séparées en deux parties principales :

- La « Boîte à outils » qui contient les algorithmes de compression et de décompression.
- L'interface graphique qui permet uniquement d'exécuter et de comprendre le fonctionnement des procédés.

III. Présentation des applications :

Dans ce point nous allons présenter les interfaces graphiques des deux applications. Ces interfaces sont simples et conviviales afin qu'elles soient facile à utiliser par de simple utilisateurs, en plus ce sont le lien entre l'utilisateur et la boite à outil des applications.

III.1. La méthode proposée :

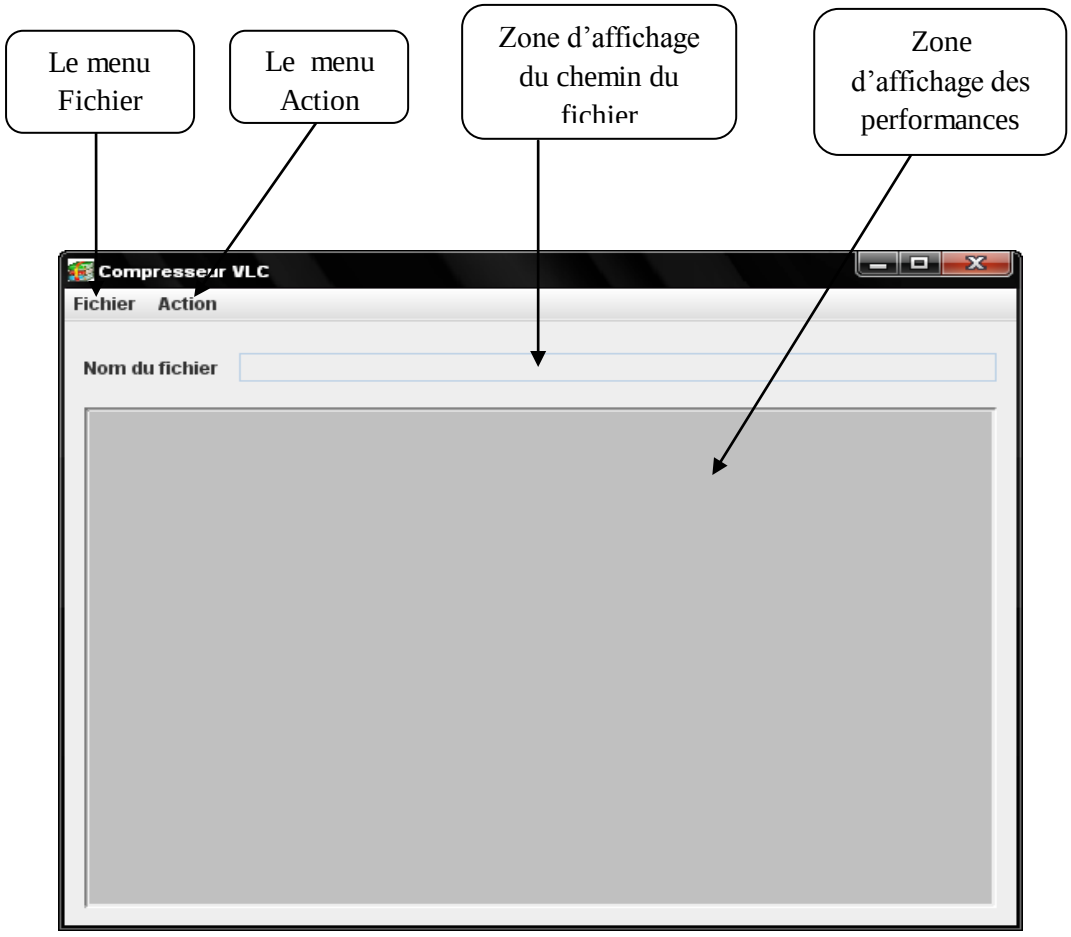


Figure5.2 : L'interface de la méthode proposée.

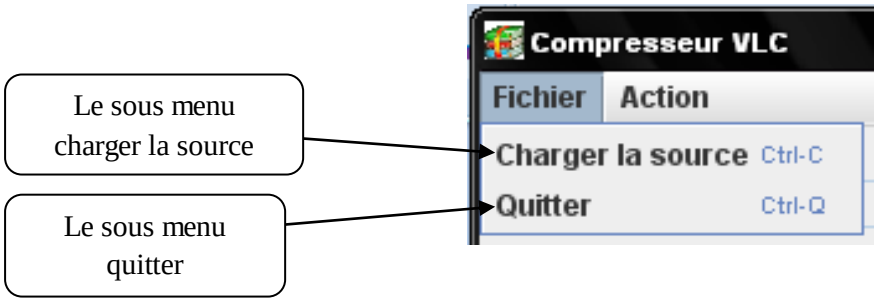


Figure5.3 : Le menu Fichier.

