
Cours Compression
Master M2 Année 2009-2010
Version 0.0

Université de Montpellier
Place Eugène Bataillon
34095 Montpellier Cedex 5

RODOLPHE GIROUDEAU
161, RUE ADA
34392 MONTPELLIER CEDEX 5
MAIL : {*rgirou*}@LIRMM.FR

Table des matières

1	Compression de données	1
1.1	Introduction	2
1.2	Théorie de l'information	2
1.2.1	Longueur moyenne d'un code	2
1.2.2	L'entropie comme mesure de la quantité d'information	2
1.2.3	Théorème de Shannon	3
1.3	Preliminaires	4
1.3.1	Décoder sans ambiguïté	4
1.3.2	Préfixe et propriétés	5
1.3.3	Théorèmes de Kraft et McMillan	7
1.4	Codage statique	11
1.4.1	Algorithme de Huffman	11
1.4.2	Codage Arithmétique	15
1.4.3	Codes adaptatifs	15
1.5	Heuristiques de réduction d'entropie	23
1.5.1	Run Length Encoding : RLE	23
1.5.2	Move-to-Front	24
1.5.3	BWT : Transformation de Burrows-Wheeler	25
1.6	Codes compresseurs usuels	27
1.6.1	Algorithme de Lempel-Ziv 77	27
1.6.2	Algorithme de Lempel-Ziv 78	32
1.7	Conclusion	34
2	Compression d'images	35
2.1	Introduction	36
2.2	Codes compresseurs usuels	36
2.2.1	Formats GIF et PNG	36
2.3	Compression avec perte	36
2.3.1	Dégradation de l'information	36

2.3.2	Transformation des informations audiovisuelles	37
2.3.3	Le format JPEG	38
2.3.4	Le format MPEG	40
2.4	Conclusion	42
3	Compression du son	43
3.1	Introduction	44
3.2	Conclusion	44

Table des figures

1.1	<i>Arbre dégénéré</i>	8
1.2	<i>Complétion de l'arbre</i>	9
1.3	<i>Exemple de construction d'un code de Huffman</i>	12
1.4	<i>Exemple d'ordre de Gallager sur un arbre</i>	17
1.5	<i>Illustration de la mise à jour de l'arbre</i>	20
1.6	<i>Illustration de l'ajout d'une feuille de caractère x</i>	21
2.1	<i>Balayage en zigzag de JPEG</i>	40
2.2	<i>Compression JPEG</i>	40
2.3	<i>Compression MPEG-1</i>	41
2.4	<i>Compression du son MP3</i>	42

Liste des Algorithmes

1.1	Description de l'algorithme de Huffman	11
1.2	Algorithme de compression Huffman dynamique	16
1.3	Mise à jour (x, T)	19
1.4	Ajout d'une feuille à (x, T)	21
1.5	Algorithme de décompression pour Huffman dynamique	22
1.6	Méthode RLE	24
1.7	Réciproque de la BWT	26
1.8	Algorithme de compression pour LZ77	28
1.9	Algorithme de décompression pour LZ77	31
1.10	Algorithme de compression LZ78	33
1.11	Algorithme de décompression LZ78	34

Sommaire

1.1	Introduction	2
1.2	Théorie de l'information	2
1.2.1	Longueur moyenne d'un code	2
1.2.2	L'entropie comme mesure de la quantité d'information	2
1.2.3	Théorème de Shannon	3
1.3	Préliminaires	4
1.3.1	Décoder sans ambiguïté	4
1.3.2	Préfixe et propriétés	5
1.3.3	Théorèmes de Kraft et McMillan	7
1.4	Codage statique	11
1.4.1	Algorithme de Huffman	11
1.4.2	Codage Arithmétique	15
1.4.3	Codes adaptatifs	15
1.5	Heuristiques de réduction d'entropie	23
1.5.1	Run Length Encoding : RLE	23
1.5.2	Move-to-Front	24
1.5.3	BWT : Transformation de Burrows-Wheeler	25
1.6	Codes compresseurs usuels	27
1.6.1	Algorithme de Lempel-Ziv 77	27
1.6.2	Algorithme de Lempel-Ziv 78	32
1.7	Conclusion	34

Résumé

Nous nous intéressons au problème de la compression de données et des techniques pour résoudre ce problème.

1.1 Introduction

1.2 Théorie de l'information

1.2.1 Longueur moyenne d'un code

Tous les mots de code n'étant pas toujours de la même longueur, on utilise une mesure dépendant des fréquences d'apparition pour estimer la longueur des messages qui coderont une source. On rappelle qu'une source d'information est constituée d'un alphabet S et d'une distribution de probabilités P sur S . Pour un symbole s_i d'une source $\mathcal{S} = (S, P)$, $P(s_i)$ est la probabilité d'occurrence de s_i , et $l(m)$ désigne la longueur d'un mot m (de source ou de code).

Soient $\mathcal{S} = (S, P)$ où $S = \{s_1, \dots, s_n\}$, et C un code de \mathcal{S} dont la fonction de codage est f (C est l'image de S par f). La longueur moyenne du code C est :

$$l(C) = \sum_{i=1}^n l(f(s_i))P(s_i)$$

Exemple

$S = \{a, b, c, d\}$, $P = (1/2; 1/4; 1/8; 1/8)$. Si $C = \{f(a) = 00, f(b) = 01, f(c) = 10, f(d) = 11\}$, la longueur moyenne du schéma est 2.

1.2.2 L'entropie comme mesure de la quantité d'information

Nous arrivons aux notions fondamentales de la théorie de l'information. Soit une source $\mathcal{S} = (S, P)$. Nous ne connaissons de cette source qu'une distribution de probabilité, et nous cherchons à mesurer quantitativement à quel point nous ignorons le comportement de \mathcal{S} .

