REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE
LA RECHERCHE SCIENTIFIQUE

UNIVERSITE MOULOUD MAMMERI TIZI-OUZOU
FACULTE DE GENIE ELECTRIQUE ET INFORMATIQUE
DEPARTEMENT D'INFORMATIQUE

Mémoire

De fin d'étude

En vue de l'obtention du diplôme master II en informatique

Option: Système informatique

Thème

Compression des données avec la nouvelle méthode VLC adaptative temps réel

Proposé par:
M^{r} SADOU S.

Réalisé par:
M^{r} BELGUESSOUM Aissa

2012/2013

Remerciement

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

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 tien à lui reconnaitre 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 leur jugement mon travail.

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

Dédicaces

Je dédie ce modeste travail à :

Mes très chers parents ;

Mes très chers sœurs Fazia, Fahima, Lynda et Hassina ;

Mes très chers frère Mohamed, Moussa, Hamza ;

Mon bon frère Madjid et son fils Mourad ;

Ma belle sœur Kamilia et ça fille Anais

Ma tente zhor et ces deux enfants Sinbad et ça sœur Salema ;

Tous mes précieux amies et amis ;

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

Toute personne que j'estime.

Aissa

Sommaire

---------- Chapitre II : les méthodes VLC ----------

----------Chapitre III : la méthode adaptative----------

---------- Chapitre IV : Analyse et conception ----------

---------- Chapitre V : implémentation et évaluation ----------

Liste des figures :

Liste des tableaux :

1. Introduction

Les développements technologiques et les exigences des utilisateurs conduisent à des quantités de données très importantes. Depuis les débuts des technologies de l'information surtout les données informatique (les fichiers multimédias : image, vidéo, son) : la transmission des données sur un réseau constitue un problème, parce que la bande passante alloue pour se transfert est très limité (quelques de ko/s pour la vidéo à la demande, par exemple), c'est pour ça la compression des données est presque toujours utilisée.

De multiples études ont été menées sur les algorithmes de compression, dans le but de mettre les données sous un format tel qu'il occupe moins de volume et moins de bande passante lors de la transmission. Une fois les données sont compressées, elles ne sont pas directement accessibles il est nécessaire de les décompresser pour qu'elles redeviennent utilisables.

2. Définition [1]

La compression consiste à réduire la taille physique de bloc d'information. Chaque compresseur utilise un algorithme de compression qui sert à optimisé la taille des données, la méthode et l'algorithme utilisés pour la compression dépendent essentiellement du type de données à compresser : on ne compressera pas de la même façon une image qu'un texte ou une vidéo, après la compression on utilise un décompresseur pour rendre les données compressées utilisables grâce à des algorithmes inverses de celui utilisé pour la compression.

Les algorithmes de compression sont basés sur trois critères :

- **Le taux de compression :** c'est la taille des données compressées sur la taille initiale : $\textbf{Taux} = \frac{\textbf{taille compressé}}{\textbf{taille initiale}} * 100\ \%$
- **Qualité de compression :** sans ou avec perte.
- **La vitesse** de compression et de décompression.

3. Pourquoi la compression des données

De nos jours, la puissance des processeurs augmentent plus vite que la capacité de stockage, et énormément plus vite que la bande passante des réseaux (Internet). Il s'avère

donc nécessaire de réduire la taille de ces fichiers afin de les stocker ou bien encore afin de les transférés plus facilement. Ce qui veut dire que la compression a plusieurs causes :

Si la capacité des disques augmente d'une manière importante, le débit des réseaux augmente de manière beaucoup moins importante. La compression des données avant l'envoi permet de réduire la taille de la donnée et d'augmenter le débit. Il suffit au destinataire de les décompresser pour récupérer la donnée initiale (avant la compression).

Si les images et autres fichiers multimédias sans compression sont plus faciles à manipuler, ils prennent une place mémoire importante. Les ressources en espace de stockage qu'ils demandent sont trop importantes.

Alors la compression des données est une solution pour résoudre les problèmes de capacité de stockage et la vitesse de transmission des données.

4. Comment se fait la compression

Pour réduire la place qu'occupe un fichier sur la mémoire, nous utiliserons des logiciels de compression. Ces derniers utilisent des algorithmes, qui sont une suite d'opérations mathématiques, propre au type de données contenues dans le fichier afin de coder l'information initiale qui après la compression puis la décompression, sera retrouvée soit à l'identique soit légèrement modifiée.

5. La différence entre le codage et la compression

Le codage c'est l'envoi des données de l'émetteur vers le récepteur par le canal sous forme des signaux (des codes), mais la compression des données c'est de réduire la taille de la donnée qu'on veut envoyer à l'émetteur pour réduire le temps de transmission. Si on parle de l'ordre de ces deux opérations la première c'est l'opération de codage et la seconde c'est la compression, après ces deux opérations les données sont prêtes à la transmission.

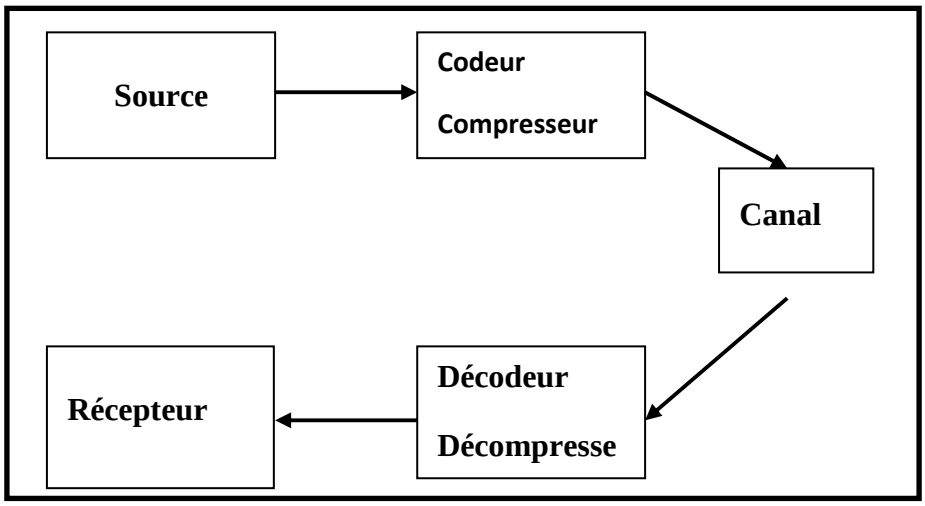


Figure I.1 : Codage et compression du texte.

6. Bref introduction au théorème de l'information

6.1. Quantité d'information [2]

Pour une source S tel que S=(S, P), la quantité d'information fournie par l'occurrence s tel que s appartiens à S est :

$$H(s) = Log2\left(\frac{1}{P(S)}\right) = -Log2(p(s))$$

Ou Log2 désigne logarithme binaire et P(S) est la probabilité d'apparition de symbole s tel que :

$$\mathbf{P(s)} = \frac{\textbf{Frequence (s)}}{\textbf{Taille en nombre de symboles}}$$

6.2. Entropie [2]

L'entropie d'une source est la quantité moyenne de l'information contenue dans la source, et l'entropie d'une source S=(S, P) est :

$$\mathbf{H} = \sum_{\mathbf{S}} \mathbf{P(S)H(S)} = -\sum_{\mathbf{S}} \mathbf{P(S)log_2(P(S))} \dots [\mathbf{1}]$$

L'entropie est maximale lorsque tous les symboles émis par la source sont équiprobables :

$$\mathbf{P(s)} = \frac{\mathbf{1}}{\mathbf{n}}$$

La relation [1] devient alors :

$$\mathbf{H=H_{max}=log_2(n)}$$

La quantité de décision D (décision content) est liée au nombre de symboles n parmi lesquels la source choisit ceux qu'elle va émettre :

$$\mathbf{D= log_2\ (n).}$$

6.3. La redondance [2]

La compression des données se fonde sur détection et élimination des données redondantes. Sachant que la donnée informatique (texte, image, vidéo ….etc.) est représentée sur une combinaison de 256 octets. Alors chaque flot de donnée supérieur présente une redondance.

La redondance moyenne mesure la quantité d'information non significative dans le débit de décision générée par une source, et est donc logiquement mesuré par :

$$\mathbf{R= D-H.}$$

7. La compression sans et avec perte [3]

7.1. La compression sans perte

La compression sans perte est principalement applicable aux données qui demandent une restitution exacte. Dans le contexte de la compression sans perte la méthode prend en entrée une série de bits X qu'elle transforme en une nouvelle série de bits Y plus courte que X. La série de bits Y est transmise au stockage ou pour usage ultérieur, lorsque l'on veut récupérer les données on prend Y et en appliquant la méthode inverse de la compression c'est la méthode de décompression pour récupérer la données X intact. La figure 2 illustre schématiquement le processus.

Dans la compression sans perte il existe trois types d'algorithmes :

- **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 présentées que d'autres dans les données que l'on veut compresser.

Dans un fichier HTML par exemple, on trouvera beaucoup de signes <, /, et >. On va chercher à coder les données qui se répètent souvent sur moins de bits, et les données moins fréquentes sur plus de bits. On va chercher à élargir la réduction des répétitions des groupes d'octets plutôt que des octets simples (principe utilisé pour les algorithmes de Lempel-Ziv, Zip, Huffman · · ·).

- **Substitution de séquences :**

Comprime les séquences de caractères identiques.

- **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 (la compression avec perte).

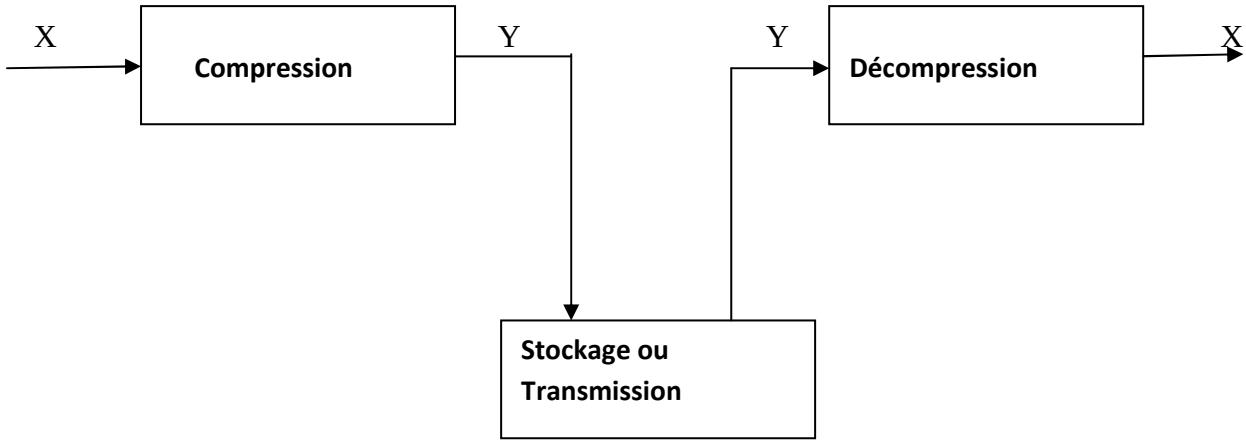


Figure I.2 : Une vision simplifie de la compression sans perte

7.2. La compression avec perte

Ce genre de méthodes s'applique aux fichiers de données audiovisuelles. Le principe est clair : nous acceptons une dégradation de donnée décompressée imperceptible aux sens humaine ou suffisamment faible pour être acceptable, en contrepartie d'un taux de compression beaucoup plus intéressant. La figure I.3 illustre le processus générale, on compresse une série de bits X pour obtenir une série de bits Y, qui est transmise ou stockée. Lorsque on décompresse Y on retrouve Z qu'est potentiellement différent de X. Si Z diffère X il faudra alors Z ~ X.

Il existe des algorithmes de compression consacrés à des usages particuliers :

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

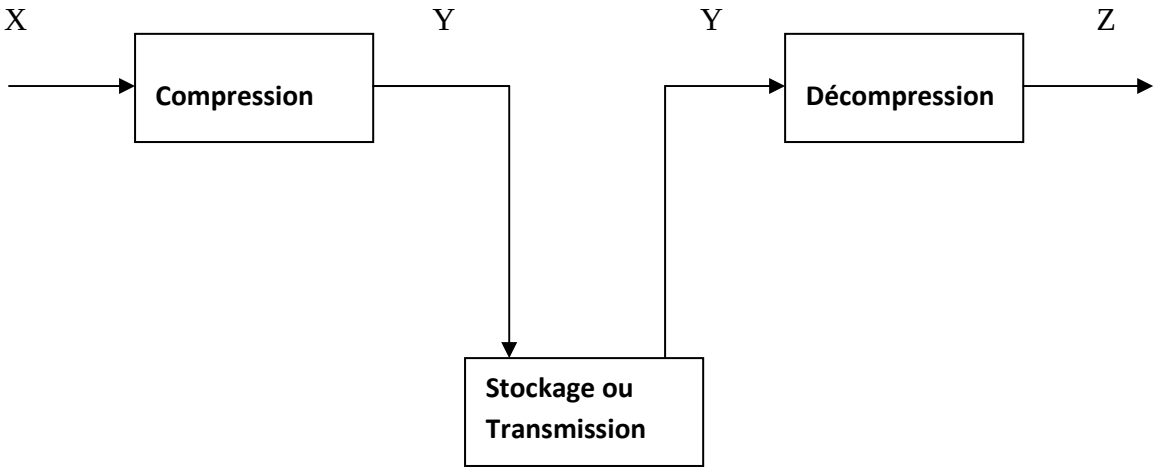


Figure I.3 : Une vision simplifie de la compression avec perte.

8. Compression symétrique / asymétrique [4]

8.1. 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 utilisé dans les transmissions de données. En général la compression est dite symétrique lorsque le codeur et le décodeur utilisent le même procédé et le temps d'exécution est égal pour les deux étapes, la figure I.4 illustre le processus.

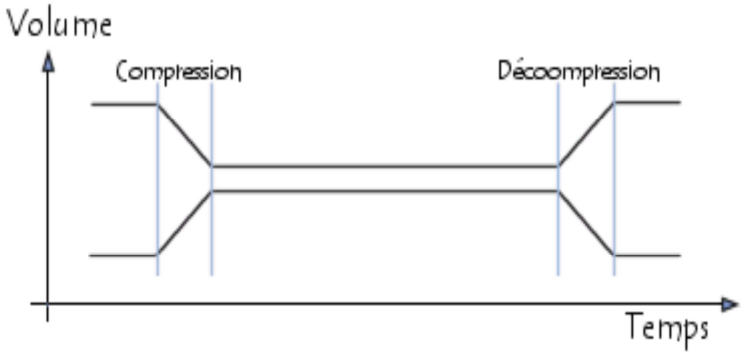


Figure I.4 : Compression de type symétrique.

8.2. La compression asymétrique

La compression asymétrique demande plus de travail pour l'une des deux opérations, la plupart des algorithmes demande 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 l'on archive des données auxquelles on accède peu souvent. Parfois une asymétrique est recherchée, lorsque le décodage doit être beaucoup plus rapide que la compression, par exemple pour les vidéos MPEG4.

9. La compression physique et la compression logique [4]

On considère généralement la compression comme un algorithme capable de comprimer énormément de données dans un minimum de place (***compression physique***), mais on peut 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 est faite sur la base de comment les données sont compressées ou plus précisément comment est-ce que les données sont réarrangées dans une forme plus compacte.

9.1. La 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.

9.2. La 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. La substitution logique ne fonctionne qu'au niveau du caractère ou plus haut et qui est basée exclusivement sur l'information contenue à l'intérieur même des données. Un autre exemple, moins heureux, de substitution logique est de remplacer 1999 par 99...

10. Compression statistique / numérique [4]

Dans la compression des données nous pouvons distinguer deux types d'algorithme : Les algorithmes qui travaillent au niveau statique et ceux qui travaillent au niveau numérique.

10.1. Compression statistique

Ce sont les méthodes basées sur le calcule de probabilité d'occurrence pour chaque caractère dans le fichier à compresser. La méthode concernée est les codes à longueurs variables.

10.1.1. Codes à longueurs variables (VLC : Variable length codes)

L'idée est la suivante : nous avons l'habitude de coder toutes les valeurs que nous avons à stocker avec le même nombre de bits, généralement un ou plusieurs octets. Nous utilisons un octet par caractère quand nous travaillons avec du texte, un ou deux octets par pixel et par couche quand nous travaillons avec des images. Est-ce-que c'est la meilleure solution ? La théorie nous apprend que non.

Exemple :

Prenons le texte « ANANAS ET BANANES », regardons la différence de quantité d'information entre le message symbolique et le message ASCII associe.

s	f(s)	p(s)	H(s)	H	ASCII
A	5	0,29411765	1,76553475	8,82767373	40
N	4	0,23529412	2,08746284	8,34985137	32
E	2	0,11764706	3,08746284	6,17492568	16
S	2	0,11764706	3,08746284	6,17492568	16
<espace>	2	0,11764706	3,08746284	6,17492568	16
B	1	0,05882353	4,08746284	4,08746284	8
T	1	0,05882353	4,08746284	4,08746284	8
TOTAL	17	1	21 ,29031179	43,8772278	136

Tableau I.1 : Caractéristique de la compression statistique [4].

s: symbole

f(s) : fréquences d'apparition du symbole

p(s) : probabilité d'apparition du symbole

H(s) : quantité d'information convoyée par le symbole

H : quantité d'information cumulée par ce symbole **(f(s) * H(s))**

ASCII : quantité de décision selon un code ASCII

Il est en effet plus efficace d'utiliser des codes plus courts pour des valeurs fréquentes et de réserver des codes plus longs pour les valeurs moins fréquentes. L'intérêt de VLC dépendra bien entendu de la distribution des valeurs à coder telle qu'on peut l'étudier en construisant l'histogramme des valeurs à coder.

Parmi les méthodes basées sur VLC on trouve la méthode de Huffman, la méthode de Shannon-Fano, …. Etc.

10.2. Compression numérique

Dans ce cas les valeurs des motifs influents sur la compression (par exemple JPEG), et les substitutions sont interdites.