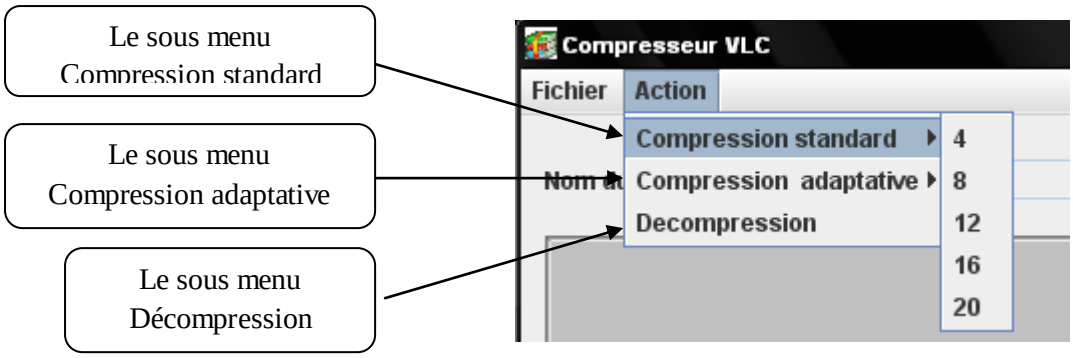


Figure5.4 : Le menu Action.

Le sous menu « Compression adaptative » a les mêmes sous menus que la sous menu « Compression standard », et ces valeurs représentent la longueur du mot à lire.

III.2. La méthode de Huffman :

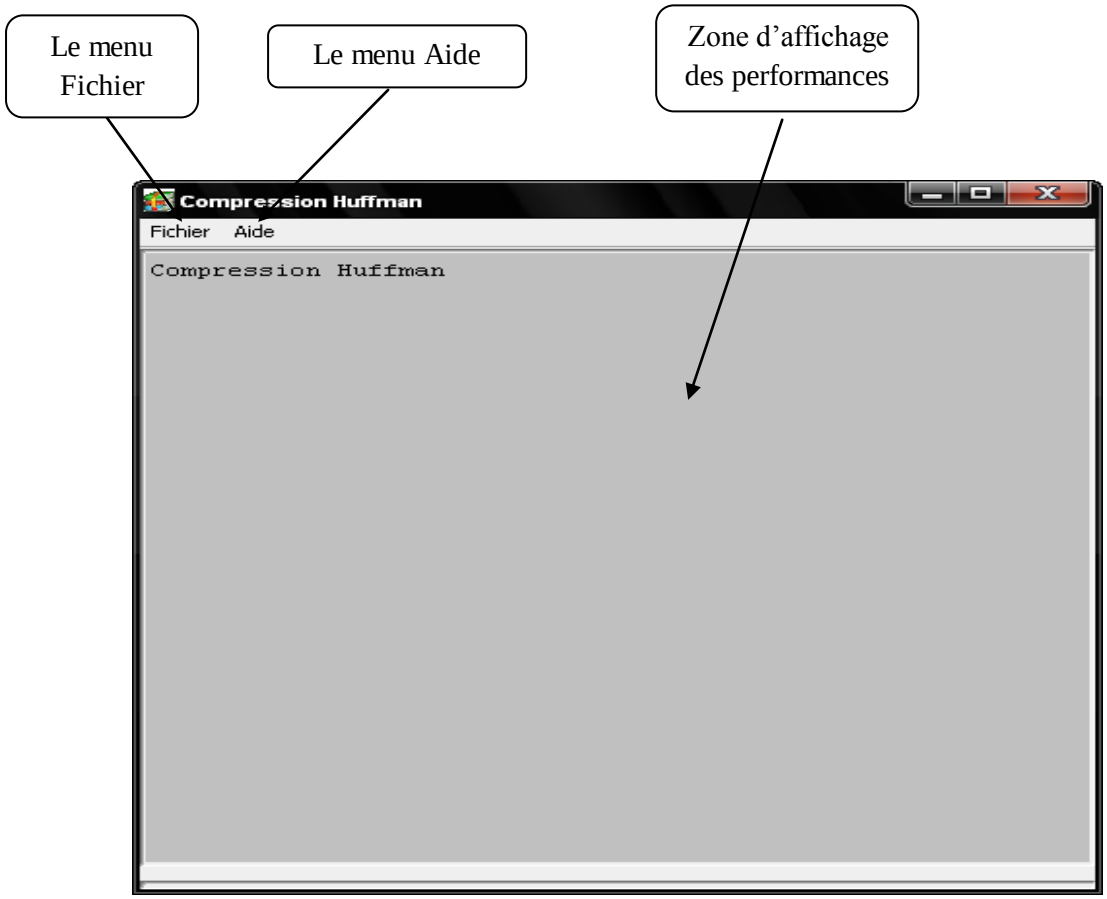


Figure5.5 :L'interface de la méthode de Huffman.

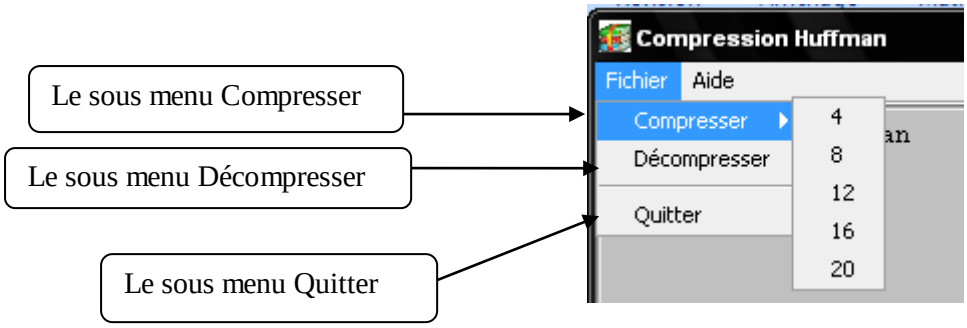


Figure5.6 : Le sous menu Compresser.