Il est ainsi naturel de choisir l'entropie pour mesure, à savoir, pour une source (S, P) , $P = (p_1, \dots, p_n)$

$$H(\mathcal{S}) = \sum_{i=1}^n p_i \log_2 \left(\frac{1}{p_i} \right)$$

C'est une mesure de l'incertitude liée à une loi de probabilités, ce qui est toujours illustré par l'exemple du dé : on considère une variable aléatoire (source) issue du jet d'un dé à n faces. Il y a plus d'incertitude dans le résultat de ce jet si le dé est normal que si le dé est biaisé. Ce qui se traduit par : pour tous les p_1, \dots, p_n , $H(p_1, \dots, p_n) \leq H(1/n, \dots, 1/n) = \log_2 n$.

1.2.3 Théorème de Shannon

Ce théorème fondamental de la théorie de l'information est connu sous le nom de théorème de Shannon ou théorème du codage sans bruit.

Nous commençons par énoncer le théorème dans le cas d'une source sans mémoire.

Théorème 1.2.1 *Soit une source S sans mémoire d'entropie H . Tout code univoquement déchiffirable de S^k sur un alphabet de taille q , de longueur moyenne l , vérifie :*

$$l \geq \frac{H}{\log_2 q}$$

De plus, il existe un code univoquement déchiffirable de S^k sur un alphabet de taille q , de longueur moyenne l_k , qui vérifie :

$$l < \frac{H}{\log_2 q} + 1$$

On en déduit pour la $k^{\text{ème}}$ extension de S .

Théorème 1.2.2 *Soit une source S sans mémoire d'entropie H . Tout code univoquement déchiffrables de S^k sur un alphabet de taille q , de longueur moyenne l_k , vérifie :*

$$\frac{l_k}{k} \geq \frac{H}{\log_2 q}$$

De plus, il existe un code univoquement déchiffirable de S^k sur un alphabet de taille q , de longueur moyenne l_k , qui vérifie :

$$\frac{l_k}{k} < \frac{H}{\log_2 q} + \frac{1}{k}$$

Preuve

Sachant que $H(S^k) = k \times H(S)$. □

Pour une source stationnaire quelconque, le théorème peut s'énoncer :

Théorème 1.2.3 *Pour toute source stationnaire d'entropie H , il existe un procédé de codage univoquement déchiffirable sur un alphabet de taille q , et de longueur moyenne l , aussi proch que l'in veut de sa borne inférieure $H/\log_2(q)$.*

En théorie, il est donc possible de trouver un code s'approchant indéfiniment de l'entropie. En pratique pourtant, si le procédé de codage consiste à coder des mots d'une extension de la source, on est limité évidemment par le nombre de ces mots ($|S^k| = |S|^k$), ce qui peut représenter un très grand nombre de mots).

1.3 Préliminaires

1.3.1 Décoder sans ambiguïté

La première des qualité que doit avoir un code est de pouvoir être décodé. C'est une évidence, mais par forcément un problème triviale. Supposons que le code est une fonction bijective, qui transforme le message composé par l'émetteur en message qui est transmis par le canal. Pour un message source $a_1 \dots a_n$, chaîne sur un alphabet source quelconque, et pour un alphabet de code V , appelons f la fonction de codage. On a laors le message codé $c_1 \dots c_n = f(a_1) \dots f(a_n)$, avec $c_i \in V^+, \forall i$. Le code, en tant qu'ensemble des mots de codes, est alors l'image de la fonction de codage f . Le fait que f soit bijective ne suffit cependant pas pour que le message puisse être décodé sans ambiguïté par le récepteur. Prenons l'exemple du codage des lettres de l'alphabet $S = \{A, \dots, Z\}$ par les entiers $C = \{0, \dots, 25\}$ écrits en base 10.

$$f(A) = 0, f(B) = 1, \dots, f(J) = 9, f(K) = 10, f(L) = 11, \dots, f(Z) = 25$$

Le mot de code 1209 peut alors correspondre à différents messages : par exemple, BUJ ou MAJ ou $BCAJ$.

Il est donc fondamental d'ajouter des contraintes sur le code pour qu'un message quelconque puisse être déchiffré sans ambiguïté. Un code C sur un alphabet V est dit non ambigu (on dit parfois univoquement déchiffirable), si $\forall x = x_1 \dots x_n \in V^+$, il existe au plus une séquence $c = c_1 \dots c_m \in C^+$ telle que

$$c_1 \dots c_m = x_1 \dots x_n$$

Théorème 1.3.1 *Un code C sur un alphabet V est non ambigu si et seulement si pour toutes séquences $c = c_1 \dots c_n$ et $d = d_1 \dots d_m$ de C^+ .*

$$c = d \Rightarrow (n = m \text{ et } c_i = d_i, \forall i = 1 \dots n)$$

Exemple : Sur l'alphabet $V = \{0, 1\}$

– le code $C = \{0, 01, 001\}$ n'est pas univoquement déchiffirable.

- le code $C = \{01, 10\}$ est uniquement déchiffable.
- le code $C = \{0, 10, 110\}$ est uniquement déchiffable.

La contrainte de déchiffabilité implique une longueur minimale aux mots de code. Le théorème de Kraft donne une condition nécessaire et suffisante sur la longueur des mots de code pour assurer l'existence d'un code uniquement déchiffable.

1.3.2 Préfixe et propriétés

Définition 1.3.1 On dit qu'un code C sur un alphabet V a la propriété du préfixe (on dit aussi instantané, ou irréductible) si et seulement si pour tout couple de mots distincts (c_1, c_2) , c_2 n'est pas un préfixe de c_1 .

Exemple : $a = 1011000$, $b = 01$ et $c = 1010$, b n'est pas un préfixe de a mais c est un préfixe de a .

En utilisant la propriété du préfixe, nous pouvons déchiffrer les mots d'un tel code dès la fin de la réception du mot (instantanéité), ce qui n'est toujours possible pour des codes uniquement déchiffables : par exemple, si $V = 0, 01, 11$ et si on reçoit le message $m = 0011111111 \dots$ il faut attendre l'occurrence suivante d'un 0 pour pouvoir déchiffrer le second mot 0 ou 01 ?).