11. Codage adaptatif, semi-adaptatif et non-adaptatif [3]

11.1. 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 comme LZW ou Huffmann dynamique déterminent la dépendance des données en construisant leur dictionnaire à la volée. Ils n'ont pas de listes prédéfinies de chaînes de caractères, 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.

11.2. Codage non adaptatif

Certains codeurs statiques comme le codage de Huffmann 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.

11.3. Codage semi-adaptatif

Un mélange des ces deux méthodes d'encodage à l'aide de dictionnaire et la méthode d'encodage semi-adaptative. Un codage 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.

12. Quelques algorithmes de compression sans perte

12.1. L'algorithme de Huffman []

Huffman en 1952 a proposé un algorithme de codage à longueur variable d'une source à L messages. Cet algorithme permet d'obtenir le nombre moyen de bits par mot minimum tout en garantissant un code uniquement décodable et instantané.

Le principe du codage de Huffman repose sur la création d'un arbre composé de nœud. Supposons que la phrase à coder est «*Recherche chat châtain* ». On recherche tout d'abord le nombre d'occurrences de chaque caractère.

- On repère la lettre la plus utilisée : le *c* apparait 4 fois, de même que *h*.
- Puis les lettres *a* et *e* apparaissent 3 fois.
- Ensuite les lettres *t*, *r* apparaissent 2 fois.
- Pour finir les lettres *i* et *n* apparaissent 1 fois.

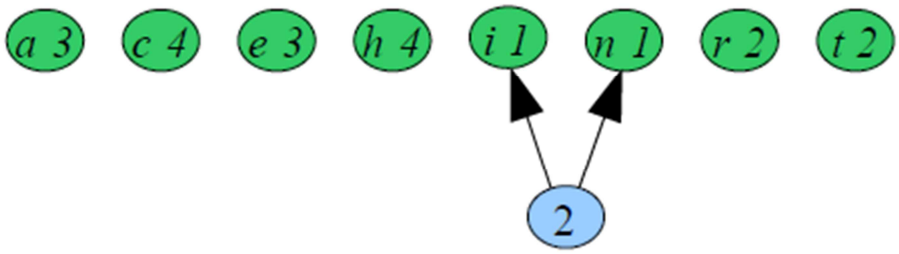
On obtient donc le tableau suivant :

symboles	a	c	e	h	i	n	r	t
poids	3	4	3	4	1	1	2	2

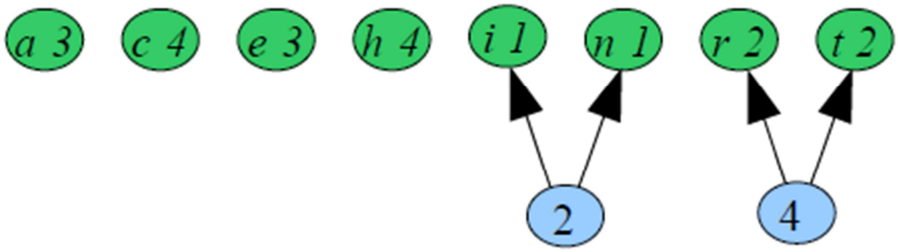
12.1.1 Construction de l'arbre

Ce tableau nous permet de construire les feuilles de l'arbre:

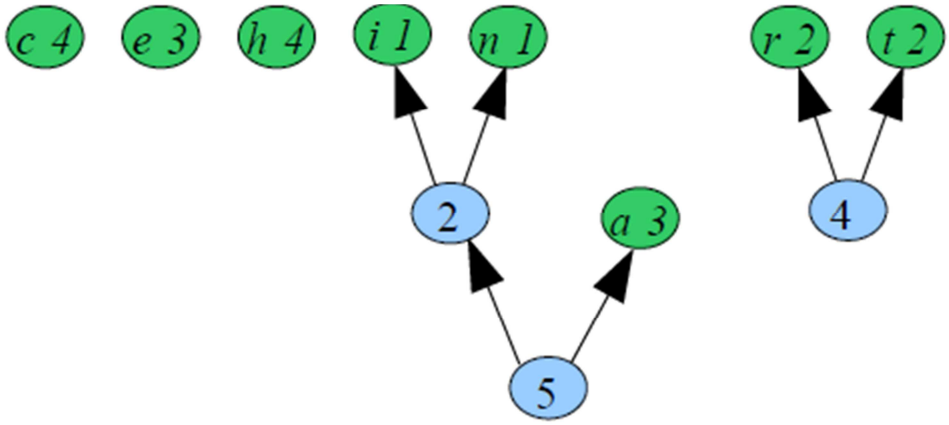
On détermine ensuite les deux lettres de poids le plus faible : ici les lettres *i* et *n* sont de poids On représente deux branches qui se rejoignent et qui relient les deux lettres les moins fréquemment utilisées:



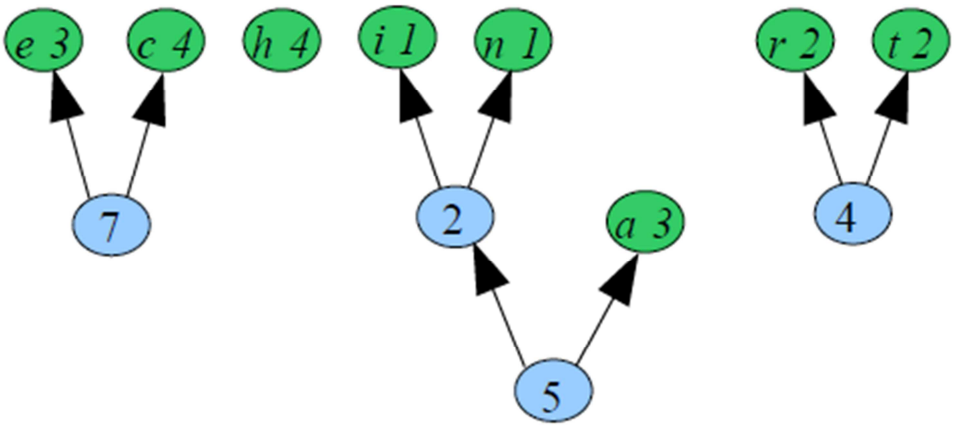
On répète le processus en constatant que les deux caractères de poids minimum sont *r* et *t* :



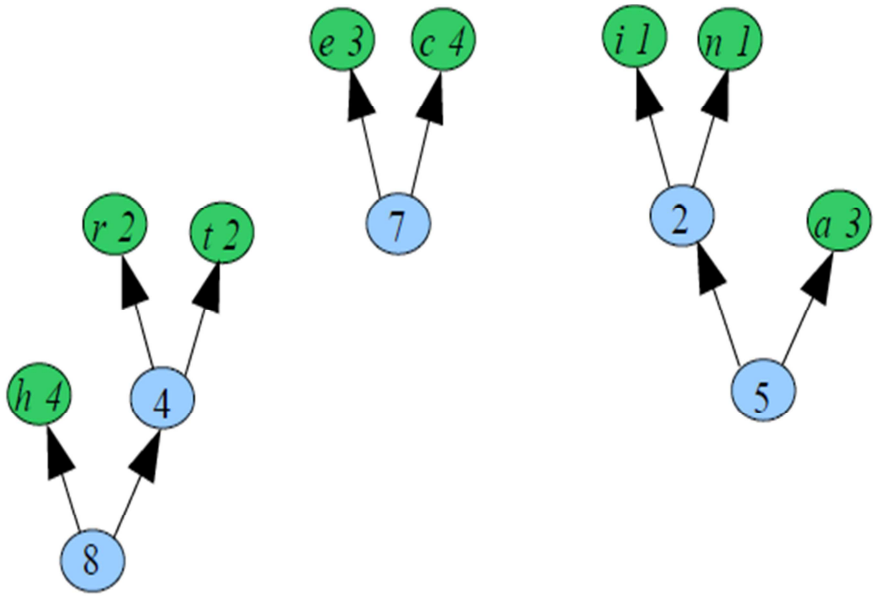
Les deux poids minimums sont ceux du nœud *a* et de poids 2 :



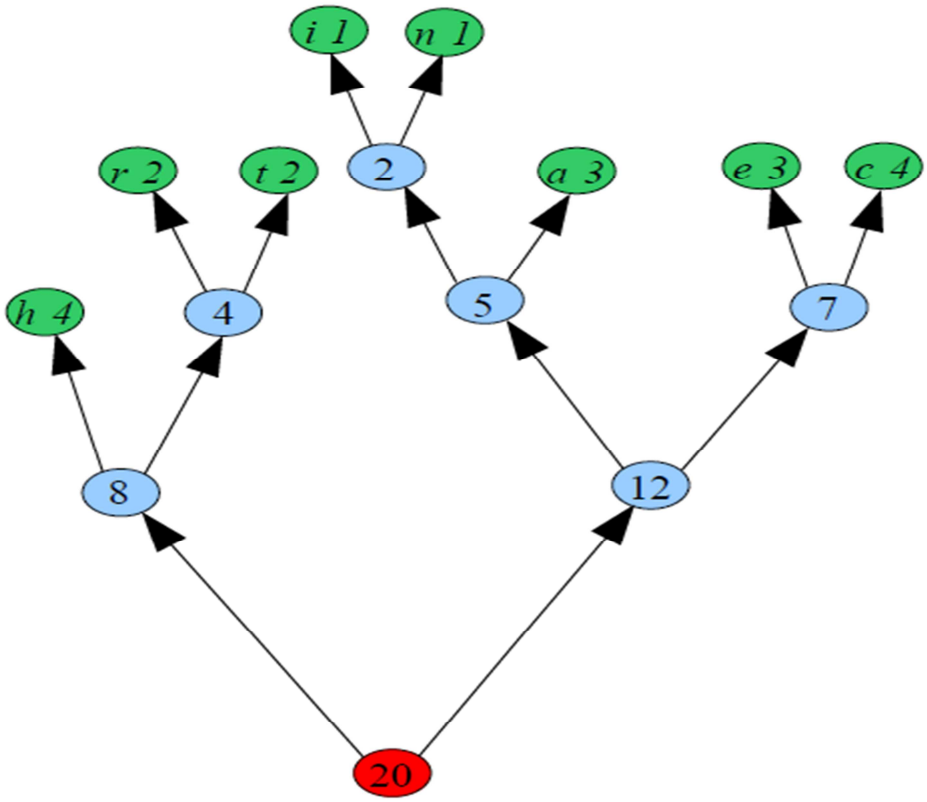
Les deux minimums sont ensuite les nœuds dont les caractères sont *e* et *c* :



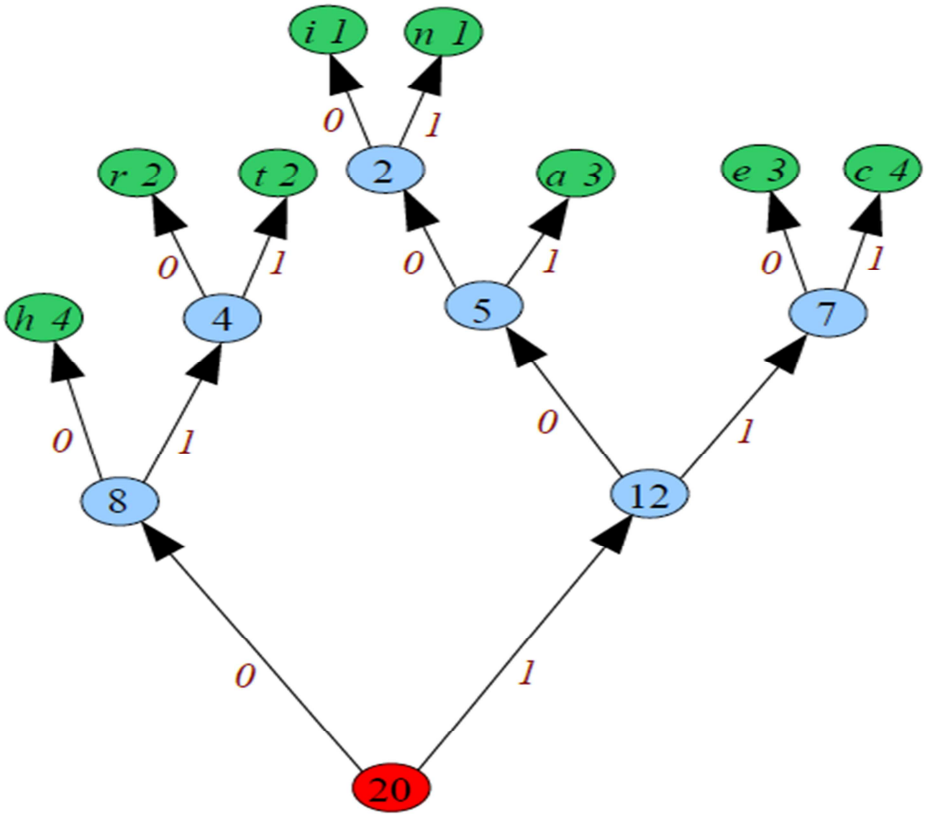
Les deux minimums sont alors les nœuds de poids 4 :



On relie ensuite les nœuds de poids 5 et 7 pour obtenir un nœud de poids 12. On arrive ainsi nœud final en reliant ces deux derniers nœuds :



Ensuite, par convention, on place un 0 a gauche de chaque branche et un 1 a droite de chaque branche



Ce qui va nous permettre de coder les lettres a l'aide de 0 et 1. Ainsi le codage de *r* donne *100*, celui de *e* donne *001* et ainsi de suite. Le codage de «*Recherche chat châtain* ». Donne :

0101101110011001011100110111001010111110010101110110001001

12.1.2 La décompression

Pour décompresser le code précédent il suffit de prendre le code binaire produit par la compression et de parcoure l'arbre de la racine vers ces feuille.

Si on reprend le code binaire produit par la compression de la chaine des caractères «*Recherche chat châtain* ». De l'exemple précédent :

Le premier bit est 0, on se dirige vers la gauche, le second est 1, on se dirige vers la droit, le troisième bit est 0, on se dirige vers la gauche dans ce cas on va trouver la feuille r. on fait la même chose pour le reste de code.

12.2. L'algorithme RLE (Run-Length Encoding) [3]

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 utiliser un algorithme de compression plus complexe ou laisser l'image sans compression.

12.2.1. Principe

Le codage RLE (run-length encoding), ou « codage par plages » consiste à réduire la taille d'un texte en remplaçant des suites de caractères identiques en un couple de valeurs (caractère remplacé, nombre d'occurrences). Par exemple « AAAAABBBBBB » pourrait être représenté par « 5*[A], 6*[B]».

Un paquet RLE (RLE packet) est codé sur 2 octet, le premier octet pour représenter le nombre de répétition de caractère dans le paquet le second octet pour représenter le caractère lui-même. Alors la chaine précédente est représenté par 4 octet, 2 octet pour le paquet (5*[A]) et 2 octet pour le paquet (6*[B]).

Si l'on suppose que notre chaîne contient maintenant 4 passages différents

AAAAAABBBXXXXXT

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

6A3B5X1T

Ainsi, après le codage, la chaîne de 15 octets de départ prendra seulement 8 octets de données pour représenter la chaîne.

De longs passages sont rares dans certains types de données 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, on a besoin de 2 caractères, par conséquent, il s'avère que si l'on code des chaînes de caractères en RLE constituées de passages composés uniquement de 2 caractères on n'aura 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.

Exemple

Cannibalisation

Après le codage RLE :

1C1a2n1i1b1a1l1i1s1a1t1i1o1n

On peut déduire de tout cela que le codage RLE est simple et efficace mais 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. (C'est l'une des raisons pour laquelle on exprime la complexité d'une image en fonction du nombre de couleurs différentes qu'elle contient).

12.3. L'algorithme LZW (Lempl - Ziv - Welch) [3]

C'est une variante populaire de LZ78, développée par Terry Welch en 1984. Sa caractéristique principale est d'éliminer le deuxième champ du signe. Un signe de LZW se compose juste d'un pointeur au dictionnaire. Pour mieux comprendre LZW, nous pensons au dictionnaire comme un tableau de chaînes de caractères de taille variables.
La méthode LZW commence par initialiser le dictionnaire à tous les symboles du code ASCII, les 256 premières entrées du dictionnaire (entrées 0 à 255) sont occupées avant que n'importe quelles données soient entrées. Puisque le dictionnaire est initialisé, le prochain caractère d'entrée sera toujours trouvé dans le dictionnaire. C'est pourquoi le signe de LZW peut se composer juste d'un pointeur et ne doit pas contenir un code et un caractère comme dans LZ77 et LZ78.

12.3.1. Compression:

Le principe de LZW est que l'encodeur lis des symboles un à un et les accumule dans une chaîne I. Après que chaque symbole soit lu et soit concaténé à I, le dictionnaire recherche la chaîne I. Aussi longtemps que I est trouvé dans le dictionnaire, le processus continue.
À un certain point, l'ajout du prochain symbole x fait échouer la recherche; la chaîne I est dans le dictionnaire mais la chaîne Ix (symbole x concaténé à I) n'est pas. À ce moment l'encodeur (1) sort le pointeur de dictionnaire qui pointe sur la chaîne I dans le flux de sortie, (2) sauvegarde la chaîne Ix (qui s'appelle maintenant une phrase) dans la prochaine entrée disponible du dictionnaire, et (3) initialise la chaîne I au symbole x.
Pour illustrer ce processus, nous employons encore la chaîne: **"recherché chat "**. Les étapes sont comme suit:

1). Initialiser les entrées 0-255 du dictionnaire à chacun des 256 octets.

2). Le premier symbole "r" est lu et est trouvé dans le dictionnaire (dans entrée 114, puisque c'est le code d'ASCII de "r "). Le prochain symbole "e" est lu, mais "re" n'est pas trouvé dans le dictionnaire. L'encodeur exécute ce qui suit: (1) sortir 114, (2) sauvegardé la chaîne "re" dans la prochaine entrée de dictionnaire.