Les sous menu du sous menu Compresser représente la longueur du mot à lire.

III.3. Le noyau des applications :

III.3.1.La méthode proposée :

Le noyau de cette application est constitué de plusieurs fonctions et chacune a un rôle dans l'accomplissement des buts de l'application. Ces fonctions principales sont :

- void choisir () : cette fonction est utilisée afin de sélectionner un fichier soit pour la compression, soit pour la décompression.

- void lecture () : le rôle de cette fonction est de parcourir tout le fichier et lire octet par octet.

- void getFreq (byte[] texte) : cette fonction a comme paramètre le fichier mis dans un tableau de bytes (octets), elle calcule les fréquences.

- void calculCar() : le rôle de cette fonction est de calculer le nombre de caractères qui se trouve dans un fichier. Cette fonction est utile pour la compression adaptative car on utilise son résultat pour calculer la longueur de la partie *preffix*.

- void ordonner () : cette méthode est appelée après le calcul des fréquences car son rôle est de trier et ordonner la table des fréquences.

- void tabInverse () : son rôle est de remplir la table inversée de la table triée, c'est-à-dire parcourir la table triée et mettre l'indice de la case dans la case qui correspond au contenu de cette case. Elle facilite la compression.

- void compresser() : cette fonction s'occupe de la lecture du fichier source et la création du fichier final, fait les appels à la fonction de codage est écris le résultat dans le fichier final.

- String codage (int a, int Lp) : cette fonction a le rôle de retourner sous une chaîne de caractères (chaîne binaire) le code de l'entier « a » tout en connaissant la longueur du *preffix* « Lp » à la fonction de compression.

- void decompresse () : le rôle de cette fonction est comme son nom l'indique, elle parcourt le fichier compressé, lit l'en-tête qui l'accompagne puis elle appelle la fonction du décodage et écris dans le fichier décompressé.

- String decodage (String ch) : cette fonction prend en paramètre une chaîne de caractères en binaire fourni par la fonction « decompresse » et la décode pour trouver le symbole correspondant.

- void TauxComp () : cette fonction comme son nom l'indique calcul le taux de compression à chaque fin de compression.

III.3.2. La méthode de Huffman :

Cette application contient plusieurs fonctions mais nous n'allons pas les citer toutes, nous allons seulement citer les plus importantes qui sont :

- File selectFile () : cette fonction c'est pour sélectionner un fichier et retourne le fichier sélectionné.

- void getFreqTable () : elle est appelé pour calculer la table des fréquences.

- Void buildHuffmanTree() : cette fonction s'occupe de la construction de l'arbre de Huffman.

- void getCorresp() : elle fait la correspondance entre les caractères et leurs codes.

- void encodeFile (DataInputStream dis, String[] corresp) : son rôle est de parcourir le fichier à compresser par le flux de lecture « dis » qui est en paramètre et ensuite trouver les codes des caractère dans la table « corresp ».

- void writeHeader () : cette fonction est appelée avant la compression pour écrire l'en-tête dans le fichier compressé.

- void readHeader() : elle est appelée pour lire l'en-tête du fichier compressé avant de le décompresser.

- void decodeFile() : elle s'occupe de la tâche de décompression.

- void showStat() : son rôle est de calculer et afficher les performances de la compression et la décompression.

IV. Evaluation et comparaison des deux méthodes :

IV.1. Evaluation du taux de compression :

Pour évaluer le taux de compression nous allons utiliser les résultats donnés par les applications, ces derniers sont calculés par une fonction spéciale. L'équation suivie pour calculer le taux est la suivante :

$$Taux\ de\ compression = \frac{(Taille\ originale - Taille\ finale) * 100}{Taille\ originale}$$

Avec cette équation nous allons trouver le résultat sous forme de pourcentage.

Nous devons faire cette évaluation et comparaison par rapport à deux critères : le type de fichiers et aussi à la longueur du mot à lire.

Dans cette étude comparative nous allons comparer entre la compression adaptative de la méthode à longueur variable proposée et la méthode de Huffman, car c'est plus intéressant que de la faire avec la compression standard puisque cette dernière ne peut pas donner un résultat meilleur que la compression adaptative.

Dans cette comparaison, nous avons pris pour chaque type de fichier quelque soit texte, image, son ou bien semi structuré un ensemble de fichier de même type mais de taille différentes et nous avons fait les tests sur chacun d'eux, puis nous avons fait la moyenne. Pour chaque type de fichier, nous avons fait varier les valeurs de N, où N est la longueur du mot à lire et ses valeurs sont : 4, 8, 12, 16 et 20.