Théorème 1.3.2 Tout code possédant la propriété du préfixe est uniquement déchiffable.

Preuve

Soit un code C sur V qui n'est pas uniquement déchiffable. Alors il existe une chaîne $a \in V^n$ telle que $a = c_1 \dots c_l = d_1 \dots d_k$, les c_i et d_i étant des mots de C et $c_i \neq d_i$ pour au moins un i . Choisissons le plus petit i tel que $c_i \neq d_i$ (pour tout $j < i$, $c_j = d_j$). Alors $l(c_i) \neq l(d_i)$, sinon vu le choix de i , on aurait $c_i = d_i$, ce qui est en contradiction avec la définition de i . Si $l(c_i) < l(d_i)$, c_i est un préfixe de d_i et dans le cas contraire d_i est un préfixe de c_i . C n'a donc pas la propriété du préfixe. \square

La réciproque est fautive : le code $C = \{0, 01\}$ est uniquement déchiffable, mais en possède pas la propriété du préfixe. Le théorème suivant est évident, mais assure la déchiffabilité d'une catégorie de code très usité.

Théorème 1.3.3 Tout code dont les mots sont de même longueur possède la propriété du préfixe.

L'arbre de Huffman est un objet permettant de représenter facilement tous les codes

qui ont la propriété du préfixe, et cette représentation facilite grandement leur manipulation.

Définition 1.3.2 On appelle arbre de Huffman un arbre binaire (il peut être facilement étendu à un arbre q -naire) tel que tout sous-arbre a soit 0 soit 2 fils (il est localement complet). Dans ce dernier cas, on assigne le symbole 1 à l'arête reliant la racine locale au fils gauche et 0 au fils droit. A chaque feuille d'un arbre de Huffman, on peut associer un mot de $\{0, 1\}^+$, c'est la chaîne des symboles marquant les arêtes d'un chemin depuis la racine jusqu'à la feuille. Le maximum sur l'ensemble des longueurs des mots d'un arbre de Huffman est appelé hauteur de cet arbre. On appelle code de Huffman l'ensemble des mots correspondant aux chemins d'un arbre de Huffman ; la hauteur de cet arbre est appelée aussi hauteur du code C .

L'introduction des arbres de Huffman est justifiée par les deux théorèmes suivants, qui assurent que tous les codes instantanés pourront être manipulés avec de tels arbres.

Théorème 1.3.4 Un code de Huffman possède la propriété du préfixe.

Preuve Si un mot de code c_1 est un préfixe de c_2 , alors le chemin représentant c_1 dans l'arbre de Huffman est inclus dans le chemin représentant c_2 . Comme c_1 et c_2 sont, par définition associés à des feuilles de l'arbre $c_1 = c_2$. Il n'existe donc pas deux mots différents dont l'un est le préfixe de l'autre, et le code de Huffman admet la propriété du préfixe. \square

Théorème 1.3.5 Tout code qui possède la propriété du préfixe est contenu dans un code de Huffman.

Preuve Soit un arbre de Huffman complet (toutes les feuilles sont à distance constante de la racine) de hauteur l , la longueur du plus long mot de C . Chaque mot c_i de C est associé à un chemin depuis la racine jusqu'à un noeud. On peut élaguer le sous-arbre dont ce noeud est racine (tous les mots pouvant être représentés dans les noeuds de ce sous-arbre ont c_i pour préfixe). Tous les mots de C sont toujours dans les noeuds de l'arbre résultant. On peut effectuer la même opération pour tous les mots. On a finalement un arbre de Huffman contenant tous les mots de C . \square

Lemme 1.3.1 Un code est préfixe si et seulement si les feuilles de son arbre¹ sont exactement ses mots de code.

¹ Les codes peuvent être représentés par des arbres tel que :
 – chaque branche a pour attribut un des symboles binaires 0 ou 1,
 – deux branches partant du même noeud ont des attributs différents,

Preuve Dire qu'un mot de code est une feuille est équivalent à dire qu'il n'est pas le préfixe d'un autre mot de code \square

1.3.3 Théorèmes de Kraft et McMillan

Il existe des conditions nécessaires et suffisantes pour l'existence de codes préfixes et déchiffrables. Pour les codes préfixes, il s'agit d'une propriété classique des arbres (inégalité de Kraft). Le résultat reste vrai pour un code déchiffrable (Théorème de McMillan).

Théorème 1.3.6 Soit un arbre binaire dont les K feuilles ont pour ordre n_1, \dots, n_K (l'ordre d'un noeud ou d'une feuille correspond au nombre de branches le séparant de la racine). Alors,

$$\sum_{k=1}^K 2^{-n_k} \leq 1$$

Preuve

Nous définissons le poids de chaque noeud de l'arbre de la façon suivante :

- le poids d'une feuille d'ordre i est égale à 2^{-i} ,
 - le poids d'un noeud qui n'est pas une feuille est la somme des poids de ses fils.
1. Le poids d'un noeud d'ordre i est inférieur ou égal à 2^{-i} . Sinon, l'un de ses fils (d'ordre $i + 1$) aurait un poids $> 2^{-(i+1)}$, et par récurrence, il existerait au moins une feuille de poids $> 2^h$, où h est l'ordre de cette feuille. Ceci serait contradictoire avec la définition d poids.
 2. le poids de la racine de l'arbre est égale à la somme des poids de ses feuilles.

La racine de l'arbre est un noeud d'ordre 0 et a donc un poids ≤ 1 , ce qui nous donne l'igalité. recherchée. \square

Lemme 1.3.2 Soit un arbre binaire dont les K feuilles ont pour ordre n_1, \dots, n_K avec $m = \max(n_1, \dots, n_K)$. Alors

$$(1 - \sum_{k=1}^K 2^{-n_k} > 0) \Rightarrow (1 - \sum_{k=1}^K 2^{-n_k} \geq 2^{-m})$$

- chaque noeuda pour attribut la concatenation des attributs des branches reliant la racine à ce noeud.