3). Le deuxième symbole "e" est lu et est trouvé dans le dictionnaire (dans entrée 101, le code d'ASCII de e). Le prochain symbole "c" est lu, mais "ec" n'est pas trouvé dans le dictionnaire. L'encodeur exécute ce qui suit: (1) sortir 101, (2) sauvegardé la chaîne "ec" dans la prochaine entrée de dictionnaire.

4) Le troisième symbole "c" est lu et est trouvé dans le dictionnaire (dans entrée 99, le code d'ASCII de c). Le prochain symbole "h" est lu, mais "ch" n'est pas trouvé dans le dictionnaire. L'encodeur exécute ce qui suit: (1) sortir 99, (2) sauvegardé la chaîne "ch" dans la prochaine entrée de dictionnaire. La même chose pour le reste de la chaine.

Le tableau suivant résume les différentes étapes de la compression de la chaine **"recherché chat"**

Etapes	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
S	r	e	c	h	e	r	c	ch	e	sp	c	ch	a	t
T	e	c	h	e	r	c	h	e	sp	c	h	a	t	
S+T	re	ec	ch	he	er	rc	ch	che	esp	spc	ch	cha	at	
Ued	non	non	non	non	non	non	oui	non	non	non	oui	non	non	
Ajouté Au déction	re	ec	ch	he	er	rc		che	epc	spc		cha	at	
ASCII	114	101	99	104	101	114		256	101	32		257	97	116

Tableaux I.2 : La compression "recherché chat" avec l'algorithme LZW [3].

Pour décompresser le code ASCII il suffit de suivre le dictionnaire produit par la compression.

Code ASCII	114	101	99	104	101	114	256	101	32	257	97	116
symbole	r	e	c	h	e	r	ch	e		ch	a	t

12.4. La compression arithmétique [3]

La technique d'encodage arithmétique due à P. Elsa n'a pas contrairement à celle de Huffman ou celle de Shannon, pour restriction de ne devoir traduire les probabilités d'occurrence 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, se traduisant par un gain de place de l'ordre de 4% ou 5% au plus, sauf dans des cas très particuliers.

Pour comprendre le principe de cette méthode prenons le message "BILL GATES" comme exemple. Sa distribution de probabilités a l'allure suivante :

SPACE	0.1
A	0.1
B	0.1
E	0.1
G	0.1
I	0.1
L	0.2
S	0.1
T	0.1

Figure I.5 : Probabilités d'apparition des symboles dans le message "BILL GATES"

On va maintenant associer à chaque symbole un domaine r à l'intérieur de l'espace des probabilités compris entre 0 et 1. L'association de chaque symbole avec un domaine particulier n'est pas critique, mais il est vital que le décodeur et l'encodeur utilisent la même table d'associations.

SPACE	0.1	0.00 <= r < 0.10
A	0.1	0.10 <= r < 0.20
B	0.1	0.20 <= r < 0.30
E	0.1	0.30 <= r < 0.40
G	0.1	0.40 <= r < 0.50
I	0.1	0.50 <= r < 0.60
L	0.2	0.60 <= r < 0.80
S	0.1	0.80 <= r < 0.90
T	0.1	0.90 <= r < 1.00

Figure I.6 : Association de domaines de probabilités aux symboles

Le premier caractère, 'B', se voit assigner un domaine entre 0.20 et 0.30. Le message final aura donc une valeur comprise entre 0.2 et 0.3, ce qui devient notre nouvel espace de probabilités pour le message. Le caractère I, qui obtient le domaine de 0.5 à 0.6, va utiliser le

domaine compris entre 50% et 60% du nouvel espace de probabilités, ce qui amène le message à un nouveau domaine compris entre 0.25 et 0.26.

Symbole	bas	haut
	0.0	1.00
B	0.2	0.3
I	0.25	0.26
L	0.256	0.258
L	0.2572	0.2576
<Espace>	0.25720	0.25724
G	0.257216	0.257220
A	0.2572164	0.2572168
T	0.25721676	0.2572168
E	0.257216772	0.257216776
S	0.2572167752	0.2572167756

Figure I.7 : Evolution du codage arithmétique de "BILL GATES"

0.2572167752 est la représentation arithmétique du message "BILL GATES". En effet, n'importe quelle quantité telle que $0.2572167752 <= x < 0.2572167756$ peut en principe représenter le message désiré. Le décodage est relativement simple : sachant que le message encodé se trouve entre les bornes 0.2 et 0.3, on voit immédiatement que le premier caractère est B. On soustrait ensuite 0.2 (la borne inférieure) au message, ce qui donne 0.0572167752, ce que l'on divise par l'intervalle de probabilité du symbole B, soit 0.1 : le résultat est 0.572167752, ce qui correspond à l'intervalle du symbole I, et ainsi de suite.

On utilise souvent un symbole spécial (EOF) pour encoder la fin du message, de manière à savoir quand le processus doit s'interrompre.

12.5. L'algorithme de Shannon- Fano [3]

Les principaux pionniers de la compression de données furent sans doute C.Shannon et R.M.Fano. Ils démontrèrent que l'on pouvait coder un message en utilisant la probabilité d'apparition d'un symbole ainsi que la base 2. Ils ont établi 3 propriétés qui permettent de compresser un message:

• Chaque code (ou presque) a un nombre de bits différent.

• Le code correspondant à un symbole dont la probabilité d'apparition dans le message est faible sera codé avec plus de bits.

• Même si les codes ont des longueurs différentes, cela ne doit pas empêcher de pouvoir les décoder de façon distincte.

L'algorithme de Shannon-Fano se décompose en différentes étapes:

a. Dresser une table triée par ordre croissant des fréquences d'apparition des symboles.

b. Diviser cette table en deux parties. La somme des fréquences de la première partie doit être le plus égale possible à la somme des fréquences de la deuxième partie.

c. Affecter le chiffre binaire 0 à la moitié supérieure, la moitié inférieure prenant le chiffre binaire 1.

d. Répéter les opérations 2 et 3 aux deux parties, jusqu'à ce que chaque symbole ne représente plus qu'une partie de la table.

Voyons à présent un exemple de l'algorithme de Shannon-Fano:

Exemple: prenons la chaîne de 39 caractères suivante:

"BCAACADBDCADAEEEABACDBACADCBADABEABEAAA".

1ère étape: Dresser une table triée par ordre croissant des fréquences d'apparition des symboles:

symboles	A	B	C	D	E
fréquences	15	7	6	6	5

2ème et 3ème étapes: Diviser cette table en deux parties. La somme des fréquences de la première partie doit être le plus égal possible à la somme des fréquences de la deuxième partie. Affecter le chiffre binaire 0 à la moitié supérieure, la moitié inférieure prend le chiffre binaire 1.

4ème étape: Répéter les étapes 2 et 3 pour obtenir l'arbre ci-dessous:

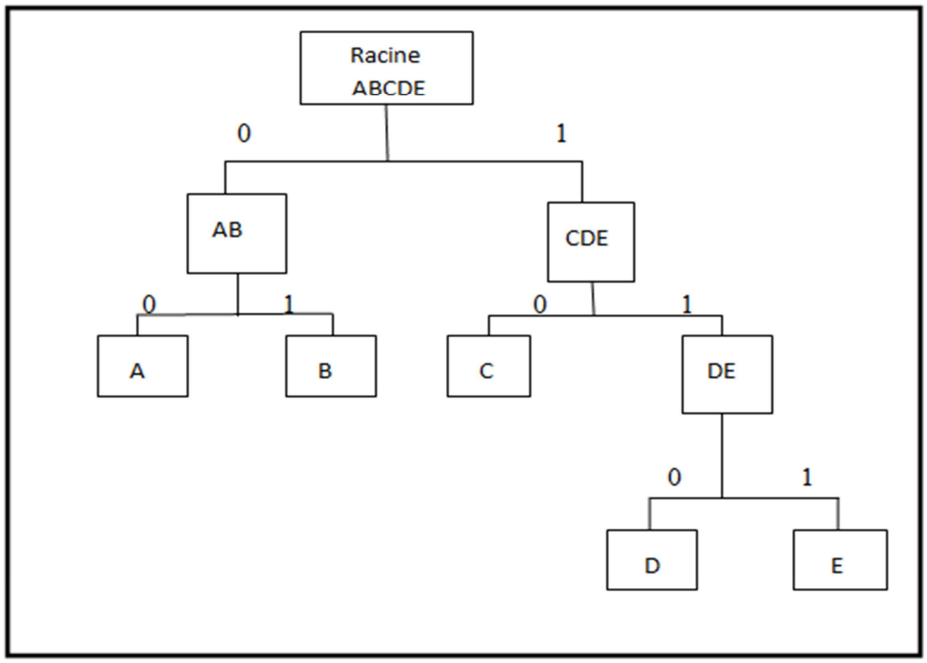


Figure I.8 : Arbre simple de Shannon- Fanon [3].

Pour représenter notre chaine sous forme binaire il suffit de parcourir l'arbre produit par la compression de haut en bas.

symbole	Code binaire
A	00
B	01
C	10
D	110
E	111

13. 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 et la décompression des données, ensuite nous avons vu les types de compression (sans perte et avec perte) et quelques algorithmes sur la compression sans perte.

Dans le chapitre qui suit on va décrire la méthode la plus récente et la plus efficace de compression. Cette méthode s'appelle la méthode VLC.

Introduction générale

Dans ce projet, nous avons fait une étude sur le développement de la nouvelle méthode de compression (la méthode VLC), on restant toujours adaptatif en demande de développement actuel.

Pour mener à bien notre étude, nous avons décomposé le travail en cinq chapitres.

- **Chapitre I :** dans ce chapitre nous avons présenté des généralités sur la compression des données ainsi le concept de base de la compression et quelques définitions sur la compression avec t sans perte.

- **Chapitre II :** ce chapitre est consacré pour la méthode VLC ainsi quelques algorithmes et technique qui utilisent se type de codage.

- **Chapitre III :** dans ce chapitre nous avons donné quelques algorithmes sur les méthodes de **Huffman** adaptatives et celle de **Shannon-Fanon,** pour comprendre le principe de méthode VLC adaptative.

- **Chapitre IV :** ce chapitre est consacré pour l'analyse et la conception de notre application, nous avons délivré comment notre application fonctionne.

- **Chapitre V :** ce chapitre présente l'environnement de développent et les logiciels utilisés pour notre application et nous avons fait quelques tests sur des fichiers (texte et image).

Enfin nous avons terminé par la conclusion.

1. Introduction

Nous distinguons deux types de systèmes de codage, les systèmes de codage à longueur fixe et les systèmes à longueur variable. Dans le codage à longueur fixe tous les mots ont la même longueur par exemple le codage ASCII, dans ce cas l'alphabet est un ensemble de 128 caractères.

Mais dans le codage à longueur variable les mots n'ont pas nécessairement la même longueur en bit, les codes à longueur variable sont apparus dés les premiers travaux de Shannon sur la théorie des communications.

2. Définition [5]

Un code à longueur variable (*VLC : Variable-Length Code*) est un code tel que les différents mots de code n'ont pas nécessairement la même longueur, en bits. On note $\boldsymbol{l_i}$ la longueur en bits du mot de code ***ci***. La distribution des longueurs du code est donc (l_1, l_2,.., l_m).

Les codes à longueur variable sont souvent choisi pour la compression des données, parce qu'avant toute opération d'enregistrement des données on assigne des codes plus court pour les symboles plus fréquents, et des codes plus long pour les symboles qui sont rarement utiliser.

Par exemple, considérons le code à longueur variable suivant :

(0, 10, 110,101, 111) avec des longueurs de code $(1, 2, 3, 3, 3)$ pour l'alphabet (A, B, C, D, E), et la chaine source BBBAAAAAAAAADDC avec des fréquences pour chaque symbole $(9, 3, 1, 2, 0)$. La moyenne en nombre de bit est :

$$l = \frac{1x9+2x3+3x1+3x2+3x0}{15} = 1{,}6 \text{ bits / symbole.}$$

Pour appliquer ce genre de codage (code VLC) sur n'importe type de données (dans notre cas on va utiliser les fichiers texte pour comprendre), il faut connaitre deux paramètres principaux :

- Connaitre la fréquence de chaque symbole, ce paramètre permet d'assigner les codes plus long (une suite de 1 et de 0) pour les symboles les moins fréquent, et les plus petits codes pour les symboles les plus fréquents.

- Une fois le symbole est remplacé par son code de longueur variable, le résultat est un autre fichier (compressé) qui contient la suite des codes correspondant à la séquence des symboles. le processus de décompression doit être capable de lire ce fichier et d'extraire sans ambigüité les différents codes. Donc l'unicité du code est indispensable.

3. Exemple d'application de code VLC (Variable-Length Code) [6]

➢ Le domaine de fabrication de microprocesseurs. Le principe est de chercher une instruction en mémoire et de l'exécuter, ces instructions sont généralement codé avec des tailles fixes, cela facilite leur manipulation par le microprocesseur. Mais les processeurs modernes optent pour une taille d'instructions variables. Il est alors possible de réduire la taille des programmes en utilisant des petits codes pour les instructions les plus fréquentes dans les programmes (exemple : l'instruction de chargement : load). Cela réduit aussi la taille physique du microprocesseur et la consommation d'énergie.

➢ Le standard ISBN (International Standard Book Number), est un nombre unique qui est attribué aux livres. Une partie de ce code désigne le code de pays de publication, la longueur de ce code varie entre 1 et 5 digits, et il assigne des petits codes pour les pays qui publient plus.

➢ Le code d'appel téléphonique international. La recommandation standard internationale attribue un code de longueur variable pour les pays selon la quantité d'appareil téléphonique que ce pays possède.

4. Codes uniquement décodables [7]

Pour coder une chaine de symboles avec la méthode VLC (long lenght code) aucun algorithme ou méthode est exigé. Il suffit de lit les symboles originaux $\mathbf{a_i}$ un par un et remplace chacun $\mathbf{a_i}$ avec une longueur binaire $\mathbf{c_i}$, les codes $\mathbf{c_i}$ sont enchainés pour former une

suite de bits. L'encodeur inclut une table avec toutes les paire ($\mathbf{a_i}$, $\mathbf{c_i}$) ou il exécute une procédure pour calculer $\mathbf{c_i}$ du code des morceaux $\mathbf{a_i}$ du symbole.

Décoder est plus complexe que coder en VLC, à cause des longueurs différentes des codes $\mathbf{c_i}$. Le décodeur lit la chaine de bits bit par bit par ce qu'il ne connait pas la taille de code de chaque symbole comme le code ASCII.

Nous commençons avec un simple exemple, soit les quatre symboles codés de la manière suivante : a1=0, a2=01, a3=011, a4=111. Soit la chaine de bits '0101110110' qui représente le codage de plusieurs chaines. par exemple '0101110110' peut être décodé au minimum de manière, '01 0 111 011 0'ce code donne la chaine a2a1a4a3a1 ou '0 10 111 011 0' ce code donne la chaine a1a2a4a3a1.

Alors on dit qu'un code est uniquement décodable s'il y a une seul manière possible pour décoder tous message encodé par ce code. On déduit de cette définition que l'exemple précèdent n'est pas uniquement décodable.

On peut toujours introduire le symbole d'extra ponctuation durant l'étape de codage par exemple le symbole '/'. On reprend l'exemple précédent avec insertion le symbole '/' après chaque code '01/0/111/011/0' ce code signifie uniquement la chaine a2a1a4a3a1. Mais cette méthode est inefficace par ce que le symbole '/' a lieu être insérer après tous symbole codé.

Le code idéal dans cette situation est le code qui n'est pas uniquement un code à longueur variable, mais si on introduit la notion de code préfixe alors la méthode VLC est efficace, par exemple le code a longueur variable (0, 10, 110, 111) a la propriété de code préfixe, et on remarque que les longueurs des mots de code sont les même que le code (0, 01, 011, 111).

5. Code préfixe et arbre binaire [7]

L'une des propriétés d'auto ponctuation est le code préfixe, un préfixe est les premiers bits d'un mot de code, quand des mots de code sont de longueur différente c'est possible que les premiers bits de plus long code est les bits de code plus petit. Par exemple soit deux codes c1=01 et c2=011 on remarque que c2 est le c1 on ajoutant un bit 1 alors le code c1 est le préfixe de c2

On dit qu'un code est un code préfixe si aucun code de mots n'est pas un préfixe d'un autre mot de ce code. Par exemple si on considère les mots des deux codes (0, 10, 010, 101) et (0, 10, 110, 111).

Le premier mot de code (0, 10, 010, 101) n'est pas un code préfixe par ce que le symbole codé de manière 10 est le préfixe de symbole codé de manière 101. Par contre le mot de code (0, 10, 110, 111) est un code préfixe par ce que aucun symbole est le préfixe d'un autre symbole.

La propriété du préfixe est considérée comme une caractéristique favorite quand on cherche un code uniquement décodable. Un code avec une telle propriété est appelé code préfixe.

5.1. Comment construire l'arbre binaire

Etant donné un ensemble de symboles constituants une source de données à compresser, et chaque symbole est caractérisé par sa probabilité d'apparition dans la source. La façon avec laquelle on construit le code VCL qui assigne le plus petit code au plus fréquent symbole consiste à construire un arbre binaire avec des branches pondérées et des feuilles qui représentent les différents symboles. Pour construire l'arbre binaire d'un n'importe code nous devons suivis les étapes suivantes :

Premièrement, nous créons un nœud comme étant la racine en suite nous regardons les mots de ce code un par un pour chaque mot de code nous lisons un bit à la fois du premier jusque au dernier. Nous commençons par la racine qui nous donne deux branches l'une à droit et l'autre à gauche. Si la fin d'une branche est une feuille alors nous insérons un bit à 0, si c'est un autre nœud nous insérons un bit à 1.

Le processus se répète d'un nœud à un autre tant que nous lisons un bit par bit jusqu'à la fin de mot de code.