- ***Les fichiers « textes » :***

Nous commençons par les fichiers textes et nous allons faire varier la valeur de N. Nous avons pris un ensemble de fichiers textes, a taille moyenne de ces fichier est de 559566 octets Les résultats obtenus sont représentés sur la figure suivante :

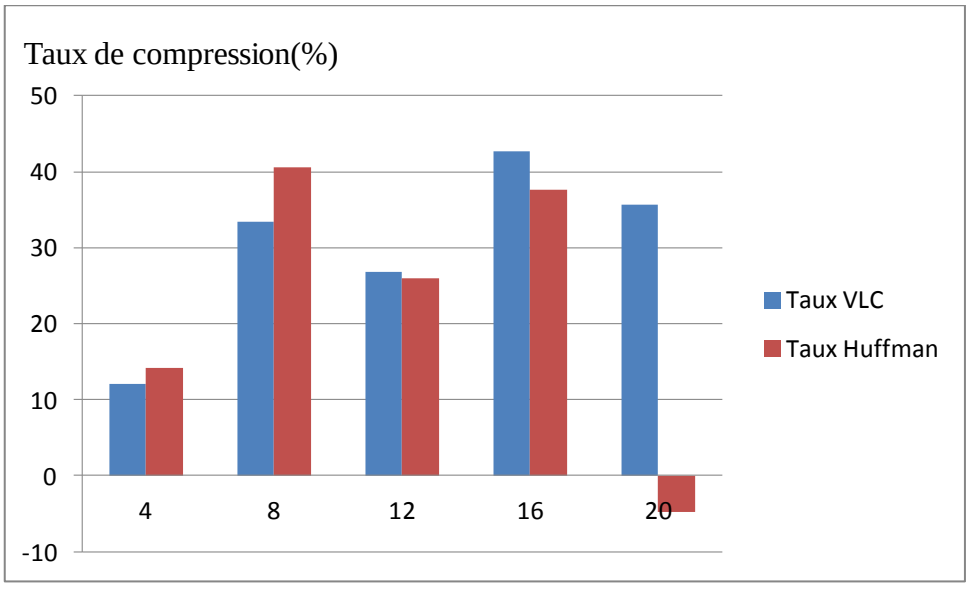


Figure5.7 : Evaluation du taux de compression pour les fichiers textes.

Nous remarquons à partir de la figure précédente que pour les deux valeurs de N 4 et 8 le taux de compression de la méthode de Huffman est un peu mieux que celui de la méthode VLC proposée. Mais pour les valeurs 12, 16 et 20 le taux est meilleur pour la méthode VLC surtout pour la valeur 20 où nous remarquons que le taux est négatif.

- ***Fichier « son » :***

Dans cette parties nous avons évalué le taux de compression pour les fichiers son de format WAV. Ce format est un format de fichier audio numérique non compressé crée par Microsoft principalement utilisé par Windows. La taille moyenne de l'ensemble des fichiers est de 176376 octets. Les résultats obtenus sont représentés sur la figure ci-dessous :

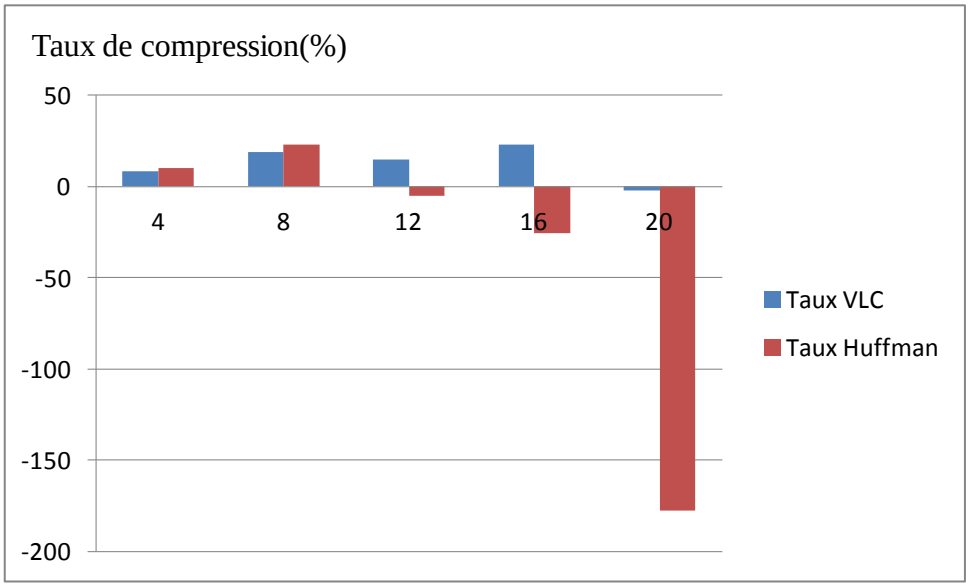


Figure5.8 : Evaluation du taux de compression pour les fichiers son.

Nous remarquons à partir du schéma précédent le taux de compression pour la méthode de Huffman est meilleur pour les valeurs 4 et 8, mais pour les valeurs 12, 16 et 20 le taux pour la méthode VLC est mieux contrairement à la méthode de Huffman où les taux sont négatifs.