Preuve En reprenant les notations de l'énoncé

$$(1 - \sum_{k=1}^K 2^{-n_k} > 0) \Rightarrow (2^m - \sum_{k=1}^K 2^{m-n_k} > 0)$$

Puisque $\forall k, m \geq n_k$, le terme de gauche de l'inégalité de droite est entier, donc

$$(1 - \sum_{k=1}^K 2^{-n_k} > 0) \Rightarrow (2^m - \sum_{k=1}^K 2^{m-n_k} \geq 1) \Rightarrow (1 - \sum_{k=1}^K 2^{-n_k} \geq 2^{-m}).$$

\square

Théorème 1.3.7 Inégalité de Kraft Il existe un code préfixe de K mots de longueurs n_1, \dots, n_K si et seulement si l'inégalité

$$\sum_{k=1}^K 2^{-n_k} \leq 1$$

est satisfaite

Preuve

1. Soit un code préfixe de K mots de longueur n_1, \dots, n_K , les mots de ce code sont exactement les feuilles de son arbre. D'autre part, la longueur d'un mot est exactement égale à l'ordre du noeud le représentant dans l'arbre. Donc en utilisant le Lemme 1.3.1, nous obtenons bien l'inégalité.
2. Soient des entiers $1 \leq n_1 \leq \dots, n_{K-1} \leq n_K$ vérifiant l'inégalité, montrons par récurrence sur K qu'il existe un arbre binaire dont les K feuilles ont pour ordre n_1, \dots, n_K .
 - (a) Si $K = 1$, l'arbre dégénéré donné à la figure 1.1 de profondeur n_1 convient

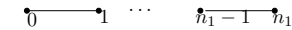


FIG. 1.1 – Arbre dégénéré

(b) En enlevant la dernière longueur n_K , nous avons

$$\sum_{k=1}^{K-1} 2^{-n_k} < 1$$

donc d'après le Lemme 1.3.2,

$$\sum_{k=1}^{K-1} 2^{-n_k} + 2^{-n_{K-1}} = \sum_{k=1}^{K-2} 2^{-n_k} + 2^{1-n_{K-1}} \leq 1$$

On en déduit par hypothèse de récurrence, qu'il existe un arbre dont les feuilles ont pour profondeurs $n_1, \dots, n_{K-2}, n_{K-1} - 1$. Nous dotons la feuille à profondeur $n_{K-1} - 1$ de deux fils pour obtenir un arbre à K de profondeur $n_1, \dots, n_{K-2}, n_{K-1}, n_K$. Si $n_K > n_{K-1}$, il suffit de rallonger l'un des fils. D'après le Lemme 1.3.1

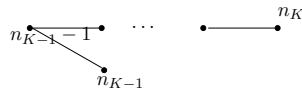


FIG. 1.2 – Complétion de l'arbre

□

Il existe un résultat similaire plus fort dû à McMillan.

Théorème 1.3.8 McMillan *Il existe un code déchiffrable de K mots de longueurs n_1, \dots, n_K si et seulement si l'inégalité*

$$\sum_{k=1}^K 2^{-n_k} \leq 1$$

est satisfaite.

Preuve

1. Si l'inégalité est satisfaite alors il existe un code préfixe donc un code déchiffrable avec les bonnes longueurs.

2. Réciproquement, soit V avec $|V| = K$, soit ϕ un code déchiffrable de V et soit Φ son codage associé. Pour tout L -uplet $(x_1, \dots, x_n) \in V^L$, nous avons $\Phi(x_1, \dots, x_n) = \phi(x_1) \parallel \dots \parallel \phi(x_L)$ et donc $|\Phi(x_1, \dots, x_n)| = |\phi(x_1)| \parallel \dots \parallel |\phi(x_L)|$. Donc

$$\begin{aligned} \left(\sum_x 2^{-|\phi(x)|} \right)^L &= \sum_{x_1} \dots \sum_{x_L} 2^{-|\phi(x_1)|} \dots 2^{-|\phi(x_L)|} \\ &= \sum_{(x_1, \dots, x_L)} 2^{-|\Phi(x_1, \dots, x_n)|} \\ &= \sum_{i=1}^{mL} a_L(i) 2^{-i} \end{aligned}$$

où $m = \max_{x \in V} l(x)$ et $a_L(i)$ est le nombre de L -uplets de V^L dont le codage est de longueur i exactement. Le code ϕ est déchiffrable, donc pour tout i , nous avons $a_L(i) \leq 2$ (sinon il existerait au moins deux L -uplets ayant la même image par ϕ). Nous en déduisons que pour tout L

$$\sum_x 2^{-|\phi(x)|} \leq (mL)^{1/L}$$

D'où l'inégalité recherchée quand L tend vers l'infini.

□

IL est important de noter que les Théorèmes 1.3.7 et 1.3.8 ne sont pas constructifs. Ils nous donnent un résultat sur l'existence de codes dont les longueurs des mots de codes vérifient l'inégalité, mais ne prétendent pas que tout code dont les longueurs vérifient l'inégalité est préfixe (ou déchiffrable).

Corollaire 1.3.1 *S'il existe un code uniquement déchiffrable, dont les mots sont de longueur l_1, \dots, l_n , alors il existe un code instantané de mêmes longueurs de mots.*

C'est une conséquence des théorèmes de Kraft et McMillan. Les codes déchiffrable qui ne possèdent pas la propriété du préfixe ne produisent pas de code aux mots plus courts que le codes instantanés, auxquels on peut donc se restreindre pour la compression d'information (leurs propriétés les rendent plus maniables).

Définition 1.3.3 *Un code est dit complet s'il vérifie la relation*

$$\sum_x 2^{-|\phi(x)|} = 1$$

	Code A	Code B	code C
s_1	00	0	0
s_2	01	100	10
s_3	10	110	110
s_4	11	111	11
$\sum_{i=1}^4 2^{-n_i}$	1	7/8	9/8

Les codes A et B sont déchiffrables, le premier étant complet. Le code C n'est pas déchiffrable.