Exemple : soit le code C (s_i)= (0, 10, 110,1111) pour l'alphabet (A, B, C, D), on construit son arbre binaire:

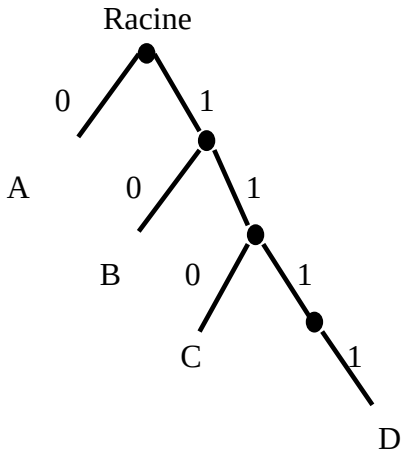


Figure II.1 : Exemple d'un arbre binaire.

Un arbre binaire ne produire pas toujours un code préfixe, pour distinguer que un code est un code préfixe à partir de son arbre binaire on vérifier les positions des symboles de ce code.

Si la position de tous les symboles dans l'arbre binaire est à la position des feuille on dira que se code est un code préfixe, mais si l'un des symboles est à la position d'un nœud on dira que se code n'est pas un code préfixe.

Exemple : soit les deux codes C_1 (s_i)= (1, 01, 001, 0100) et C_2 (s_i)= (0, 10, 110, 1111) pour l'alphabet (A, B, C, D). Nous remarquons que le code C_1 n'est pas un code préfixe car le deuxième mot 10 est un préfixe de dernier mot 0100.

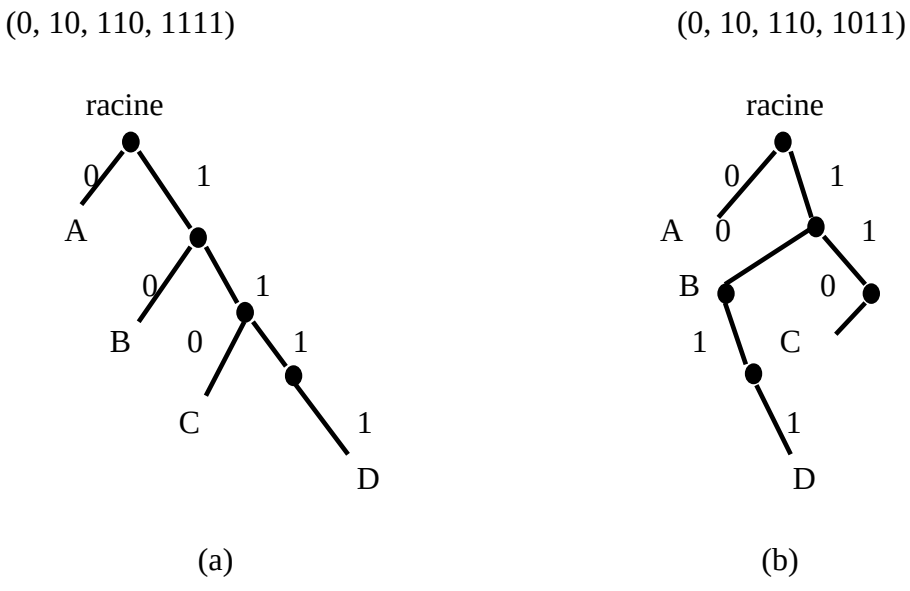


Figure II.2 : La propriété du préfixe

Nous remarquons que la position des mots de ce code sur l'arbre binaire (a) sont à la position des feuille alors le code C_1 est un code préfixe, par contre nous remarquons sur l'arbre binaire (b) que le mot B est à la position d'un nœud alors le code C_2 n'est pas un code préfixe.

6. Code préfixe et code uniquement décodable [8]

Tous les code qui sont des codes préfixes sont uniquement décodables, mais on a un code qui n'est pas un code préfixe, se code peut être uniquement décodable ou non. Dans ce ca on dit qu'un code préfixe est un sous ensemble des codes uniquement décodable.

Exemple 1 : on considère le code C (s_i) = (1, 10, 100, 1000) pour l'alphabet (A, B, C, D), on remarque que ce code n'est pas un code préfixe car le mot 1 est un préfixe pour d''autre mots 10, 100 et 1000.

Si on considéré le message 10100101000, n'as pas d'ambiguïté et il y'a un seul chemin pour le décoder : 10100101000, i.e. BCBD. Pour que le décodeur décode ce message il faut suivre les étapes suivantes :

Dans la première lecture le décodeur lit le bit 1 le décodeur vas considérer ce bit comme un A et le début de mot B, C et D mais dans la deuxième lecture le décodeur lit le bit 0 ici aucun mot commence avec un bit à 0 alors le décodeur vas prendre le mot 10 et il va décoder comme un B. ce processus répète jusqu'à ce que le décodeur trouve la chaine BCBD, la figure suivante résume le processus de décodage.

Exemple : on considère le code (1,10, 100) pour l'alphabet (A, B, C). Ce n'est pas un code préfixe car le premier mot de code est un préfixe de deuxième et troisième mot de ce code.

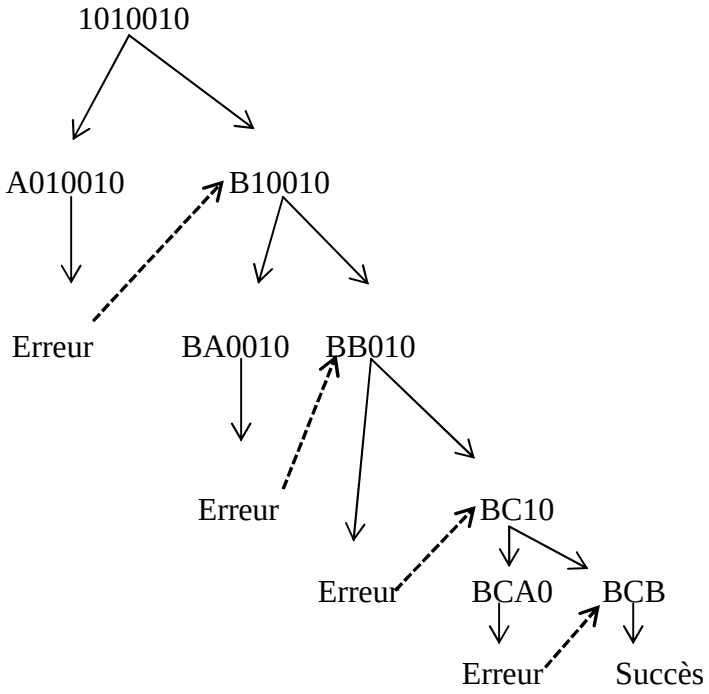


Figure II.3 : Un code qui n'est pas un code préfixe mai uniquement décodable

7. Inégalité de Kraft [8]

Les codes préfixes ont une autre propriété. Pour n'importe code qui n'est pas un code préfixe, avec longueur des mots qui satisfait certaines conditions, on peut toujours trouver un code préfixe avec les même longueurs des mots.

Exemple : si on considère le code unique décodable (1, 10, 00), ce n'est pas un code préfixe et les longueurs de ces mots est 1, 2, 2. On peut toujours trouver des codes préfixes avec les même longueurs par exemple (1, 01, 00).

La propriété du préfixe d'un code, nous garantit un code uniquement décodable, qui est efficace et correct. Pour faire une bonne compression les longueurs des mots de code sont nécessaires d'être plus court possible.

Exemple : on considère le code préfixe (0, 10, 110, 1111). Les longueurs des mots dans ce code sont 1, 2, 3, 4. Mais le dernier mot peut réduit de 4 à 3 comme (0, 10, 110, 111) qu'est aussi un code préfixe avec des longueurs suivantes : 1, 2, 3, 3. La figure 2.4 (a) et (b) montre les arbres binaire qui sont associes au code (0, 10, 110, 1111) et (0, 10, 110, 111)

respectivement. Comme on peut voir, si $\mathbf{l_j}$ est la longueur de $\mathbf{j^{iéme}}$ mot, avec **j= 1….. 4** alors le niveau de la feuille de ce mot est $\mathbf{l_j + 1}$.

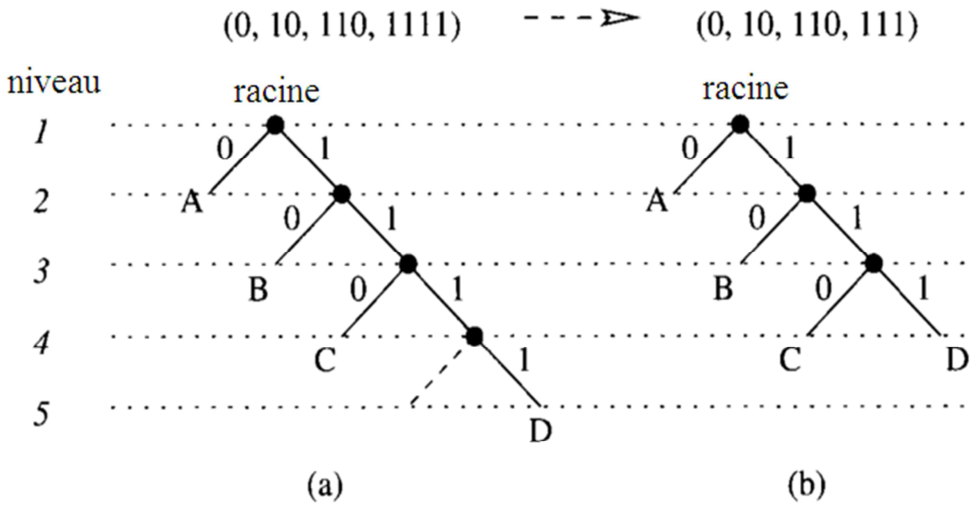


Figure II.4 : Deux codes préfixes.

Est-ce que on peut réduire encore la taille de dernier mot par un bit ? Par exemple, est ce que c'est possible de trouver un code préfixe avec les longueurs des mots 1, 2, 3, 2 pour les symboles A, B, C et D respectivement ?

Exemple : discutons s'il est possible de trouver un code préfixe avec les longueurs de ces mots 1, 2, 3, 2.

Solution : On construit un arbre binaire selon les longueurs données. Puisque il n'y a pas de problème quand la longueur est 1, le mot de code peut être 0 ou 1, nous commençons toujours par construire une branche pour 0. Maintenant nous avons un mot avec une longueur 1 (0 étiqueté comme A). En suite, on construit une branche 0 et une branche 1, et ça donne des mots de longueurs 2 (10 étiqueté comme B). En suite, on ajoute une branche 0 pour avoir le mot de longueur 3 (110 étiqueté comme C). On note dans cette approche que construire un mot de code de longueur *l* veut dire trouver un chemin de longueur *l* de la racine, ou trouver un nœud au niveau *l+1* (*l*=2 dans cette exemple, voir figure 2.5). Il y'a trois chemin valide de longueur 2 : 00, 01 et 11. Malheureusement ils sont tous pris sauf le chemin 11 qui correspond à un nœud interne.

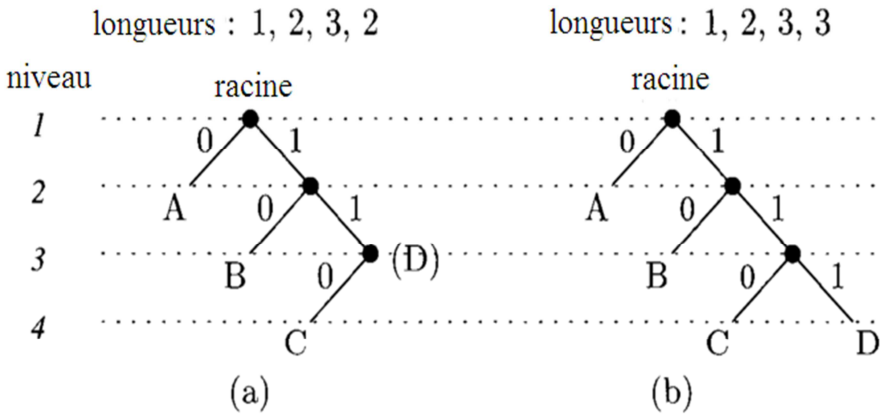


Figure II.5 : Longueurs impossibles (a) et longueurs possibles (b) pour un code préfixe

On suppose qu'on construit le mot 11 (étiqueté comme D) et on dérive éventuellement un arbre binaire pour le code avec les mots A, B, C, D qui sont de longueurs 1, 2, 3 et 2 respectivement, comme dans la figure II.5(a).

Comme on peut voir sur la figure II.5 (a), pas tous les mots qui sont sur des feuilles. Par exemple le symbole D n'est pas sur une feuille. Ça viole le facteur que pour un code préfixe, les mots doivent tous être sur les feuilles d'un arbre binaire. Et on conclu donc qu'il n'est pas possible de trouver un code préfixe avec les longueurs des mots 1, 2, 3, 2 pour les symboles A, B, C et D respectivement.

Pour la comparaison on construit l'arbre binaire pour le code avec longueurs de ces mots 1, 2, 3, 3 comme dans la figure II.5 (b), où tous les mots sont sur des feuilles.

Le théorème de kraft fourni un guide efficace pour le minimum nécessaire pour les longueurs des mots de codes préfixe. On connaissant cette limite, on peut faciliter de déduire la non validité d'un code préfixe qui est dans certains conditions de longueurs de ces mots.

Théorème : Un code préfixe $C = (c_1, c_2, \ldots, c_n)$ avec les longueurs des mots $l_1, l_2, \ldots, l_n$ respectivement, existe si et seulement si :

$$K(C) = \sum_{J=1}^{n} 2^{-lj} \leq 1$$

Cette inégalité est connue sous le nom « inégalité de Kraft. »

Avec l'inégalité de Kraft, on peut répondre facilement à la question de l'exemple précédant. Si les longueurs ne satisfirent pas l'inégalité de Kraft, on peut conclure que ce n'est pas possible de trouver un code préfixe avec ces longueurs.

On a :

$$\boldsymbol{K(C) = \sum_{J=1}^{n} 2^{-lj} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^2} > 1}$$

Il est impossible de trouver un code préfixe avec des longueurs 1, 2, 3, 2.

Cependant pour les longueurs 1, 2, 3, 3, il est possible de trouver un code préfixe avec ces longueurs car :

$$\boldsymbol{K(C) = \sum_{J=1}^{n} 2^{-lj} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^3} = 1}$$

8. Algorithmes et techniques de construction de code préfixe

8.1. Le code unaire [9]

Le code à longueur variable le plus simple et le plus intuitif à lequel on peut penser est le code unaire. Ce code satisfit la propriété de préfixe.

Pour code une chaine de symbole on utilise le code unaire il faut suivis les étapes suivantes :

- Trier symboles dans l'ordre croissante de grand fréquence au plus mois fréquence.
- Affecter chaque symbole à un nombre positif, le nombre 0 pour le symbole la grande fréquence le nombre 1 pour le suivant symbole qu'a la fréquence mois que le premier symbole…etc.
- Coder les nombre positif, une suite de N bits à '1' suivis d'un bit '0' à la fin de code.

Le tableau suivant illustre le code unaire :

Nombre N	symbole	fréquence	code	Code inverse
0	A	10	0	1
1	B	15	10	01
2	C	20	110	001
3	D	25	1110	0001
4	E	35	11110	00001
5	F	40	111110	000001
6	G	45	1111110	0000001

Tableau II.1 : Le code Unaire [9].

D'après le tableau on peut distinguer quelques remarques :

1. On peut utiliser le code inverse, c'est le même principe avec qu'on a expliqué déjà juste que se code inverse c'est une suite de N bits à '0' suivis de un bit à '1' à la fin.
2. Toujours le symbole de plus grande fréquence ce code est le 0, on n'ajoute pas un 1.
3. Le code unaire respecte toujours la propriété de code préfixe.

8.2. Le code d'Elias (Elias Codes) [10]

Peter Elias a décrit plusieurs codes à longueur variables. Ce code est issu d'un formalisme mathématique appliqué aux entiers. En fait, pour chaque entier N, il existe un entier M tel que $2^M \leq N < 2^{M+1}$, on peut alors écrire $N=2^M+L$ avec L est au plus M bits de longueur ainsi on aura un code constitué de deux partie M et L. selon les paramètres M et L, Elias a proposé plusieurs codes.

- **Code Gamma**

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 16 codes Gamma

$1=2^0+0=1$ (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 011
$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=2^3+0=$0001 000	$16=2^4+0=$0001 0000

Tableau II.2. Code Gamma

Pour le processus de décompression se fait en deux étapes :

- Lire les zéros '0' depuis le code jusqu'à rencontrer le bit '1', trouver le nombre de '0' et noter M.
- Lire les M bits comme un entier L. et calculer $N=2^3+L$.

- **Code Delta**

Elias a proposé un autre algorithme de construction d'un VLC code qui est le code Delta 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 les deux codes M.L

Exemple : 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=2^3+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

Tableau II.3 : Code Delta

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. puis calculer $N=2^M+L$.

- **Code Omega**

Le code Omega est une autre variété du code Elias le tableau ci–dessous illustre quelques valeurs de ce code

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

Tableau II.4 : Code Omega

- **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	2
2	3	4	3
3	3	4	4
4	5	5	4
5-7	5	5	5
8-15	7	8	6-7
16-31	9	9	7-8
32-63	11	10	8-10
64-88	13	11	10
100	13	11	11
1000	19	16	16
10^4	27	20	20
10^5	33	25	25

Tableau II.5 : Taille des trois codes d'Elias

8.3. Algorithme de Shannon- Fano [8]

L'objectif de l'algorithme de Shannon-Fanon est de donner un code avec des longueurs variables et un code qui respecte la propriété de code préfixe.

L'idée générale est de classer d'abord les symboles source par ordre de probabilité décroissante (de haut en bas), puis diviser l'ensemble des symboles en deux sous ensemble de telle sorte que les deux sommes des occurrences soient à peu près égales. L'un des groupes aura le préfixe 0, l'autre 1. Ensuite on recommence récursivement dans chacun des deux groupes : on répartit leurs symboles en deux sous-groupes de façon que les occurrences soient équilibrées...