- ***Fichier « semi structuré » :***

Pour cette évaluation nous avons pris des fichiers semi structuré (d'extension MHTML). Nous avons fait ces essais car dans ce type de fichier nous retrouvons beaucoup de redondances à cause de leurs codes. La taille moyenne de ces fichiers est de 267365 octets. La figure suivante illustre les résultats obtenus :

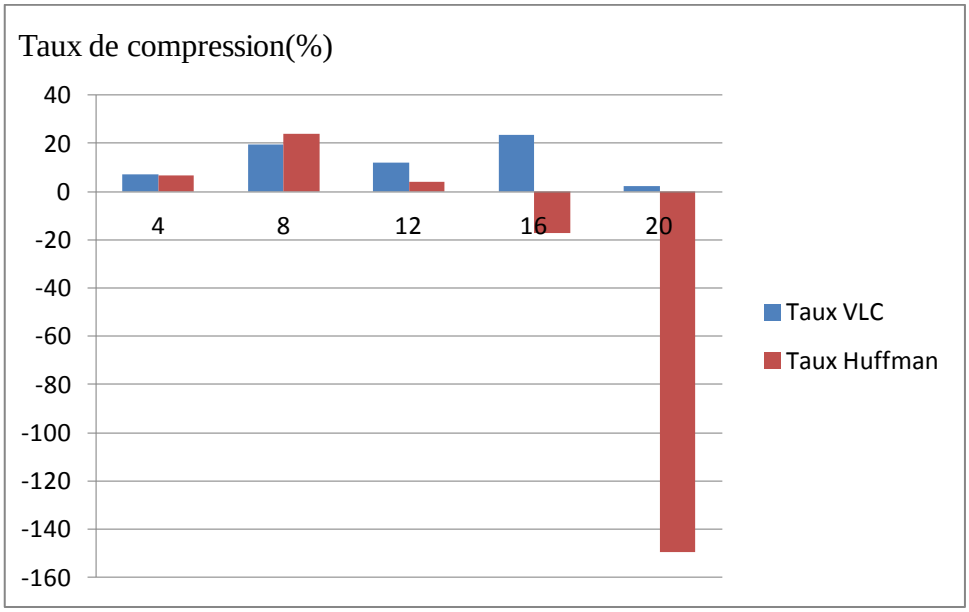


Figure5.9 : Evaluation du taux de compression pour les fichiers semi structurés.

Nous remarquons de la figure précédente que le taux de compression pour la méthode de Huffman est meilleur seulement pour la valeur 8 et pour la valeur 4 le taux est presque le même pour les deux méthodes, mais pour les valeurs 12, 16 et 20 le taux pour la méthode VLC est visiblement beaucoup plus meilleur.

- ***Fichier « images » :***

Pour l'évaluation du taux de compression des fichiers image nous avons pris une collection d'image de format bitmap car ce format d'image n'est pas compressé contrairement aux autres formats tels que JPEG, GIF, PNG…..etc. La taille moyenne de ces images est de 502532 octets et les résultats sont représentés sur la figure suivante :

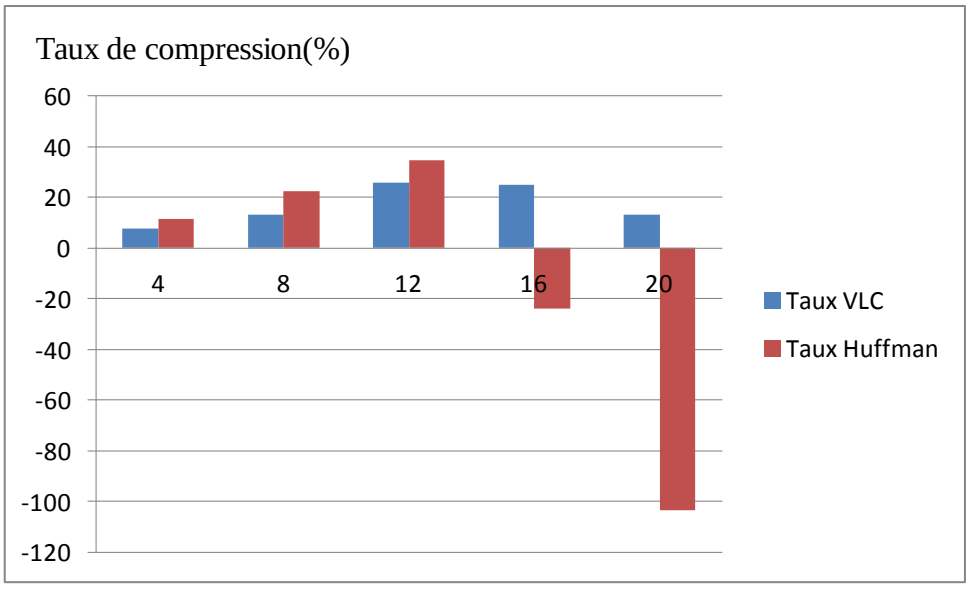


Figure5.10 : Evaluation du taux de compression pour les fichiers images.

A partir de la figure ci-dessus nous constatons que pour la compression des fichiers image, nous avons un taux de compression qui est meilleur pour la méthode de Huffman dans les trois cas où le N est égal à 4, 8 et 12 contrairement aux deux autres valeurs 16 et 20 où nous avons le taux pour la méthode de Huffman qui est non seulement inférieur par rapport à celui de la méthode VLC mais il est négatif.

IV.2. Evaluation du temps de compression :

Dans cette partie d'évaluation, nous allons comparer les deux méthodes par rapport à leur temps d'exécution. Les valeurs du temps représentées dans les figures qui suivent sont ceux calculées à partir des tests précédents pour calculer les taux de compression. Nous avons fait la moyenne des temps obtenus pour chaque type de fichier et pour chaque valeur de N. Le temps est représenté en seconde.