1.4 Codage statique

Les codages statistiques utilisent la fréquence de chaque caractère de la source pour la compression, et en codant les caractères le plus fréquents par des mots plus courts, se positionnent proches de l'entropie.

1.4.1 Algorithme de Huffman

1.4.1.1 L'algorithme de Huffman est optimal

Cette méthode permet de trouver le meilleur schéma d'encodage d'une source sans mémoire $\mathcal{S} = (S, P)$. L'alphabet de codage est V , de taille q . Il est nécessaire à l'optimalité du résultat de vérifier que $q - 1 \mid |S| - 1$ (afin d'obtenir un arbre localement complet). Dans le cas contraire, il est facile de rajouter des symboles à S , de probabilités d'occurrence nulle, jusqu'à ce que $q - 1 \mid |S| - 1$. Les mots de codes associés (les plus longs) ne seront pas utilisés.

Algorithme 1.1 Description de l'algorithme de Huffman

On construit avec l'alphabet source \mathcal{S} un ensemble de noeud isolé auquel on associe les probabilités de P . Soient p_{i_1}, \dots, p_{i_q} les q symboles de plus faibles probabilités. On construit un arbre dont la racine est un nouveau noeud et auquel on associe la probabilité $p_{i_1} + \dots + p_{i_q}$, et dont les branches sont incidentes aux noeuds p_{i_1}, \dots, p_{i_q} .

On recommence ensuite avec les q plus petites valeurs parmi les noeuds du plus haut niveau (les racines), jusqu'à n'obtenir qu'un arbre (à chaque itération, il y a $q - 1$ éléments en moins parmi les noeuds de plus haut niveau), dont les mots de S sont les feuilles, et dont les mots de code associés dans le schéma ainsi construit sont les mots correspondant aux chemins de la racine aux feuilles.

Les étapes successives de l'algorithme sont décrites par la figure 1.3

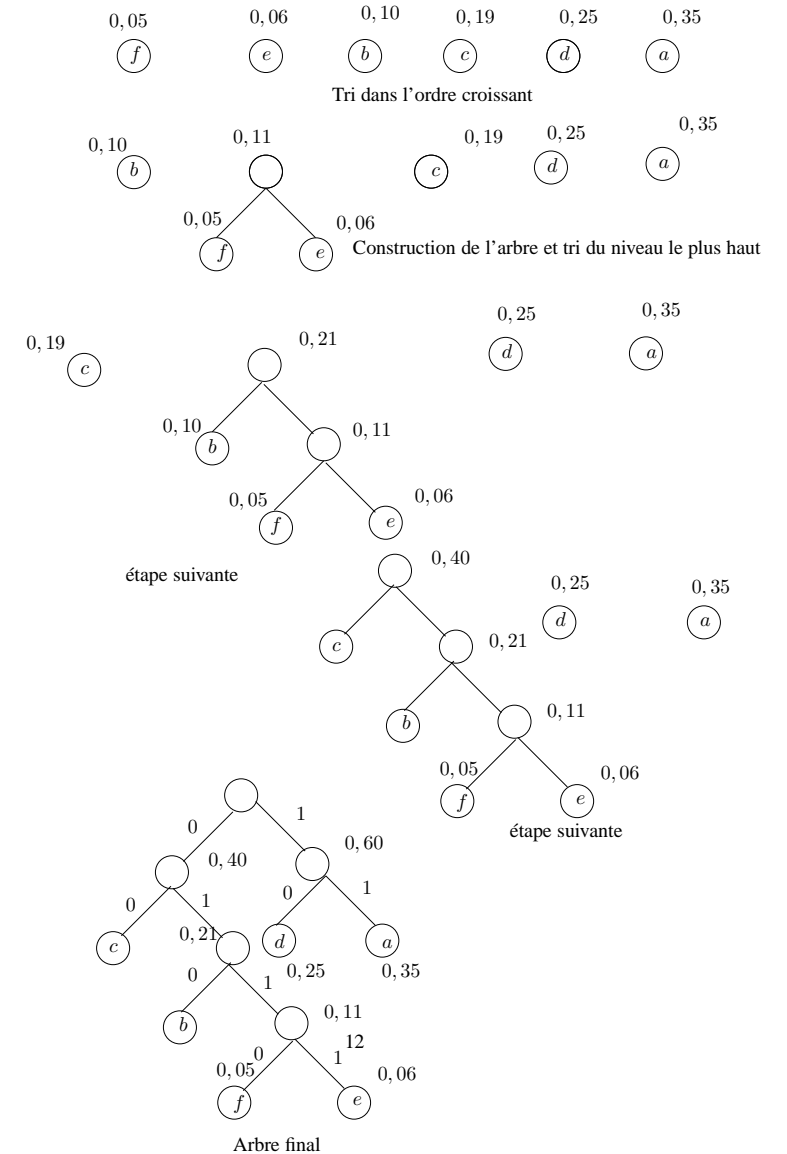


FIG. 1.3 – Exemple de construction d'un code de Huffman

Symbole	Probabilité
<i>a</i>	0,35
<i>b</i>	0,1
<i>c</i>	0,19
<i>d</i>	0,25
<i>e</i>	0,06
<i>f</i>	0,05

TAB. 1.1 – Résultat

Symbole	Mot de code
<i>a</i>	11
<i>b</i>	010
<i>c</i>	00
<i>d</i>	10
<i>e</i>	0111
<i>f</i>	0110

TAB. 1.2 – Résultat

Le code de Huffman construit est alors :

Théorème 1.4.1 *Le code issu de l'algorithme de Huffman est optimal parmi tous les codes instantanés de S sur V .*

Preuve

Prenons $q = 2$. Nos avons qu'un code instantané peut être représenté par un arbre de Huffman. Soit A l'arbre représentant un code optimal, et H l'arbre représentant le code issu de l'algorithme de Huffman.

Remarquons que dans A , il n'existe pas de noeud avec un seul fils dont les feuilles contiennent des mots du code (en effet, on remplace un tel noeud par son fils et on obtient un meilleur code).