Voici l'écriture récursive de cet algorithme

```
   Initialiser l'arbre binaire à vide
appeler la fonction CreerCodes("", tous les symboles)
def fonction CreerCodes(prefixe, groupe de symboles avec leurs occurrences):
si le groupe ne contient qu'un seul symbole, alors sortir de la fonction, le
code de ce symbole est : préfixe
couper le groupe en deux sous-groupes tels que les sommes des occurrences soit
équivalents (note n°1)
CreerCodes(prefixe+"0", premiergroupe)
CreerCodes(prefixe+"1", second groupe)
```

On va appliquer cet algorithme sur l'exemple suivant :

Soit la chaine de symbole S = "CITRONTRESCONTRIT".

On commence par compter les occurrences : C/2, I/2, T/4, R/3, O/2, N/2, E/1, S/1, la somme des occurrences est bien 17=L longueur du signal.

Il faut faire deux groupes de symboles de manière à ce que leurs sommes soient d'environ 17/2. Il y a généralement plusieurs possibilités. Par exemple, on peut proposer ce découpage : groupe 0 = {C/2, T/4, O/2, E/1} et groupe 1 = {I/2, R/3, N/2, S/1}.

Ensuite dans le premier groupe, il faut parvenir à deux groupes de poids à peu près égaux : sous-groupe 00 = {C/2, O/2, E/1} et sous-groupe 01 = {T/4}. Dans l'autre groupe : sous-groupe 10 = {I/2, N/2} et sous-groupe 11 = {R/3, S/1}. Encore une fois, il y a d'autres

possibilités pour découper en sous-groupes, on verra après la conséquence sur le taux de compression.

Il reste encore des groupes non réduits à des singletons, on peut les découper ainsi : sous-sous-groupe 000 = { C/2 }, sous-sous-groupe 001 = { O/2, E/1 }, sous-sous-groupe 100 = { I/2 }, sous-sous-groupe 101 = { N/2 }, sous-sous-groupe 110 = { R/3 }, sous-sous-groupe 111 = { S/1}.

On répète encore pour obtenir : sous-sous-sous-groupe 0010 = {O/2 }, sous-sous-sous-groupe 0011 = { E/1 }.

L'algorithme s'arrête car tous les groupes sont réduits à des singletons. On obtient donc le codage : C = 000, I = 100, T = 01, R = 110, O = 0010, N = 101, E = 0011, S = 111.
Le signal S est codé = "000100011100010101011100011111000001010101110100001". Pour décoder ce code il faut d'abord faire une table qui contient les symboles avec le code produit par l'algorithme précèdent après en décode on suivant la table.

8.4. Algorithme de Huffman [8]

Dans l'algorithme de Huffman, on procède différemment pour un résultat comparable. On commence par choisir les deux symboles qui ont le moins d'occurrences, on leur donne les codes 0 et 1 et on les regroupe dans un arbre binaire auquel on attribue un nombre d'occurrences : la somme des deux symboles qu'il regroupe. Ensuite, on continue : on choisit deux symboles ou arbres qui ont le moins d'occurrences et on les regroupe. On regroupe ainsi deux par deux des symboles ou des arbres en fonction du nombre d'occurrences qu'ils représentent.

On va appliquer cet algorithme sur le même exemple qu'on a déjà appliqué à l'algorithme de Shannon-Fanon pour voir la déférence entre eux.

Soit le signal S = "CITRONTRESCONTRIT".
On commence 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, on met la branche 0 à gauche, la branche 1 à droite. On continue donc avec { C/2, I/2, T/4, R/3, O/2, N/2, {E, S}/2 }. On a 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 }.

On continue et on obtient { {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 }.

On continue : { {{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 :

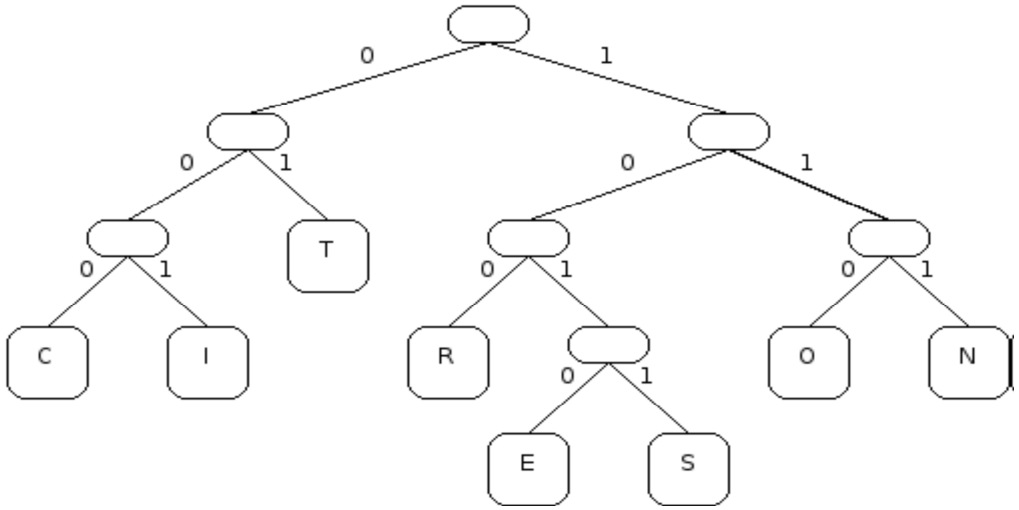


Figure II.6 : Exemple de constriction de l'arbre de Huffman [8].

On obtient donc les codes suivants :

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

Le signal S est codé = "000101011001101111011001010101100011011101100101 01" sur 49 bits. Ce qui fait 1 bit de moins qu'avec le codage de Shannon. C'est dû aux choix qu'on a fait de regrouper tel ou tel symbole. Dans d'autres exemples, les deux procédés sont équivalents.

Remarque

En principe, le codage de Huffman donne toujours le meilleur résultat possible en termes de compression d'un signal aléatoire. Avec ces algorithmes, les données compressées doivent être accompagnées de la table ou arbre de codage.

9. Conclusion

Dans ce chapitre nous avons présenté quelques définitions sur la méthode VLC, et quelques méthodes de construction des codes préfixes (code unaire, algorithme de Huffman et Shannon. Un code préfixe est un code uniquement décodable.

Ces méthodes de compression sont statiques qui exigent deux lectures (l'une pour le calcule des fréquences et l'autre pour le codage). Dans le prochain chapitre on va traiter une nouvelle méthode dite la méthode dynamique qui fait une seule lecture.

1. Introduction

Dans le chapitre précédent nous avons étudies la méthode de codage statique mais ces méthodes ne sont pas efficaces car chaque fois le codeur requiert de connaitre à l' avance les fréquences d'apparition des symboles d'un alphabet pour construire l'arbre qui est utilisée pour obtenir les mots de code. Malheureusement, cela implique que l'encodage doit être fait en deux phases : une phase pour accumuler les statistiques et une phase pour encoder les données. Cet aspect du codage peut être problématique quand la rétention des données est difficile ou lorsque la technique est utilisée en temps réel.

Pour régler ce problème on va proposer dans ce chapitre deux techniques qui sont de type de compression des données adaptatif, Huffman adaptatif et LZW.

2. Huffman adaptatif [3]

Le codage de Huffman adaptatif est une variante du codage de Huffman qui ne requiert pas de connaitre à l'avance les fréquences d'apparition des symboles : au fur et à mesure que les données sont lues, l'arbre de Huffman, qui est utilisé pour obtenir les mots de code, est mis à jour. Il n'est donc plus nécessaire de procéder en deux phases pour encoder les données : une seule phase suffit. De plus, un décodeur n'a plus à se faire informer de l'arbre de Huffman qui a été utilisé pour obtenir les mots de code : il peut désormais construire l'arbre de Huffman lui-même en s'y prenant de la même façon que l'encodeur.

2.1. Principe de l'algorithme de Huffman adaptatif (dynamique)

L'algorithme de Huffman dynamique permet de compresser un flot à la volée en faisant une seule lecture de l'entrée ; à la différence de l'algorithme statique d'Huffman, il évite de faire deux parcours d'un fichier d'entrée (un pour le calcul des fréquences, l'autre pour le codage). La table des fréquences est élaborée au fur et à mesure de la lecture du fichier ; ainsi l'arbre de Huffman est modifié à chaque fois qu'on lit un nouveau caractère.

2.2. Algorithme de compression

On suppose qu'on doit coder un fichier de symboles binaires lus à la volée par blocs de k bits (k est souvent un paramètre) : on appelle donc « caractère » un tel bloc. A l'initialisation, on définit un caractère symbolique (noté @ par exemple) et codé initialement

par un symbole prédéfini (par exemple 0). Lors du codage, à chaque fois que l'on rencontre un nouveau caractère pas encore rencontré, on le code sur la sortie par le code de @ suivi de k bits du nouveau caractère. Le nouveau caractère est alors entré dans l'arbre de Huffman. Pour construire l'arbre de Huffman et le mettre à jour, on compte le nombre d'occurrences de chaque caractère et le nombre de caractères déjà lus ; on connaît donc, à chaque nouvelle lecture, la fréquence de chaque caractère depuis le début du fichier jusqu'au caractère courant, les fréquences sont donc calculées dynamiquement. Après avoir écrit un code (soit celui de @, soit celui d'un caractère déjà rencontré, soit les k bits non compressés d'un nouveau caractère), on incrémente d'un le nombre d'occurrences du caractère écrit. En prenant en compte les modifications de fréquence, on met à jour l'arbre de Huffman à chaque itération.
L'arbre existe donc pour la compression (et la décompression) mais n'a pas besoin d'être envoyé au décodeur.

```
Soit nb(c) le nombre d'occurrence d'un caractère  c
Initialement l'arbre de huffman (AH) avec le caractère  @
    While on n'est pas à la fin de la source do
          Lire le caractère c de la source
           If  c'est la première occurrence de c then
                Nb (c) := 0 ;
                Nb(@) := nb(@) + 1 ;
                Afficher en sortie le code de @ dans (AH) suivi de c ;
           Else
                Afficher le code de c dans (AH) ;

           Endif
        Nb(c) := nb(c) + 1 ;
        Mètre à jour (AH) ave les nouvelles fréquences ;
  End.
```

2.3. Algorithme de décompression

A l'initialisation le décodeur connait un seul code celui de @, par exemple 0. Alors Il lit le 0 qu'est un code associe au symbole @, il en déduit que les k bits suivants contiennent un nouveau caractère. Il écrit les k bits et met à jour l'arbre de Huffman, qui contient déjà le symbole @ avec ce nouveau caractères.

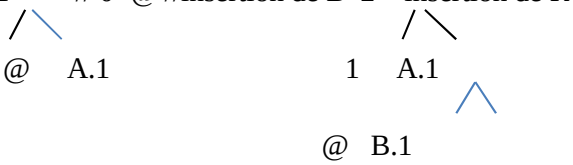
Le point critique est que le codeur et le décodeur maintient chacun leur propre arbre de Huffman, mais utilisent tous les deus le même algorithme déterministe pour mettre ajour à partir des occurrences (fréquences) des caractères déjà lu. Ainsi les arbres de Huffman calculés séparément par le codeur et le décodeur sont exactement les mêmes.

Puis le décodeur lit le code suivant et le décode via son arbre de Huffman. Si il s'agit de code de @, il lit les k bits correspondant à un nouveau caractère, il les écrit sur la sortie et ajoute à son arbre de Huffman (le nouveau caractère est désormais associe à un code). Si non, il s'agit d'un caractère déjà rencontré via son arbre de Huffman il trouve les k bits associe au caractère et les écrits sur la sortie. Il incrémente d'un le nombre d'occurrence du caractère qui vient d'écrire et de @ s'il s'agit de nouveau caractère et mettre ajour l'arbre de Huffman.

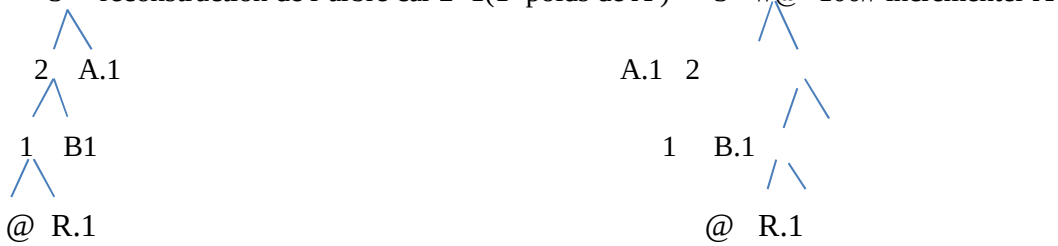
```
Soit nb(c) le nombre d'occurrence d'un caractère  c
Initialiser l'arbre de Huffman (AH) avec le caractère @
While on 'est pas à la fin de code C_B do
        Lire le code C_B ;
        B= C_B ;
        Char c = décoder le code (AH, B) ;
                If (c !=@) then
                   { Envoyer  char(c);  }
                Else
                { nb(@)= nb(@) + 1;
                    Lire l'octet C_B ;}
         Nb(c) = nb(c)+1 ;
          Mettre ajour l'arbre (AH) ;
```

Exemple d'application de l'algorithme de Huffman adaptatif

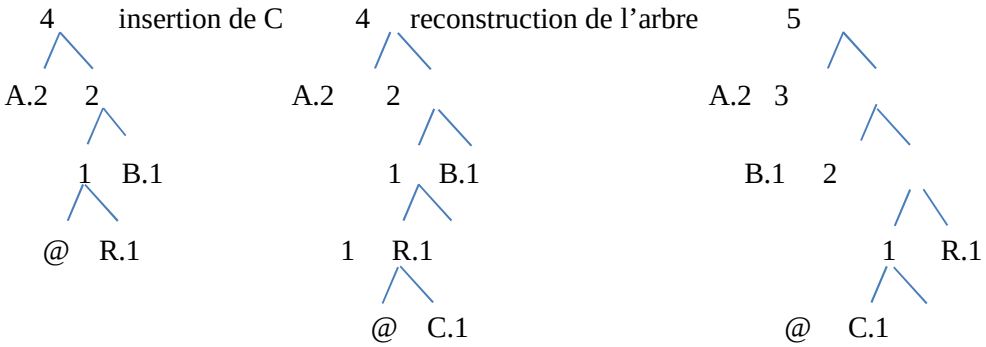
@ // ' '=@ //insertion de A 1 // 0=@ //insertion de B 2 insertion de R



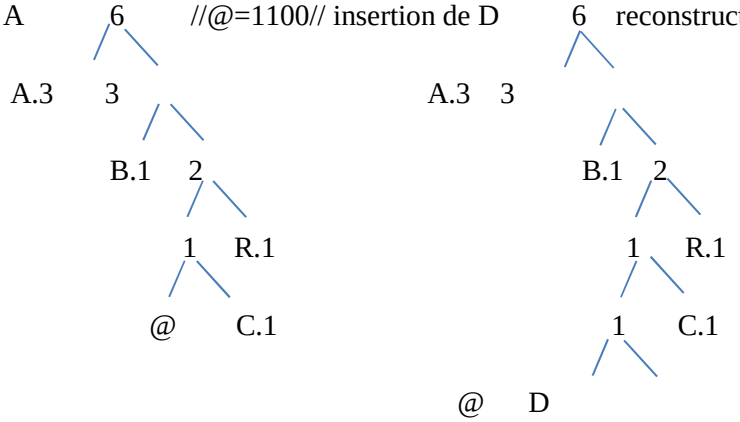
3 reconstruction de l'arbre car 2>1(1=poids de A) 3 //@=100// incrémenter A

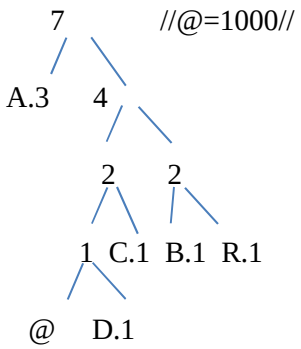


4 insertion de C 4 reconstruction de l'arbre 5



Incrémentation de A 6 //@=1100// insertion de D 6 reconstruction d'arbre





A la fin nous aurons le code suivant (0, 110, 101, 1001,111) pour l'alphabet (A, B, C, D, R).

Remarque : l'arbre de Huffman dans ces exemples est complet c.-à-d. que chaque nœud possède 0 fils ou 2 fils.

3. La méthode LZW [3]

3.1. Définition

La méthode LZW est l'une des meilleures méthodes conservative de compression, mais elle reste la moins utilisée en raison de sa protection par un brevet. Le principe de l'algorithme LZW, c'est d'utiliser un dictionnaire dynamique qui contient des motifs du fichier traité. L'inconvénient des algorithmes à dictionnaire, c'est qu'il est à priori nécessaire de transmettre le dictionnaire avec les données, ce n'est pas le cas avec LZW. Le dictionnaire est construit à la compression et reconstruit à la décompression. Ce qui est également intéressant, c'est qu'il est inutile de lire et d'étudier le fichier avant de le compresser. La compression s'effectue au fur et à mesure de la lecture, le compresseur détecte au fur et à mesure les séquences qui se répètent.

3.2. Historique de LZW :

En 1977, Abraham Lempel et Jacob Ziv ont introduit un algorithme de compression. Une année après (en 1978), ils modifièrent cet algorithme et le référencèrent par « compression basée dictionnaire ». L'abréviation du premier algorithme est LZ77 et son successeur est LZ78. Vu les problèmes de gestion de mémoire et de dictionnaire de la méthode LZ (LZ77 et LZ78), Terry Welch a modifié le LZ78 en 1984 en proposant un nouvel algorithme appelé LZW.

3.3. Caractéristiques de LZW :

La méthode LZW a certaines caractéristiques, comme :

- L'utilisation d'un dictionnaire qu'elle construit dynamiquement, au cours de la compression et de la décompression.
- Le dictionnaire n'est pas stocké dans le fichier compressé (pas de table d'entête).
- Elle comprime en une seule lecture (à la volé) ce qui donne une compression en un temps meilleurs que les méthodes statistique.
- Pour être efficace et reconnaitre des longues chaines, elle a besoin d'un apprentissage

3.4. Compression :

Dans le processus de compression, l'algorithme construit un dictionnaire où chaque séquence sera numérotée par une adresse dans celui-ci. Chaque séquence ne se trouvant pas dans le dictionnaire sera rajoutée. A la fin, on aura des adresses pointues vers une séquence contenue dans le dictionnaire. La taille de dictionnaire n'est pas fixe et les 256 premières adresses (de 0 à 255) du dictionnaire contiennent les codes ASCII. La première séquence formée aura l'adresse 256.