- ***Fichier « textes » :***

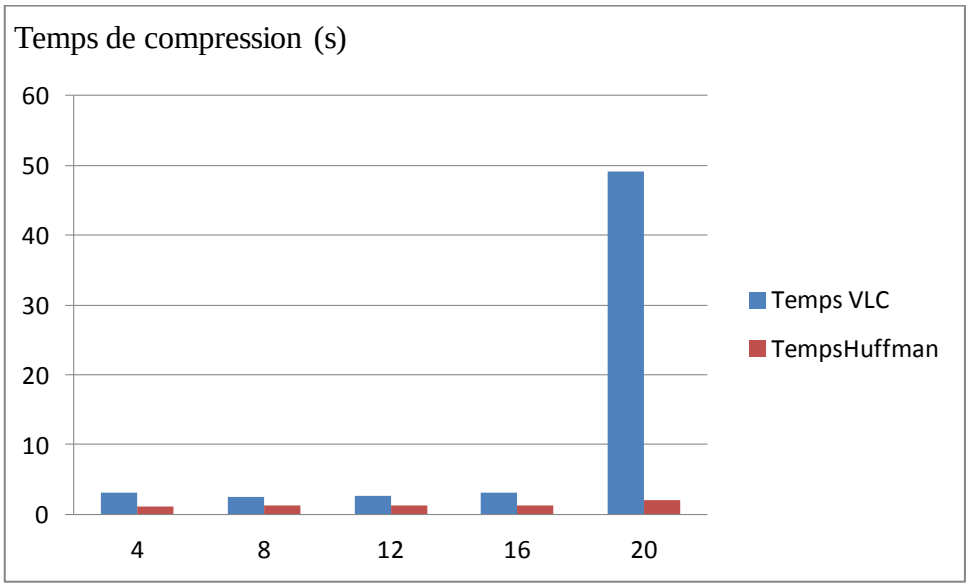


Figure5.11 : Variation du temps de compression des fichiers texte.

- ***Fichier « son » :***

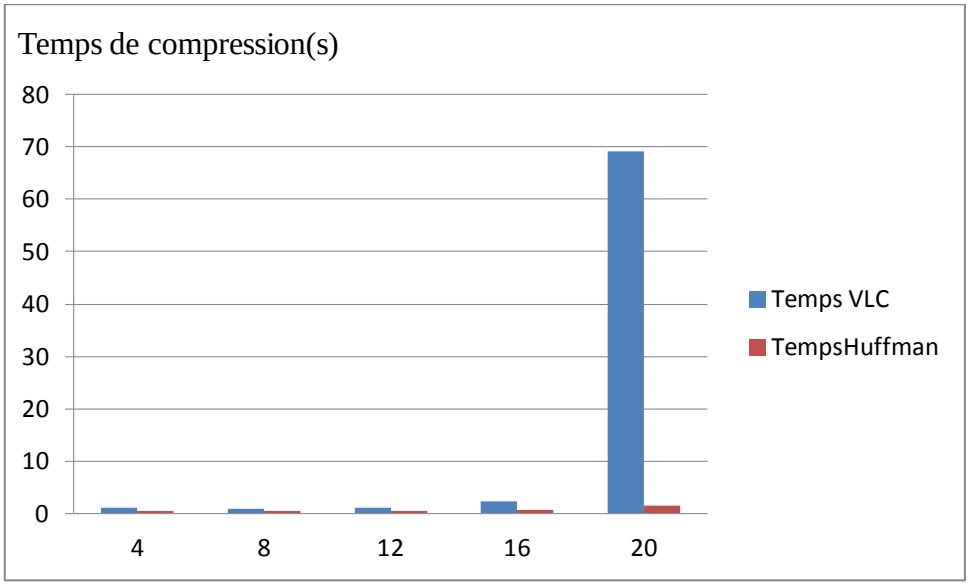


Figure5.12 : Variation du temps de compression des fichiers son.

- ***Fichier « semi structuré » :***

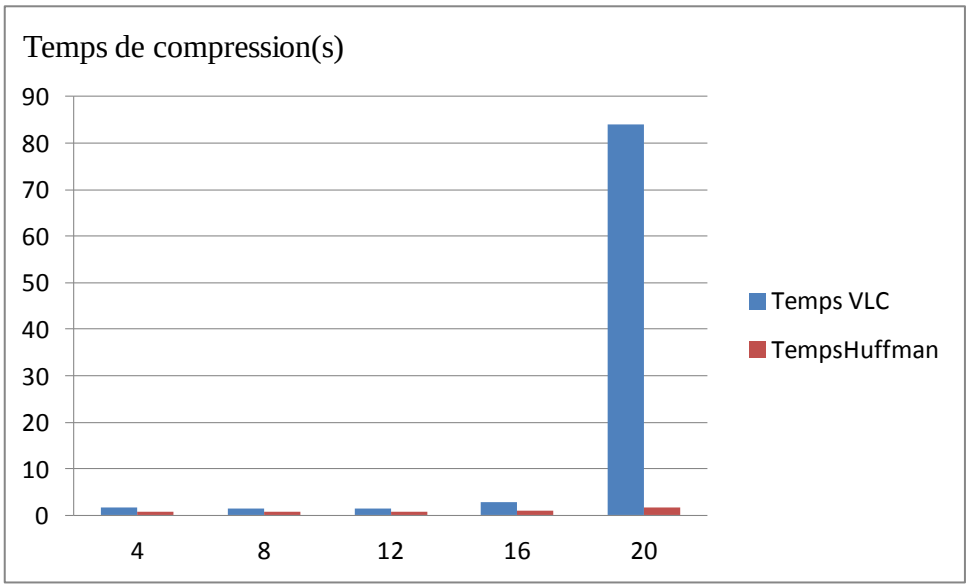


Figure5.13 : Variation du temps de compression des fichiers semi structurés.