Remarquons maintenant que dans A , si pour des mots c_1 et c_2 les probabilités d'apparitions respectives p_1 et p_2 satisfont $p_1 < p_2$, alors les probabilités respectives des feuilles représentant c_1 et c_2 : l_1 et l_2 satisfont $l_1 \geq l_2$ (en effet, dans le cas contraire, on remplace c_1 par c_2 dans l'arbre et on obtient un code meilleur). On peut donc supposer que A représente un code optimal pour lequel les deux mots de plus faibles probabilités sont deux feuilles frères (admettent le même père).

On raisonne maintenant par récurrence sur le nombre de feuilles n dans A . Pour $n = 1$, le résultat est évident.

Pour $n \geq 2$ quelconque, on considère les deux feuilles frères correspondant aux

mots c_1, c_2 de plus faibles probabilités d'apparition p_1, p_2 dans A . D'après le principe de construction de Huffman, c_1 et c_2 sont des feuilles frères dans H . On définit alors $H' = H \setminus \{c_1, c_2\} \cup \{c\}$, c étant un mot de probabilité d'apparition $p_1 + p_2$. H' représente par le principe de récurrence le meilleur code instantané sur C' , donc de longueur moyenne inférieure à $A' = A \setminus \{c_1, c_2\}$. Donc, et d'après les premières remarques, la longueur moyenne de H est inférieure à la longueur moyenne de A . \square

Le théorème précédent indique que l'algorithme de Huffman n'est pas le meilleur pour coder une information dans tous les cas : mais en fixant pour modèle une source S sans mémoire sur un alphabet V , il n'y a pas de code plus efficace qui a la propriété du préfixe.

On peut obtenir des codes plus efficaces à partir des extensions de la source, comme on peut le voir sur l'exemple suivant :

Soit $S = (S, P), S = (s_1, s_2), P = (1/4, 3/4)$. Un codage de Huffman pour S donne évidemment $s_1 \rightarrow 0$ et $s_2 \rightarrow 1$, et sa longueur moyenne est 1.

Un codage de Huffman pour $S^2 = (S^2, P^2)$ donne

$$\begin{aligned} s_1 s_1 &\rightarrow 010 \\ s_1 s_2 &\rightarrow 011 \\ s_2 s_1 &\rightarrow 00 \\ s_2 s_2 &\rightarrow 1 \end{aligned}$$

et sa longueur moyenne est $3 \times (1/16) + 3 \times (3/16) + 2 \times (3/16) + 9/16 = 27/16 = 1,6875$. La longueur moyenne de ce code est donc $l_1, 6875$, et en comparaison avec le code de S (les mots de S^2 sont de longueur 2), $l = 1,6875/2 = 0,84375$ ce qui est meilleur que le code sur la source originelle. Nous pouvons encore améliorer ce codage en examinant la source S^3 . Il est aussi possible d'affiner le codage par une meilleure modélisation de la source : souvent, l'occurrence d'un symbole n'est pas indépendante des symboles précédemment émis par une source (dans le cas d'un texte, par exemple). Dans ce cas, les probabilités d'occurrence sont conditionnelles et il existe des modèles (le modèle de Markov, en particulier) qui permettent un meilleur codage. Mais ces procédés ne conduisent pas à des améliorations infinies. L'entropie reste un seuil pour la longueur moyenne, en-deçà duquel on ne peut pas trouver de code.

1.4.2 Codage Arithmétique

1.4.3 Codes adaptatifs

1.4.3.1 Algorithme de Huffman 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.

La commande « pack » de Unix implémente de cet algorithme.

1.4.3.2 Compression dynamique

La compression est décrite par l'algorithme 1.4.3.2. 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 comme un 257^{ème} caractère virtuel de code ASCII). 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 de 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. Enfin, il ya plusieurs choix pour le nombre d'occurrences de @ ; dans l'algorithme 1.4.3.2 c'est le nombre de caractères distincts (cela permet d'avoir peu de bits pour @ au début), il est aussi possible de lui attribuer par exemple une valeur constante très proche de zéro (dans ce cas le nombre de bits pour @ évolue comme la profondeur de l'arbre de Huffman, en laissant aux caractères très fréquents les codes les plus courts).

Algorithme 1.2 Algorithme de compression Huffman dynamique

```

Soit  $nb(c)$ , le nombre d'occurrence d'un caractère  $c$ 
Initialiser 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  $AH$ 
  end if
   $nb(c) := nb(c) + 1$ 
  Mettre à jour  $AH$  avec les nouvelles fréquences
end while

```

1.4.3.3 Mise en oeuvre de l'algorithme de Huffman adaptatif

Nous identifions un code préfixe d'une source discrète X à un arbre binaire à $|X|$ feuilles étiquetées par les lettres de X . Le code est dit complet si chaque noeuds possède 0 ou 2 fils.

Définition 1.4.1 Soit T un code préfixe d'une source discrète X , le poids relatif à (T, X) est défini de la façon suivante :

- le poids d'une feuille est la probabilité de sa lettre
- le poids d'un noeud est la somme du poids de ses fils

Définition 1.4.2 Un arbre de Huffman de X est un code préfixe de X qui peut être obtenu par l'algorithme de Huffman

Définition 1.4.3 Soit T un code préfixe complet d'une source discrète X de cardinal K . Un ordre de Gallager sur T relativement à X est un ordre u_1, \dots, u_{2K-1} sur les noeuds de T vérifiant

1. les poids des u_i relativement à (T, X) sont décroissants,
2. u_{2i} et u_{2i+1} sont frères dans $T, \forall 1 \leq i < K$.

Théorème 1.4.2 Gallager Un code préfixe T (i.e. un arbre) d'une source X est un arbre de Huffman de X si et seulement si il existe un ordre de Gallager sur T relativement à X .