3.4.1. Algorithme de compression :

Données :

- le dictionnaire des symboles rencontrés : Dico
- le fichier à compresser : Fichier

Résultat :

- le fichier compressé : Fichier

L'algorithme :

début

s=premier octet du fichier

tant que *le fichier n'est pas à sa fin*

t=octet suivant

u=concaténation(s,t)

si *(u appartient Dico)*

```
            s=u
            sinon
                ajouter (u) dans Dico
                  écrire adresse de s
                  s=t
              fin si
        fin tant que
        écrire adresse de s
    fin
```

Exemple : soit à compresser la chaine « RECHERCHE-CHAT» en LZW.

3.4.1.1. Explication de déroulement de l'algorithme :

On lit le premier caractère soit 'R'. puis on lit le suivant 'E'. on forme la séquence **u=s+t** soit 'RE', cette séquence n'est pas dans le dictionnaire, donc on l'ajoute dans ce dernier. cette séquence aura l'adresse 256. Enfin on attribue l'adresse de **s** (valeur ASCII de 'R') dans le fichier soit **82** (1010010 en binaire). On refait cette action jusqu'à la fin du fichier.

La compression effective aura lieu lorsqu'on rencontrera pour la seconde fois une paire ('CH' par exemple) déjà présenté dans le dictionnaire. Dans ce cas on émettra l'adresse de la séquence 'CH' et non pas l'adresse de la séquence 'C' et celle de la séquence 'H'. On ajoutera par la suite dans le dictionnaire la séquence de **3** caractères (soit 'CHE' dans notre exemple).

Au début les codes sont sur 8 bits pour les 256 premières entrées du dictionnaire (de 0 à 255), puis continue avec 9 bit pour les entrées de 256 à 511, et ainsi de suite.

Les étapes du déroulement du l'algorithme sont présenté dans le tableau ci-dessous :

étape	Val S	Val t	Val u	U ∈ Dico	Ajouter u à dico	Ecrire @ de S
1	R	E	RE	non	RE → 256	R → 82
2	E	C	EC	non	EC → 257	E → 69
3	C	H	CH	non	CH → 258	C → 67
4	H	E	HE	non	HE → 259	H → 72
5	E	R	ER	non	ER → 260	E → 69
6	R	C	RC	non	RC → 261	R → 82
7	C	H	CH	oui		
8	CH	E	CHE	non	CHE → 262	CH → 258
9	E	-	E-	non	E- → 263	E 69
10	-	C	-C	non	-C → 264	- → 45
11	C	H	CH	oui		
12	CH	A	CHA	non	CHA → 265	CH → 258
13	A	T	AT	non	AT → 266	A → 65
14	T					T → 84

Tableau III.1 : Exemple de compression avec la méthode LZW.

s : symbole lu.
t : symbole suivant.
u : s+t.
Dico : dictionnaire.

Le dictionnaire résultant de l'algorithme est :

Mot	Code
ASCII	0..255
RE	256
EC	257
CH	258
HE	259
ER	260
RC	261
CHE	262
E-	263
-C	264
CHA	265
AT	266

Tableau III.2 : Le dictionnaire résultant par l'algorithme LZW

La chaine compressée est : 82.69.67.72.69.82.258.69.45.258.65.84

3.5. Décompression :

Concernant la décompression, l'algorithme LZW reconstruit le dictionnaire au fur et à mesure de la lecture du fichier compressé. Ce processus est très proche de celui de la compression.

Le processus de compression débute par la lecture des codes du fichier en partant d'une taille de 8 bits. Puis suivant la méthode utilisée, lorsque le décompresseur rencontre un code spécial il augmente la taille de son tampon pour lire les adresses. Il continue à lire les adresses jusqu'à la fin du fichier et il inscrit la séquence correspondante à l'adresse dans le fichier décompresser.

3.5.1. Algorithme de décompression :

Données :

- le dictionnaire des symboles rencontrés : Dico
- le fichier à décompresser : Fichier

Résultat :

- le fichier décompressé : Fichier.

L'algorithme :

début
a=Séquence(première adresse contenu dans le fichier)
écrire a
tant que *le fichier n'est pas à sa fin*
b=adresse suivante
si *(b appartient Dico)*
s=Séquence(b)
sinon
s=concaténation(a,t)

fin si

écrire séquence(s)

t=s[0]

ajouter (Séquence(a)+t) dans Dico

a=b

fin tantque

fin

Exemple : soit à décompresser le « 82.69.67.72.69.82.258.69.45.258.65.84».

3.5.1.1. Explication de déroulement de l'algorithme :

On lit la première adresse soit 82. On écrit la séquence associé à cette adresse du dictionnaire. Ensuite on lit l'adresse suivante. On s'assure que l'adresse lue appartient au dictionnaire, puis on écrit la séquence associé à cette adresse. On ajoute dans le dictionnaire la concaténation de l'ancienne séquence et la première lettre de la séquence en cours. On refait ensuite cette action jusqu'à la fin du fichier.

Les étapes du déroulement du l'algorithme sont présenté dans le tableau ci-dessous :

étape	Val a	Val b	b∈ Dico	Val s	Val t	Ajouter au Dico	Ecrire la séquence
1	R=séq(82)	69	oui	E=séq(69)	E	RE→ 256	R puis E
2	E=séq(69)	67	oui	C=séq(67)	C	EG→ 257	C
3	C=séq(67)	72	oui	H=séq(72)	H	CH→258	H
4	H=séq(72)	69	oui	E=séq(69)	E	HE→ 259	E
5	E=séq(69)	82	oui	R=séq(82)	R	ER→ 260	R
6	R=séq(82)	258	oui	CH=séq(258)	C	RG→ 261	CH
7	CH=séq(258)	69	oui	E=séq(69)	E	CHE→262	E
8	E=séq(69)	45	oui	- =séq(45)	-	E→ 263	-
9	- =séq(45)	258	oui	CH=séq(258)	C	-C→ 264	CH
10	CH=séq(258)	65	oui	A=séq(65)	A	CHA→265	A
11	A=séq(65)	84	oui	T=séq(84)	T	AT→266	T

Tableau III.2 : La décompression avec la méthode LZW.

Le dictionnaire résultant est le même que celui de la compression.

On remarque que l'algorithme de décompression est plus rapide que celui de la compression (11 étapes contre 14 pour la compression dans notre exemple).

La chaine décompressée est : « RECHERCHE-CHAT».

3.6. Les variantes de LZW

Il existe plusieurs variantes des compressions LZW, parmi celle-ci on trouve :

LZC :

LZC est la variante de l'algorithme LZW qui est utilisé dans le système d'exploitation libre UNIX. Il compresse selon le principe LZW. La longueur maximale d'un index peut être définie par l'utilisateur du programme de compression en tenant compte de la mémoire disponible (de 9 à 16 bits). Il commence d'abord par 9 bits pour les 512 premières entrées du dictionnaire, puis continue avec 10 bit pour les entrées de 512 à 1023, et ainsi de suite jusqu'à la limite spécifiées par l'utilisateur. Enfin, l'algorithme devient un dictionnaire statique, vérifie régulièrement le taux de compression. Quand il détecte une baisse de taux de compression, il jette le dictionnaire et commence à construire un nouveau à partir de zéro.

LZT :

C'est une autre variation sur le thème LZW, l'algorithme est presque la variante LZC, la seule différence étant qu'il fait de la place pour de nouvelles phrases à ajouter au dictionnaire en enlevant le moins récemment utilisé.

3.7. Performance de LZW :

On se basant sur l'exemple traité dans les sections 5.1.2 et 5.2.2 on a :

La taille de fichier original est : (14*8)=112 bit

La taille de fichier compressé est : (10*8) + (2*9)= 98 bit

Le gain en espace est : G= (1- (98/112)) * 100= 12.5%

Lorsque la taille du fichier est grande, le taux de compression est supérieur à celui obtenu par une méthode de compression statistique. En plus de taux de compression,

l’algorithme LZW est nettement plus rapide qu’un algorithme de type statistique en termes de traitement.

4. Conclusion

Dans ce chapitre nous avons vus deux méthodes de compression adaptative, la méthode de Huffman adaptatif et la méthode de LZW. Tel que la méthode de Huffman est basée sur le calcul de fréquence des symboles et la mise ajour de l’arbre mais la méthode de LZW est basée sur un dictionnaire.

1. Introduction

Dans ce chapitre nous allons présenter la méthode utilisée pour la compression des données, cette méthode s'appelle la méthode de compression de données adaptative (VLC) temps réel. Dans ce chapitre nous présenterons l'architecture de notre application en expliquant les processus de compression et de décompression, nous présenterons ainsi le processus de codage et de décodage des données. Elle doit fournir en plus des fonctionnalités de compression/décompression telles que le calcul de temps, la taille du fichier avant et après la compression, et le taux. Donc ce chapitre comporte deux parties :

La première partie, c'est la présentation générale de la nouvelle méthode de compression/décompression de données avec des exemples.

La deuxième partie, nous allons expliquer et présenter les différents modules nécessaires à la mise en œuvre de l'application.

2. Présentation générale de la méthode

Dans cette partie on va essayer d'expliquer les deux variantes qui sont importantes dans la méthode VLC la partie suffixe et la partie préfix. La méthode VLC peut se classer parmi les méthodes réversibles (sans perte) car elle reconstitue exactement le fichier original après la décompression. Elle peut être donc utilisée aussi bien pour résoudre le problème de l'espace de stockage surtout lorsqu'il s'agit de compresser des fichiers textes.

2.1.1 Le Code proposé

Le code proposé est un code qui est constitué de deux parties : une partie fixe et une partie variable.

A. Codage

Le codage fixe est facile à générer. Pour un code fixe sur N bits on aura 2^N Symboles $S= \{S_0, S_1, \dots S_{2^n-1}\}$.

Le codage VLC correspondant à cet ensemble de symboles est :

$$C(S_i) = (preffix)_i.(suffix)_i \ / \ i \in [0, 2^n - 1]$$

$C(S_i)$: Codage du symbole S_i.

- ✓ **La signification des deux champs $preffix$ et $suffix$:** Le code VLC proposé est constitué d'une partie fixe ($preffix$) et une partie variable ($suffix$).

- **$preffix$:** ce champ représente deux (02) informations :
  1) Il indique la position (P) du bit le plus significatif dans le code fixe de S_i. Sa valeur va de 0 à N-1 c'est-à-dire :

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

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

Exemple 1 : $S_4 = (4)_{10} = (\underbrace{00 \ldots \ldots 100}_{N\ bits})_2$

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

  2) Il indique la longueur du champ $suffix$

La Longueur de la partie préfixe : 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 :

$$\log_2(8) + 1 = \frac{\log(8)}{\log(2)} + 1 \approx 1 + 3 = 4 \text{ bits}$$

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) :
$L(preffix) = L(N-1) = log_2((N-1)+1) = log_2(N)$ *bits.*
$L(preffix)$ *:* Longueur du $preffix$ en nombre de bits.

- ***suffix :*** c'est la partie restante du code fixe S_i qui suit le bit significatif.

1. $\text{Val}(\text{suffix}_i) = S_i - 2^{\text{preffix}_i} \rightarrow S_i = 2^{\text{preffix}_i} + \text{suffix}_i$
2. $L(suffix_i) = \begin{cases} 1 & si\ preffix_i = 0 \\ preffix & sinon \end{cases}$

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

En reprend le même code S_4 de l'exemple 1 et on retrouve la valeur du $suffix$.

$S_4 = \left(\underbrace{00 \ldots \ldots 1 \ \underbrace{\mathbf{00}}_{\text{suffix}}}_{N\ bits} \right)_2$, de là on a :

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

A partir de l'exemple 1 et l'exemple 2, on peut conclure le code VLC correspondant au symbole $S_4 = (00\ldots100)_2 \rightarrow C(S_4) = preffix_4.suffix_4 = \underbrace{010}_{preffix}\ \underbrace{\mathbf{00}}_{suffix} = 01000$

Exemple 3 : N=4 bits, $S_{i=1,15}$ l'ensemble des symboles.

Compression de la chaine CH=$S_1S_1S_1S_2\ S_3S_3$.

La représentation binaire de CH est 0001 0001 0001 0010 0011 0011, la taille de CH est : L(CH)=4x6=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 (S_i)	Probabilité P (s_i)
0	S_1	0.5
1	S_3	0.33
2	S_2	0.16

✓ Construire le code VLC

Au lieu de coder les symboles eux-mêmes, on procède au codage des indices des symboles.

$$C(S_i) = Preffix_i . Suffix_i$$

$$N = 4\ bits \rightarrow L(preffix_i) = \log_2 4 = 2\ bits$$

Indice	Indice binaire	Position du bit poids fort	Code C (s_i)	
			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

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

CH_c : chaine compressée. L(CH_c) = 3x6 =18 bits.

La moyenne de bits par symbole :

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

$$\sum_{i=1}^{N} P(s_i)\ L(c_i) \approx 3\ bits/symbole.$$

B. Décodage

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

✓ Lire Log_2 N premier bits($preffix$)

✓ 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.

C. 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 : $L(suffix_i) = \begin{cases} 1 & si\ preffix_i = 0 \\ preffix & sinon \end{cases}$

Donc :

$$L(C_i) = \begin{cases} \log_2(N) + 1\,, & i = 0 \\ \log_2(N) + preffix_i\,, & \forall\ i \in [1, 2^N - 1] \end{cases} \quad \ldots\ldots(2)$$

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

Et sachant que : $preffix_i \in [0, N-1]$ → Le nombre de taille du codes est N différentes tailles, Alors (2) devient :

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

Remarque : pour un ensemble de symboles sur N bits, on a 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.

$$C(S_i) = preffix_{\ i}\ .suffix_{\ i}\ , \forall\ i \in [0, 2^8 - 1]$$

Calculer le nombre de bits de la partie préfixe :

$$L(preffix_{\ i}) = \log_2(n) = \log_2(8) = \frac{\log(8)}{\log 2} = 3\ bits.$$

Le tableau suivant illustre les différents codes:

Indice	**Symbole S_i**	**bit poids fort**	**Code C_i**		**$L(C_i)$ $Log_2(8)+p$**	**Nombre de code $[2^{L(suffix)}]$**
			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

Tableau IV.1 : Codes VLC pour N=8.

- **Code adaptatif**

On a déjà vu comment ça déroule la méthode VLC avec ces deux variantes (préfix et le suffixe) mais dans notre cas, c'est insuffisant, on doit le mettre adaptatif temps réel. Alors notre méthode doit s'adapter au nombre de caractères qu'on doit coder.

Nous avons mentionné plus haut, la longueur du préfixe est donnée par $\log_2(N)$, tel que N représente le nombre de bits sur lequel sont codés les symboles, donc on aura 2^N symboles (caractères) possibles à coder, et delà on peut déduire que :

la longueur de préfixe de VLC adaptatif égale à $\log_2(\log_2(2^N))$.

Mais dans le cas ou le nombre de symboles est beaucoup inferieur à 2^N soit par exemple n, alors la taille du préfixe sera $\log_2(\log_2(n))$ au lieu de $\log_2(N)$.

Or par cette stratégie on pourra minimiser de la taille du préfixe dans la plus part du temps.

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 on a besoin seulement de coder.

Longueur du préfixe :

- dans le 1^{ere} cas : préfixe = $\log_2(N) = \log_2(8) = 3$
- dans le 2^{eme} cas : préfixe = $\log_2(\log_2(16)) = \log_2(4) = 2$

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

3. Analyse et conception de la méthode proposée

Pour l'analyse et la conception, on va utiliser une architecture facile à comprendre, basée sur une approche modulaire. Les modules sont des entités indépendantes intégrées dans une architecture pour produire une application.

Pour réaliser ce que nous avons dit sur notre application on doit suivre une méthode de conception appelée : la méthode descendante qui consiste à décomposer le problème en sous problème.

3.1. Les 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 crée. Nous allons ainsi appliquer les concepts vus au préalable pour définir les différents modules qui la constituent.

Pour faire notre application, on doit alors offrir la possibilité de gérer les différentes tâches et les demandes de l'utilisateur. Elle doit accomplir la compression adaptative, la décompression adaptative, le calcul, l'affichage des résultats, les performances de compression / décompression et éventuellement les sauvegarde.