➢ ***Fichier « image » :***

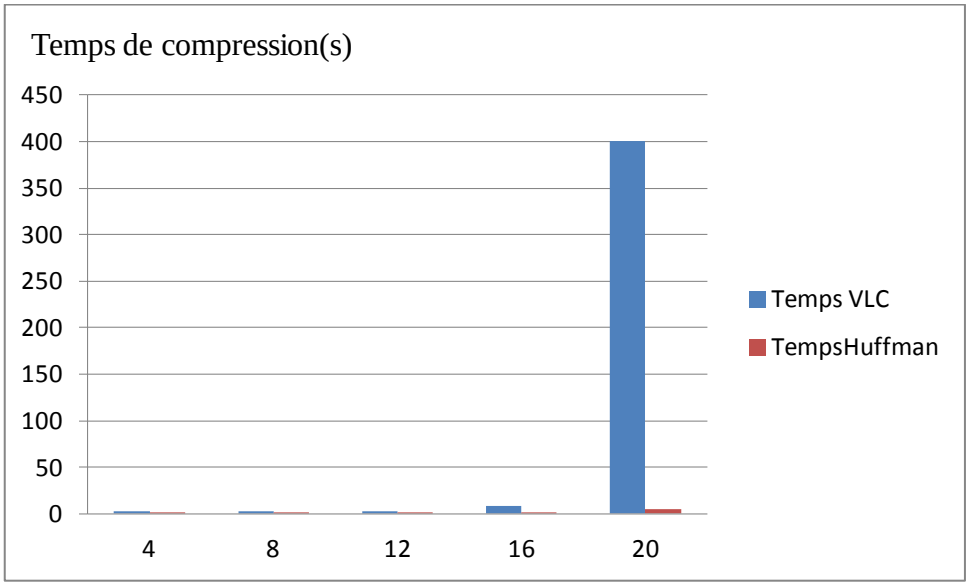


Figure5.15 : Variation du temps de compression des fichiers images.

Interprétation des figures de l'évaluation du temps de compression :

Nous remarquons à partir des 4 figures précédentes que le temps de compression est toujours meilleur pour la méthode de Huffman quelque soit le type du fichier ou bien la valeur de N. Pour les valeurs 4, 8, 12 et 16 la différence est presque négligeable, mais la grande différence se voit pour la valeur 20 et c'est le même cas pour tous les types de fichier.

V. Comparaison des deux méthodes :

D'après les tests d'évaluation que nous avons fait sur les deux méthodes de compression, et après avoir vu les résultats obtenus, nous pouvons constater les points suivants :

- ✓ Pour ce qui concerne le taux de compression, nous avons une meilleure compression avec la méthode de Huffman pour les deux valeurs 4 et 8, mais pour les trois autres valeurs 12, 16 et 20 le taux est beaucoup meilleur avec la méthode VLC proposée, sauf de le cas de compression d'images où nous avons un taux meilleur pour la méthode Huffman pour la valeur 12.

- ✓ Pour ce qui concerne le temps de compression, nous avons dans tout les cas le temps de compression de Huffman est petit par rapport à celui de la méthode VLC, et cette différence est beaucoup plus remarquable pour la valeur 20.

- ✓ Pour le temps de décompression nous pouvons dire qu'il est proportionnel à la taille du fichier à décompresser pour les deux méthodes.

- ✓ Les ressources mémoire utilisées pour l'exécution du programme de l'application de Huffman est grand par rapport à celui de VLC car la méthode de Huffman doit charger l'arbre de Huffman dans la mémoire pour travailler sur elle.

Nous interprétons les résultats obtenus par ces causes :

- ✓ La compression avec la méthode de Huffman n'est pas vraiment intéressante dès que la taille du mot à lire dépasse une certaine valeur à cause du volume que l'arbre de Huffman prend dans le fichier compressé et comme nous l'avons vu dans le chapitre de la méthode de Huffman le nombre de nœuds dans l'arbre est égal au nombre de caractères à coder multiplier par 2 moins 1.

- ✓ Dans le cas de la méthode VLC il suffit de sauvegarder la table triée qui est de taille égale au nombre de caractères à coder.

- ✓ Le temps de compression est grand pour la méthode VLC à cause du nombre de calculs qu'il faut faire pour le codage des données contrairement à la méthode Huffman où une fois l'arbre est construit, il est parcouru lors du codage.