Preuve

- **Rappel** un arbre binaire irréductible à m feuilles admet $2m - 1$ noeuds. Nous savons que les mots de poids le plus faible sont jumeaux. Nous fusionnons les deux feuilles pour obtenir un arbre avec une feuille de moins, soit $2(m-1) - 1 = 2m - 3$ noeuds. L'arbre obtenu est un arbre de Huffman, qui admet un ordre de Gallager (induction) $u'_1 \geq \dots \geq u'_{2m-3}$. Et le noeud fusionné est quelque part dans la liste. On prend ses deux fils, et on les mets à la fin, ce qui donne un ordre de Gallager. $u'_1 \geq \dots \geq u'_{2m-3} \geq u_{2m-2} \geq u_{2m-1}$.
- Réciproquement, supposons que T admette un ordre de Gallager. Les noeuds sont ordonnés de la sorte

$$u_1 \geq \dots \geq u_{2m-3} \geq u_{2m-2} \geq u_{2m-1}$$

où u_{2m-2} et u_{2m-1} sont les noeuds de poids minimal, qui sont des feuilles frères. Soit T' l'arbre u_1, \dots, u_{2m-3} il admet l'ordre $u_1 \geq \dots \geq u_{2m-3}$. C'est donc un arbre de Huffman (induction). Par l'algorithme de Huffman, l'arbre u_1, \dots, u_{2m-2} est de Huffman. □

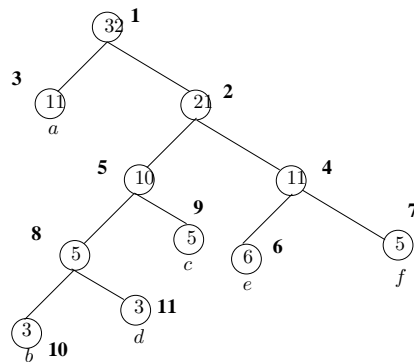


FIG. 1.4 – Exemple d'ordre de Gallager sur un arbre

Nous considérons un texte $(x_1, x_2, \dots, x_n, \dots)$ et une instance de l'algorithme de Huffman adaptatif sur ce texte. Nous notons X_n la source obtenue après lecture de la n -ème lettre et T_n un arbre de Huffman de cette source.

LA probabilité d'une lettre de la source X_n est proportionnelle à son nombre d'occurrence dans les n premières lettres du texte. Ainsi sans rien changer aux définitions (en particulier pour un ordre de Gallager), nous pourrions utiliser pour le poids le nombre d'occurrence de chaque lettre (la lettre vide a un nombre d'occurrences égal à 0). Notons que dans ce cas, le poids est entier et que le poids de la racine de T_n est n . Soit x la $(n + 1)$ -ème lettre du texte. Soit u_1, \dots, u_{2K-1} est la feuille vide.

Lemme 1.4.1 Si $x \in X_n$ et si tous les noeuds $u_{i_1}, u_{i_2}, \dots, u_{i_k}$ du chemin entre x et la racine de T_n sont les premiers de leur poids dans l'ordre de Gallager sur T_n relativement à X_n , alors T_n est un arbre de Huffman de X_{n+1} , et u_1, \dots, u_{2K-1} est un ordre de Gallager relativement à X_{n+1} .

Preuve Si l'on pose $T_{n+1} = T_n$, le poids des noeuds reste inchangé, sauf pour les sommets $u_{i_1}, u_{i_2}, \dots, u_{i_k}$ du chemin de x à la racine. Chacun de ces noeuds étant le premier de son poids dans un ordre de Gallager sur T_n relativement à X_n , l'ordre reste décroissant après l'incrément des poids. LA condition sur les frères est toujours vérifiée, car ni l'arbre ni l'ordre n'ont changé, et nous obtenons bien un ordre de Gallager de T_{n+1} relativement à X_{n+1} . □

Le cas ci-dessus est idéal et n'est pas vérifié en général. Toutefois, quitte à faire quelques échanges de noeuds dans l'arbre, on peut toujours s'y ramener.

Pour décrire les algorithmes nous allons munir les noeuds de trois champs :

- *pere* un noeud
- *poids* et *ordre* des entiers

Initialement, pour tout noeud de T_n

- le champ *pere* contient le père de u dans T_n (nil s'il s'agit de la racine)
- le champ *poids* contient le poids de u relativement à (T_n, X_n)
- le champ *ordre* contient l'indice de u dans un ordre de Gallager sur T_n relativement à X_n

Lemme 1.4.2 Si $x \in X_n$, alors à l'issue de la procédure mise à jour (x, T) , l'arbre T_{n+1} défini par les champs *pere* est un code binaire préfixe de X_{n+1} . Les poids sont relatifs à (T_{n+1}, X_{n+1}) (à un coefficient multiplicatif près), et les champs *ordre* induisent un ordre de Gallager sur T_{n+1} relativement à X_{n+1} .

Preuve Les propriétés suivantes sont des invariants de la boucle :

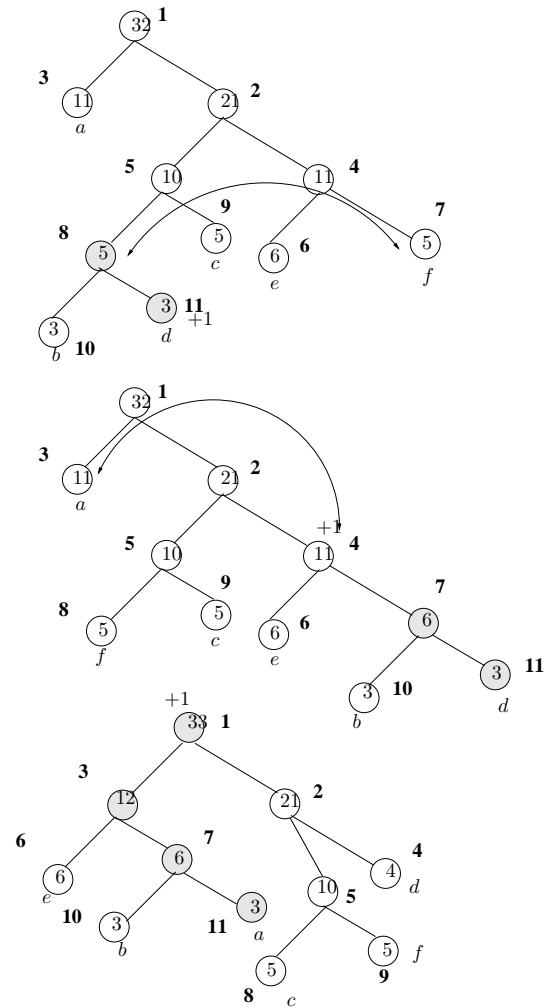
- deux noeuds partageant le même *pere* ont des ordres consécutifs.
- l'ordre des noeuds est celui des poids décroissants.