Donc nous pouvons décomposer le noyau de l'application en trois modules principaux :

1. Module de compression adaptatif.
2. Module de décompression adaptatif.
3. Module de calcule des performances.

La figure qui suit nous montre cette décomposition :

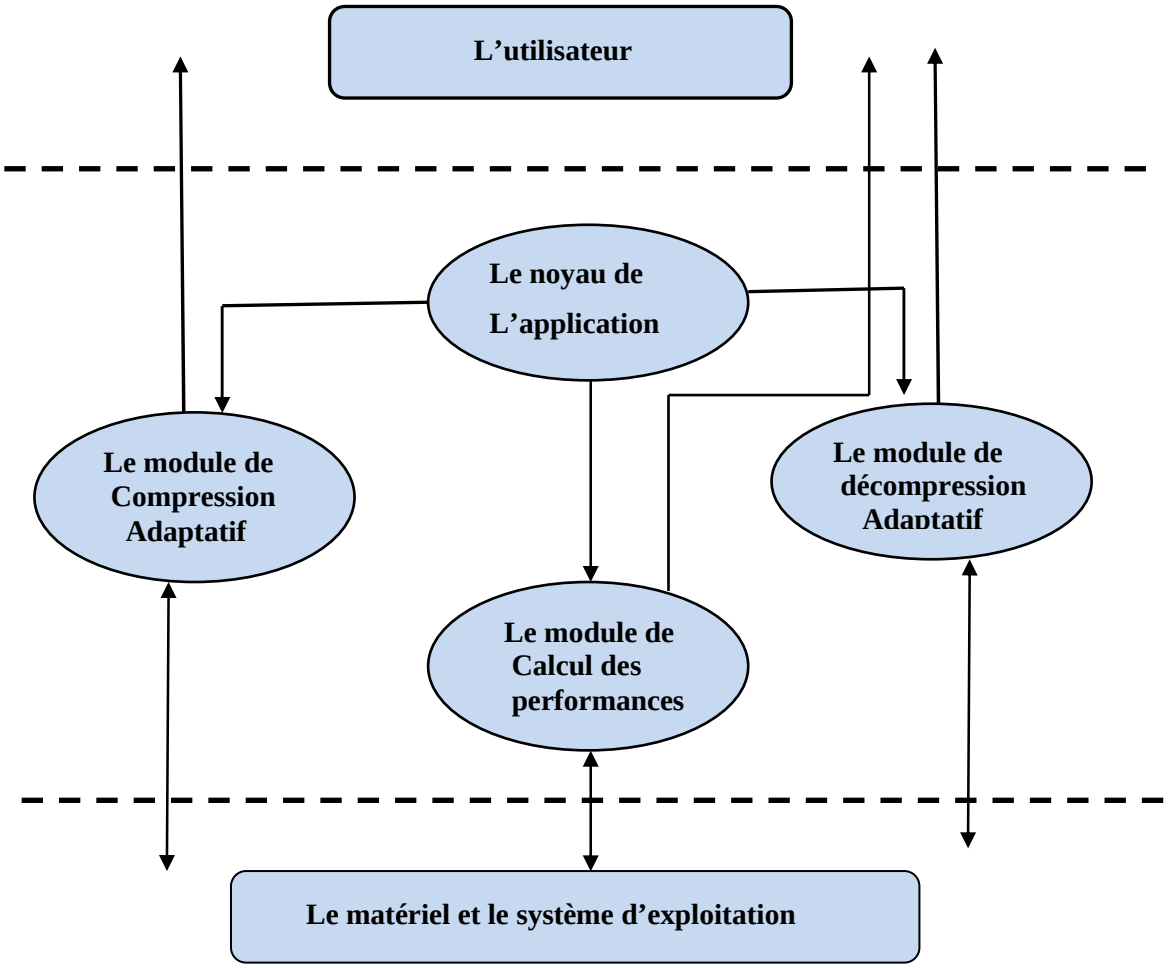


Figure IV.1: décomposition du module principale de l'application

3.1.1 Le module de compression

Avant de lancer l'opération de compression, l'utilisateur doit d'abord donner un fichier en entrée. Au lancement de la compression ce module va lire caractère par caractère, en ce moment le module compressé va créer deux tables :

La première c'est la table d'indexe qui va affecter à chaque symbole un indexe

La deuxième c'est la table code qui sert à coder et calculer les fréquences des symboles. Pendant ce processus les deux tables indexe et code vont se mettre à jour en même temps

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

1. Lecture des caractères courants.
2. La mise à jour la table des indexes.
3. Codage des symboles.

La figure ci- dessus donne le schéma général de ce module :

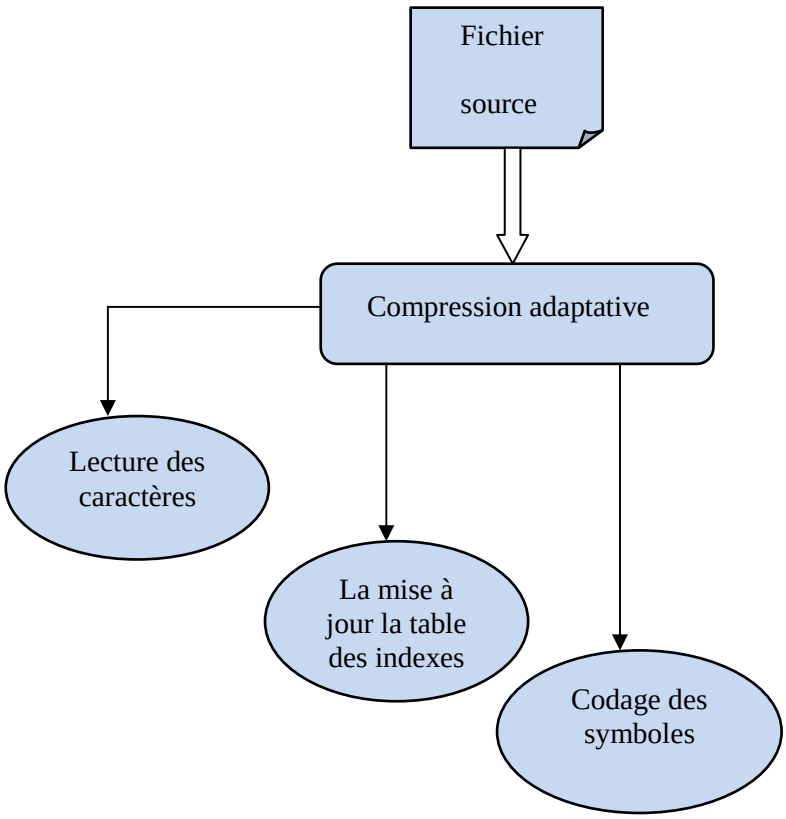


Figure IV.2: schéma général du module de compression.

Exemple

Nous prenons par exemple un fichier source avec 6 caractères de 3 symboles,

Fichier source= (aabbbc).

1ere lecture : on lit le caractère "a", on l'insert dans la table d'indexe.

index	symbole
0	a

On crée une table de code, dans cette dernière on code les symboles et on calcule leurs fréquences.

Code de symbole	Fréquence de symbole
$a=(01100001)_2$	1

2eme lecture : on lit toujours un "a", comme on a déjà rencontré le symbole "a" on incrémente seulement la fréquence.

Code de symbole	Fréquence de symbole
$a=(01100001)_2$	2

3eme lecture : on lit un nouveau symbole, alors on l'insert dans la table d'indexe et dans la table de code.

indexe	symbole
0	a
1	b

Code de symbole	Fréquence de symbole
$a=(01100001)_2$	2
$b=(01100010)_2$	1

Dans les prochaines lectures, lecture 4et 5 on remarque que la fréquence de symbole "b" est supérieure à celle de symbole "a", dans ce cas il faut mettre à jour la table d'indexe et la table de code.

indexe	symbole
0	b
1	a

Code de symbole	Fréquence de symbole
a=$(01100010)_2$	3
b=$(01100001)_2$	2

On répète le même processus jusqu'à la fin de fichier.

indexe	symbole
0	b
1	a
2	c

Code de symbole	Fréquence de symbole
b=$(01100010)_2$	3
a=$(01100001)_2$	2
c=$(01100011)_2$	1

Le résultat final est un fichier compressé avec le code suivant :

001100001**100**01100010**11100**01100011

3.1.2. Le module de décompression

Avant de lancer la décompression, nous devons d'abord choisir un fichier déjà compressé. Au lancement de la décompression, ce module fait toujours un teste sur le premier bit de code. Si c'est un bit à "0", il va lire les 8 bits du code car c'est un caractère.si c'est un bit à "1", il va calculer le préfixe et le suffixe parce que c'est un code d'un caractère déjà lu. Ensuite, il va décoder ce code et il va lui affecter le symbole correspondant.

D'après cette description nous pouvons déduire que le module de décompression est décomposée en sous modules qui suivent :

1. Lecture de code.
2. Teste sur le premier bit code.
3. Lecture des 8 bits ou calcul de préfixe et de suffixe.
4. Décodage des données et affectation du symbole correspondant.

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

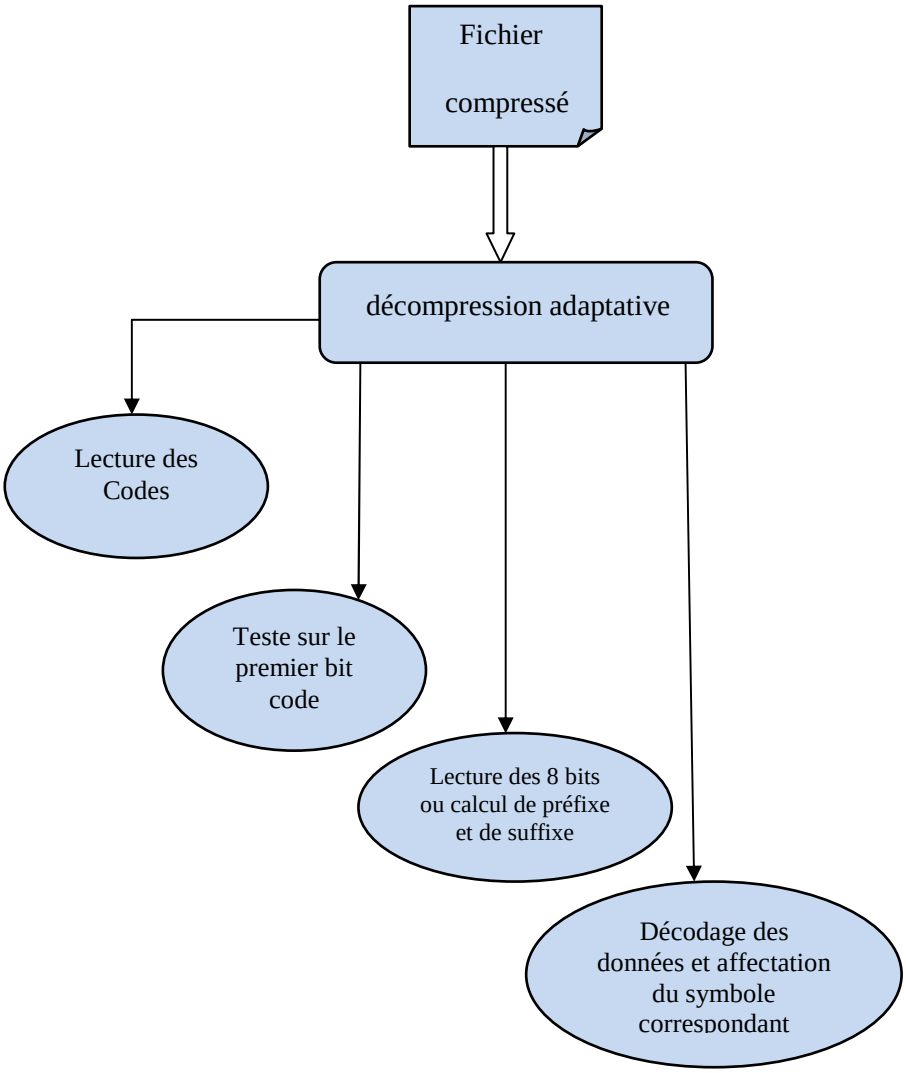


Figure IV.3: Schéma général du module de décompression.

Comme la compression adaptative est classée parmi les méthodes sans perte, alors le principe utilisé pour la décompression permet donc de décompresser un fichier compressé et de trouver le même fichier source sans aucune perte de données.

Exemple : on va décompresser le fichier compressé dans l'exemple précèdent, dont le code fichier compressé est (**0**01100001**100**01100010**11100**01100011).

Au début de chaque lecture on fait un teste sur le premier bit, si c'est bit à "0" alors c'est un caractère on lit les 8 bits de code si non on calcule préfixe et le suffixe du code.

1ere lecture : on lit le premier bit qu'est à "0", on lit les 8 bits.

Code de symbole	Fréquence de symbole
a=$(01100001)_2$	1

indexe	symbole
0	a

2eme lecture : on lit un bit à "1", alors c'est un code de caractère " a" on calcule le préfixe et le suffixe et on met à jour la fréquence de symbole "a".

Code de symbole	Fréquence de symbole
a=$(01100001)_2$	2

indexe	symbole
0	a

3eme lecture : on lit un bit à "0", alors c'est nouveau symbole, on lit les 8 bits

Code de symbole	Fréquence de symbole
a=$(01100001)_2$	2
b=$(0100010)_2$	1

indexe	symbole
0	a
1	b

Dans la lecture 4 et 5 on trouve toujours un bit à "1" on calcul préfixe et le suffixe, toujours c'est le symbole "b" dans ces deux dernières lectures. Il faut mettre à jour la table de code et la table d'indexe.

Code de symbole	Fréquence de symbole
b=$(01100010)_2$	3
b=$(0100001)_2$	1

indexe	symbole
0	b
1	a

A la fin nous affectons chaque code à son symbole correspondant:

Code de symbole	Fréquence de symbole
b=$(01100010)_2$	3
a=$(0100001)_2$	1
c=$(0100011)_2$	1

indexe	symbole
0	b
1	a
2	c

On répète le même processus jusqu'à l'obtention du même fichier source (aabbbc).

3.1.3. Module de calcul des performances

Lors des deux opérations de compression, il y a un module qui est chargé de calculer les performances de ces opérations, à savoir le taux de compression et de décompression et le temps de compression et de décompression. A la fin de l'opération le système affiche les résultats.

- **Calcul des performances pour la compression :**

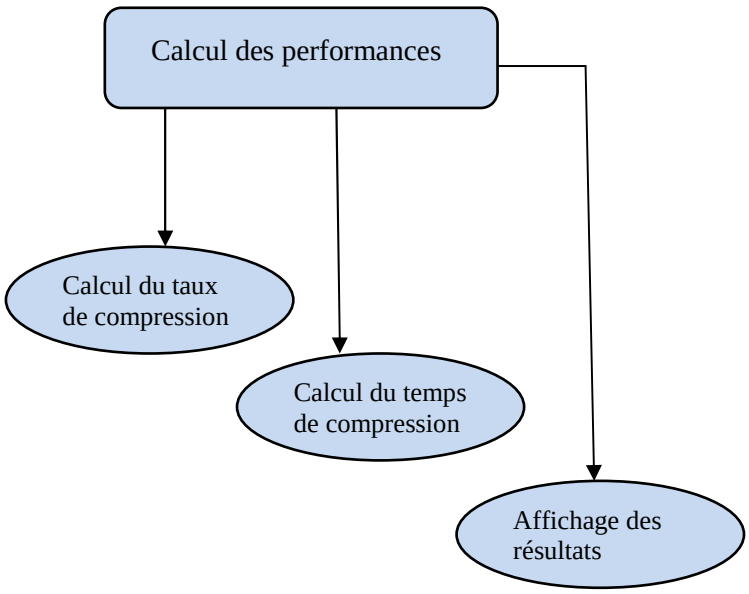


Figure IV.4 : Schéma général du calcul des performances pour la compression

- **Calcul des performances pour la décompression**

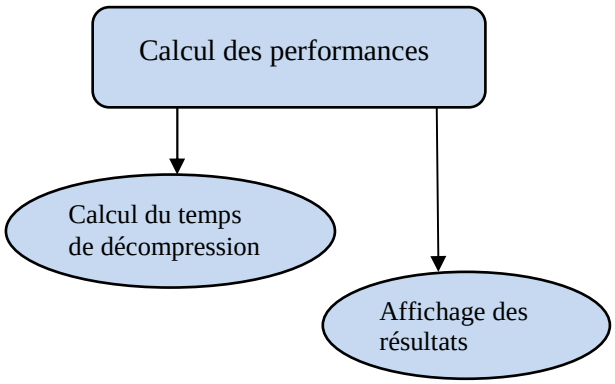


Figure IV.5: Schéma général du calcul des performances pour la décompression

4. 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 de la méthode de compression adaptative. Nous avons considéré cette méthode comme étant un problème (noyau) et nous avons décomposé en sous problème (module) et chaque module en sous module, tout ça accompagné par des schémas. Dans le chapitre qui suit nous allons faire l'implémentation de notre application en se basant sur cette conception.

1. introduction

Dans ce chapitre nous allons présenter la mise en œuvre de la méthode et ses variantes proposées, ainsi que les résultats des différents tests effectués dans le but d'évaluer les performances.

On effectue les tests de cette méthode de compression sur trois axes importants :

- Axes taux de compression : ces tests ont pour objectif d'évaluer les performances de la méthode de point de vue du taux de compression.
- Axes temps de compression : ces tests ont pour objectif d'évaluer les performances de la méthode de point de vue du temps de compression.
- Axes type de fichier : ces tests ont pour objectif d'évaluer l'utilité de la méthode pour des types de fichiers différents.

2. Environnement de développement.

Le système d'exploitation choisi pour la réalisation de notre application est le système Windows. Ces systèmes sont très utilisés pour leurs simplicités, efficacités et fiabilités.

Nous avons choisi le langage C++ pour écrire notre programme, ce choix de langage est motivé par les raisons suivantes :

- Les compilateurs C++ sont actuellement implémentés sur toutes les plates-formes, ce qui fait du langage C++ un outil de programmation très répandu.
- Le code généré par le compilateur C++ est très optimisé, ce qui rend les exécutables plus compactes et plus rapides.
- La plupart des implémentations des algorithmes standards sont implémentés à base de langage C++.

2.1. Description du langage C++ :

Apparu au début des années 90, le langage C++ est actuellement l'un des plus utilisé dans le monde, aussi bien pour les applications scientifiques que pour le des développements des logiciels. En tant qu'héritier du langage C, le C++ est d'une grande efficacité. Mais il a en plus des fonctionnalités puissantes, comme par exemple la notion de classe, qui permet d'appliquer les techniques de la programmation-objet.

Un programme écrit en C++ se compose généralement de plusieurs fichiers-sources. Il y a deux sortes de fichiers-sources :

- ceux qui contiennent effectivement des instructions, leur nom possède l'extension .cpp.
- ceux qui ne contiennent que des déclarations ; leur nom possède l'extension .h (signifiant "header" ou en-tête).

Le langage C++ constitue en soi une extension du langage C qui apporte de réels avantages :

• contrôle d'erreurs accru grâce à un typage fort des données ;
• facilité d'utilisation des langages objets ;
• grand nombre de fonctionnalités complémentaires ;
• performances du C ;
• facilité de conversion des programmes C en C++ et en particulier, la possibilité d'utiliser toutes les fonctionnalités du langage C.

2.2. Description de Dev-Cpp

Dev_Cpp est un environnement de développement intégré (en abréger EDI), qui gère l'ensemble des fichiers et des taches à effectuer sous la forme d'un projet.