- ✓ Nous pouvons aussi dire que la cause la grandeur du temps pour VLC est due au langage de programmation utilisé, c'est-à-dire le JAVA car lors de l'exécution de l'application le système fait appel à la machine virtuelle qui exécute le fichier qui contient le byte-code généré lors de la compilation.

VI. Conclusion :

Dans ce chapitre d'implémentation et évaluation nous avons présenté au premier lieu l'environnement de notre travail, c'est-à-dire les logiciels utilisés et le langage de programmation ainsi que les principales méthodes qui constituent le noyau des deux applications.

Dans la deuxième partie, nous avons donné les résultats obtenus après les tests effectués pour les deux méthodes, le taux et le temps par rapport aux types de fichiers et aussi à la longueur du mot lu avec les interprétations des schémas, suivi par la justification de ces résultats.

Conclusion générale

La compression des données est appelée à prendre un rôle encore plus important en raison du développement des réseaux et du multimédia. Son importance est surtout due au décalage qui existe entre les possibilités matérielles des dispositifs que nous utilisons : débits sur Internet, sur Numéris et sur les divers câbles, capacité des mémoires de masse, et les besoins qu'expriment les utilisateurs : visiophonie, vidéo plein écran, transfert de quantités d'information toujours plus importantes dans des délais toujours plus brefs. Quand ce décalage n'existe pas, ce qui est rare, la compression permet de toute façon des économies.

Notre travail avait comme objectif de faire une étude comparative entre deux méthodes de compression sans perte. Une qui existe déjà et qui a monté son efficacité c'est celle de Huffman, et une autre qui est proposée et qui se base sur l'encodage à longueur variable (VLC). Pour ce faire il est nécessaire de comprendre divers concepts dans le monde de la compression de données et de la programmation.

A la fin de notre travail, nous avons constaté que la méthode proposée donne de meilleurs résultats pour le taux par rapport à celle de Huffman lorsque la taille du mot à lire dépasse une certaine valeur tout en donnant des taux considérable pour les autres valeurs. Mais pour le temps de compression c'est la méthode de Huffman qui donne les meilleurs résultats.

Ce travail m'a donné l'occasion d'apprendre beaucoup de choses dans un domaine aussi vaste qui est la compression de données, les différents algorithmes utilisés ainsi de mieux se familiariser avec le langage JAVA et de l'outil de développement Eclipse. Le travail que nous avons fait peut être amélioré pour donné de bons résultats par exemple en le programmant avec un autre langage pour diminuer le temps de compression comme le langage C et aussi de le tester sur les autres types de fichier.

Nous sommes loin d'avoir épuisé toutes les pistes de recherche. Les méthodes du futur sauront sans doute s'adapter à la nature des données à compresser et utiliseront l'intelligence artificielle.

I. Références webliographiques :

[6] : http://datacompression.free.fr/

[8] : www.mpeg.org.

[10] : www.candiulb.be/forum/index.php

[19] : www.developpez.com

[23] : http://java.sun.com/

[24] : www.Siteduzero.com

[27] : www.nlp.stanford.edu/IR-book/html/htmledition/gamma-codes-1.html

II. Références bibliographiques :

[25] : Henri Nussbaumer, Téléinformatique III ; Session, Présentation, Compression des données, Couche application.

[26] : Jean –Guillaume Dumas, Jean-Louis Roch, Eric Tannier, Sébastien Varrette, Théorie des codes ; compression , cryptage, correction.

[27] : Mémoire d'ingénieure, Implémentation et évaluation d'une nouvelle méthode de compression de données basée sur le code VLC, promotion 2009/2010.

[1] : Article de Dumont Renaud, Support Compression et Stockage.

[2] : Article de MM. S.Maadi, Y. Peneveyre, et C. Lambercy, Compression de données sans pertes.

[3] : Thèse de Guillaume BOISSON Représentations hautement scalables pour la compression vidéo sur une large gamme de débits/résolutions, 2005

[4] : Article de PEREIRA Vincent - LEPRETTE Franck - HACAULT Vincent , Compression de données, Décembre 2004

[5] : Article de MM. S.Maadi, Y. Peneveyre, etC. Lambercy, Compression de données avec pertes.

[7] : Article Pierre Nerzic , Codage et compression d'images et de vidéo , 2007.

[9] : Article de Christian Quest,Compression de données, 2000.

[11] : Cours de Master II: IASIG Dr. Mvogo Ngono Joseph, Principes généraux de codage entropique d'une source Compression d'images.

[12] :Article de G.BINET, Codage de source,2007 -2008.

[13] : Article de Olivier RIOUL, Codage entropique à longueur variable, 2003.

[14]: Article de Michel Van Caneghem, La compression de données, Mars 2005.

[15]: Article de Peter Fenwick, Punctured Elias Codes for variable-length coding of the integers, December 5, 1996.

[16] : Gholam-Reza Mohammad-Khani, Chang-Ming Lee, Michel Kieffer et Pierre Duhamel, Treillis à complexité réduitepour le décodage de codes à longueur variable, 5 février 2007

[17] : Article CAVLC (Context based Adaptive Variable Length Coding)

[18] : Article Les arbres

[20] : Article de Alexandre THIL, Initiation à la Recherche Algorithmes de compression de données et de traitement d'images

[21] : Article Codage de Huffman ,2007

[22] : Extrait livre Dubois M_enissier Catherine Dubois, Conception descendante