Le développement d'un programme passe par trois phases successives :

1) écriture et enregistrement des différents fichiers-sources,
2) compilation séparée des fichiers .cpp, chacun d'eux donnent un fichier-objet portant le même nom, mais avec l'extension .obj.
3) lien des fichiers-objets (assurée par un programme appelé linker) pour produire un unique fichier-exécutable, portant l'extension .exe ; ce dernier pourra être lancé et exécuté directement depuis le système d'exploitation.

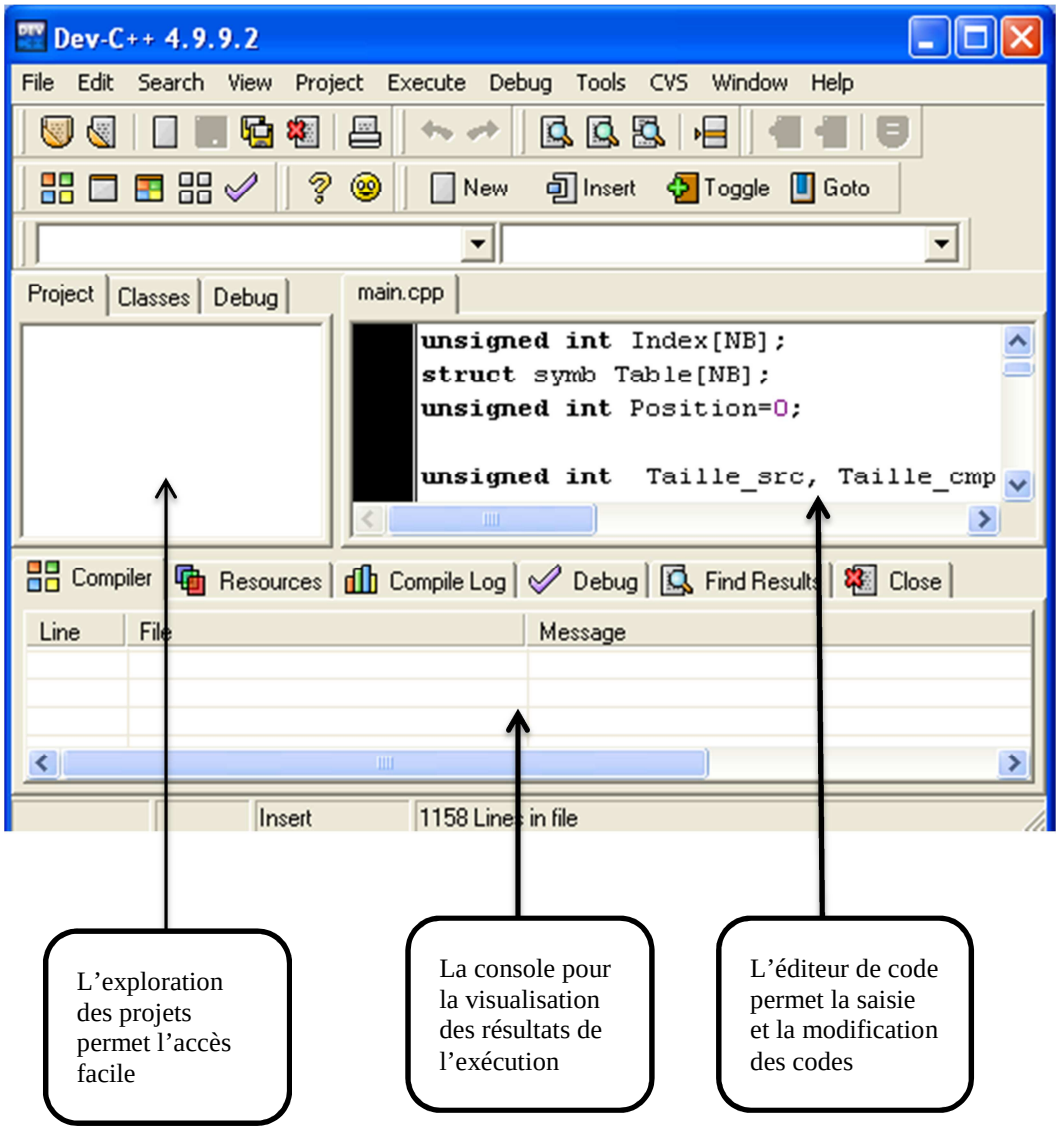


Figure V.1 : l'interface de l'environnement Dev-Cpp.

.

3. Présentation de l'application

Dans ce point, on va expliquer le fonctionnement de notre application, et quelles sont les commandes nécessaires pour faire l'exécution de l'application.

3.1. Le chois de fichier à compressé

Pour compresser un fichier il faut que le fichier et l'exécutable de l'application se trouvent dans le même répertoire. L'exécution de l'application se passe sur l'invite de commande.

3.2. Syntaxe de la commande de compression

Pour lancer la compression en tape la commande suivante :
…/>compresser.exe –co fichier_source fichier_compressé

Exemple
D:\ ajdéco12>main.exe –co bib bibc

Cette commande contient les champs suivants :

- D:\ ajdéco12> : c'est le chemin ou ce trouve l'exécutable de l'application et le fichier qu'on veut compresser.
- Main.exe : c'est le nom de l'exécutable avec l'extension .exe.
- -co : c'est le type de compression (adaptative).
- bib : c'est le nom de fichier source.
- bibc : c'est le nom de fichier compressé.

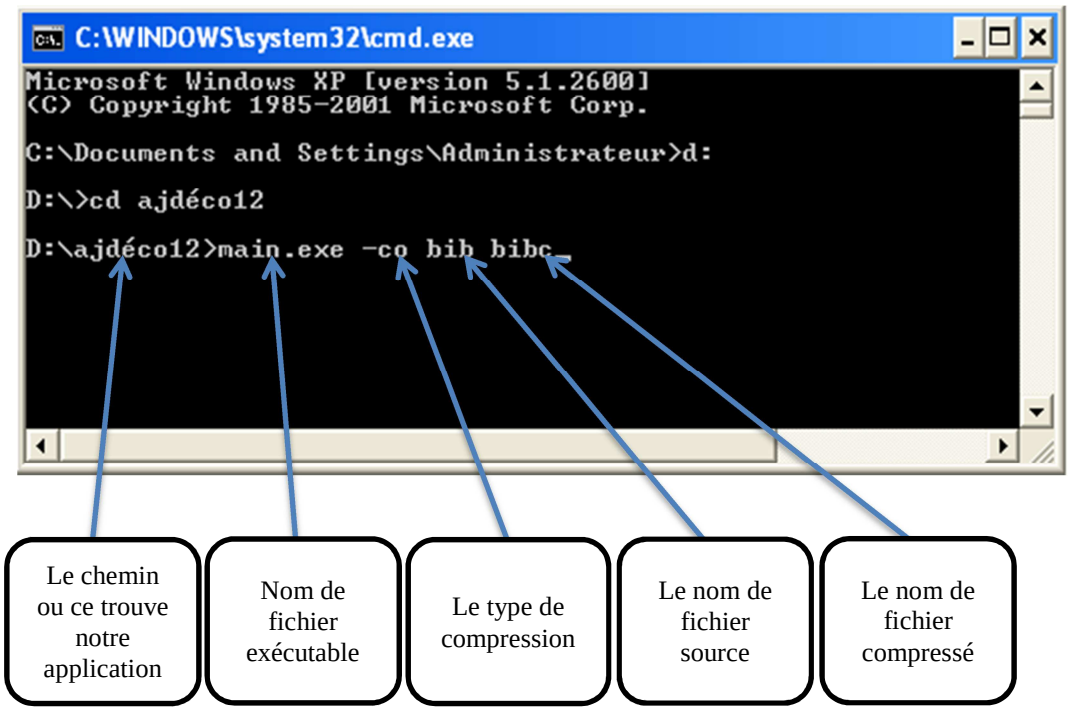


Figure V.2 : Lancement de la compression

3.3. Les résultats de la compression

Après le lancement de la compression notre application commence son travail qui consiste à compresser le fichier donné, à la fin nous aurons les résultats suivants :

- Le nombre de caractères de fichier source.
- Nombre de caractères de fichier compressé.
- Le taux de compression.
- Le temps de compression.

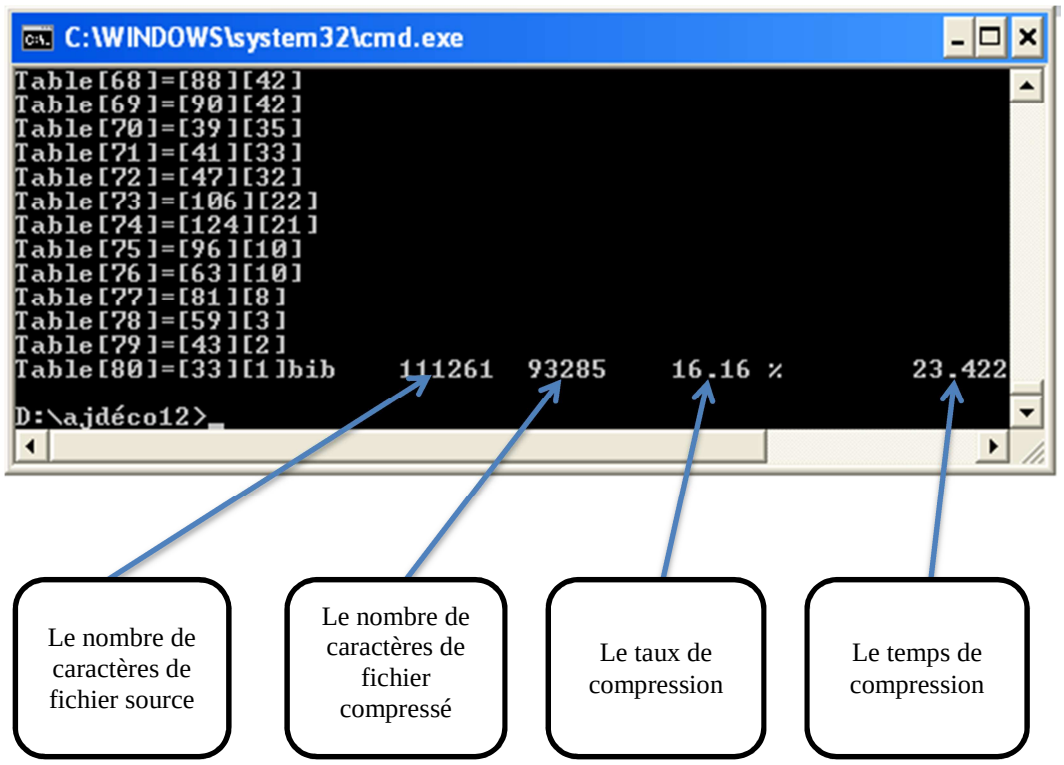


Figure V.3 : Les résultats de la compression

3.4. Syntaxe de la commande de décompression

Avant de lancer la décompression il faut donner un fichier déjà compressé. La commande qui nous permet de lancer la compression est :

…/>compresser.exe –co fichier_compressé fichier_décompressé.

Exemple

D:\ ajdéco12>main.exe –do bibc bibs.

Cette commande contient les champs suivants :

- D:\ fichier test> : c'est le chemin ou ce trouve l'exécutable de l'application et le fichier qu'on veut décompresser.
- Main.exe : c'est le nom de l'exécutable avec l'extension .exe.
- -do : c'est le type de décompression (adaptative).
- bibc : c'est le nom de fichier déjà compressé.
- bibs: c'est le nom de fichier décompressé.

3.1.5. Les résultats de la décompression

Après le lancement de la décompression notre application commence son travail qui consiste à décompresser le fichier qui est déjà compressé, à la fin nous aurons les résultats suivants :

- Temps de décompression.
- Le fichier décompressé qui est le même que le fichier source.

4. Le noyau de l'application

4.1. La méthode proposée :

Le noyau de cette application est constitué de plusieurs classes. Chaque classe a plusieurs fonctions. Et chaque fonction a un rôle dans l'accomplissement des buts de l'application. Ces fonctions sont :

- unsigned int Get preffix () : cette fonction est utilisée pour récupérer le préfixe d'un code soit dans la compression ou la décompression.
- Unsigned int Get Size Preffix () : cette fonction est utilisée pour récupérer la taille de code préfixe.
- Get Suffix () : le rôle de cette fonction est de retourner le suffixe d'un code donné.
- Get Size Suffix () : le rôle de cette fonction est de récupérer la taille de code suffixe.
- Void InitTabCode () : avant de commencer le codage d'un symbole il faut initialiser la table de code par la fonction InitTabCode ().
- AffTabCode () : A la fin de codage la fonction AffTabCode () nous permet d'afficher le code de chaque symbole.
- Void Ouvrir () : cette fonction est utilisée pour l'ouverture de fichier source, ainsi pour ouvrir un nouveau fichier qui sera comme un fichier compressé.
- WriteSymb () : pour connaître le symbole de chaque code ou caractères nous utilisons la fonction WriteSymb ().

- Unsigned char bit_read () : cette fonction est utilisée pour lire le code des caractères bit par bit jusqu'à la fin de fichier.
- Void bit_write () : la fonction qui nous permet d'écrire le code compressé dans le nouveau ficher compressé c'est la fonction bit_write ().
- Void CoderSymboles : c'est la fonction principale dans la phase de compression, car dans cette fonction on fait plusieurs opération, on calcul la fréquence de chaque symbole en même temps on met à jour la table de fréquence et la table d'indexe. A la fin, on affecte le code à chaque symbole.
- Void DécoderSymboles () : cette fonction a le même principe que la fonction CoderSympoles (), les opérations sont les mêmes : calcul de fréquence mettre à jour la table de fréquence et la table d'indexe, à la fin nous retournons les caractères de chaque codes.

5. Evaluation de l'application

Dans l'évaluation on va tester notre application sur plusieurs fichiers textes et images, on va discuter sur les résultats de taux et le temps de compression par apport à la taille de fichier source.

5.1. Les fichiers textes

nom de fichier	**taille Av_comp**	**taille AP_comp**	**taux de comp**	**temps de comp**
obj1	21 ko	20 ko	6,00%	0,328s
paper1	52 ko	43 ko	18,31%	0,062s
geo	100 ko	90 ko	10,95%	0,235s
news	369 ko	306 ko	17,04%	0,266 s
pic	502 ko	312 ko	37,84%	0,468s
book1	751 ko	518 ko	22,24%	0,484s

Tableau V.1 : Evaluation du taux et du temps de compression pour les fichiers textes

Discutions :

On remarque que le taux de compression est meilleur lorsque la taille de fichier source est plus grande, par ce que dans les plus grandes tailles on trouve beaucoup de redondances. Alors notre application va éliminer cette redondance donc on gagne beaucoup d'espace mémoire.

Le temps augmente chaque fois que la taille de fichier augmente.

5.2. Les Fichiers images

nom de fichier	taille Av_comp	taille AP_comp	taux de comp	temps de comp
leg_0_t.tif	1,64 Mo	1,54 Mo	6,24%	2,256 s
leg_1_t.tif	1,25 Mo	1,30 Mo	-3,70%	1,281 s
spine_t.tif	1,70 Mo	1,87 Mo	-10,39%	1,953s

Tableau V.2 : Evaluation du taux et du temps pour les fichiers images

Discutions :

On remarque que le taux et le temps diminue avec l'augmentation de la taille fichier image.

Conclusion :

On conclu que la compression avec la méthode VLC adaptative est faite pour les fichiers textes de grandes tailles (on a un meilleur taux). Par contre elle n'est pas adaptative pour les fichiers images (on un taux faible).

6. Conclusion :

Dans ce chapitre d'implémentation et évaluation nous avons présenté au premier lieu l'environnement de notre travail, c'est-à-dire le logiciel et le langage de programmation utilisés.

Dans la deuxième partie, nous avons donné les résultats obtenus après les tests effectués pour la méthode VLC adaptative, le taux et le temps par rapport au type des fichiers et aussi à la taille des fichiers.

Conclusion générale

La compression des données est appelée à prendre un rôle encore plus important en raison de 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 les réseaux, 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).

Notre travail avait comme objectif de rendre la méthode VLC adaptative pour rependre à l'aspect temps réel des utilisateurs. Cette méthode existe déjà mais moins efficace à cause de ces deux phases qui prennent un temps important (un pour le calcul des fréquences et l'autre pour le codage). Avec la méthode VLC adaptative, on a réduit les deux phases à une seule phase, alors cette dernière est plus efficace et elle répond au besoin du temps réel.

A la fin de notre application nous avons fait des tests sur des fichiers textes et des images et on remarque la méthode VLC adaptative nous donne des bon résultats pour le taux et le temps de compression.

Ce travail m'a donné l'occasion d'apprendre beaucoup de choses dans le domaine de la compression des données ainsi sur les différents algorithmes utilisé pour la compression et de mieux me familiariser avec le langage C^{++}et de l'outil de développement Dev-Cpp

Références bibliographiques

[9] : Alexandre Thil, initiation à la Recherche Algorithmes de compression de données et de traitement d'images. **Article**

[3] : MM. S. Maadi, Y. Peneveyre, et C. Lambercy, Compression de données sans pertes. **Article**

[8]: Olivier RIOUL, Codage entropique à longueur variable, **Article 2003.**

[10]: Peter Fenwick, Punctured Elias Codes for variable-length coding of the integers, **Article December 5, 1996.**

[4]: PEREIRA Vincent-LEPETTE Franck-HACAULT Vincent, Compression de données, **Article Décembre 2004.**

[5] : Cour de Master II : IASIG Dr. Mvogo Ngono Joseph, Principes généraux de codage entropique d'une source Compression d'images.

[6] : Gholam-Reza Mohammad-Khani, Chang-Ming Lee, Mchel Kieffer et Pierre Duhamel,Treillis à complexité réduite pour le décodage de codes à longueur variable , 5 Février 2007.

[1] : Henri Nussbaumer, Téléinformatique III ; Session, Présentation, Compression des données, couche application.

[7] : Mémoire d'ingénieur, Implémentation et évaluation d'une nouvelle méthode de compression de données basée sur le code VLC, promotion 2009/2010.

[2] : Guillaume BOISSON, Représentation hautement scalables pour la compression vidéo sur une large gamme de débits/résolution, **Thèse 2005.**