De plus, puisque nous n'avons procédé qu'à des échanges de noeuds, à la fin de l'exécution, les champs *pere* définissent un arbre binaire complet (que nous notons T_{n+1} possédant autant de feuilles que le cardinal de X_n (égal à celui de

Algorithme 1.3 Mise à jour (x, T)

$u := \text{noeud}(x, T)$; la feuille de T dont la lettre associée est x
while $u \neq \text{nil}$ **do**
 $\tilde{u} := \min\{v \mid v.\text{poids} = U.\text{poids}\}$; le noeud de plus petit ordre ayant le poids de u
echanger($u.\text{pere}, \tilde{u}.\text{pere}$)
echanger($u.\text{ordre}, \tilde{u}.\text{ordre}$)
 $u.\text{poids} := u.\text{poids} + 1$
 $u := u.\text{pere}$
end while

X_{n+1}). Il s'agit donc d'un code préfixe de X_{n+1} . La valeur des poids est relative à (T_{n+1}, X_{n+1}) , car une seule feuille a vu son poids augmenter, celle correspondant à x , les autres noeuds dont le poids a été incrémenté l'ont été conformément à la définition du poids (le chemin de x à la racine). □



20
FIG. 1.5 – Illustration de la mise à jour de l'arbre

Si $x \in X_n$, l'arbre doit augmenter de taille et aura $2K+2$ noeuds au lieu de $2K$. La nouvelle lettre va occuper la place de la lettre vide, puis nous ajoutons une feuille qui deviendra la nouvelle feuille vide.

Algorithme 1.4 Ajout d'une feuille à (x, T)

mise à jour (x, T)
 $z := \text{noeud}(\emptyset, T)$ (la feuille vide de T); z devient un noeud interne, ses deux fils u et v sont créés ci-après
 $u := \text{newfeuille}(x, T)$ (nouvelle feuille associé à x dans T)
 $u.\text{pere} := z; u.\text{poids} := 1; u.\text{ordre} := 2K$
 $v := \text{newfeuille}(\emptyset, T)$ (nouvelle feuille vide de T)
 $v.\text{pere} := z; v.\text{poids} := 1; v.\text{ordre} := 2K + 1$

Lemme 1.4.3 Si $x \notin X_n$, alors à l'issue de la procédure ajouter feuille (x, T_n) , l'arbre T_{n+1} défini par les champs *pere* est un code préfixe de X_{n+1} . Les poids sont relatifs à (T_{n+1}, X_{n+1}) (à un coefficient multiplicatif près), et les champs *ordre* induisent un ordre de Gallager sur T_{n+1} relativement X_{n+1} .

Preuve

Juste après l'appel mise à jour (\emptyset, T) , d'après les résultats précédents, les propriétés sur les ordre et sur les frères dans l'arbre sont vérifiées. L'ajout des feuilles u et v ne change pas ces propriétés, et les poids sont bien les poids relatifs à (T_{n+1}, X_{n+1}) . □

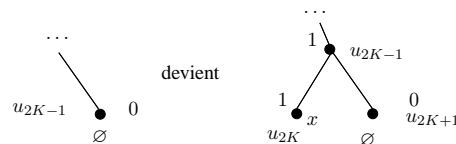


FIG. 1.6 – Illustration de l'ajout d'une feuille de caractère x

L'algorithme de Huffman adaptatif ne pourra être optimal que si la source est sans mémoire. Dans ce cas, conformément au but visé, les lemmes précédents nous permettent de montrer que « localement », le code utilisé est un code de Huffman « classique », et donc est optimal par rapport à ce qui est connu de la source à ce moment là.

1.4.3.4 Décompression dynamique

L'algorithme 1.4.3.4 donne la décompression. À l'initialisation, le décodeur connaît un seul code, celui de @ (par exemple 0). Il lit alors 0 qui est le code associé à @. Il déduit que les sk bits suivants contiennent un nouveau caractère/ Il recopie sur sa sortie csz k bits et met à jour l'arbre de Huffman qui contient déjà @, avec ce nouveau caractère.

Il faut bien noter que le codeur et le décodeur maintiennent chacun leur propre arbre de Huffman, mais utilisent tous les deux le même algorithme pour le mettre à jour à partir des occurrences (fréquences) des caractères déjà lus. Aussi, les arbres de Huffman calculés séparément par le codeur et le décodeur sont les mêmes.

Algorithme 1.5 Algorithme de décompression pour Huffman dynamique

Soit $nb(c)$, le nombre d'occurrence d'un caractère c
Initialiser l'arbre de Huffman (AH) avec le caractère @, $nb(@) := 1$
while on n'est pas à la fin du message **do**
 Lire le mot de code c du message (jusqu'à une feuille de AH)
 if $c = @$ **then**
 $nb(@) := nb(@) + 1$
 Lire dans c les k bits du message et les afficher ne sortie
 $nb(c) := 0$
 else
 Afficher le code de c dans AH
 end if
 $nb(c) := nb(c) + 1$
 Mettre à jour AH avec les nouvelles fréquences
end while

Puis, le décodeur lit le code suivant et le décode via son arbre de Huffman. S'il s'agit du code de @, il lit les k bits correspondant à un nouveau caractère, les écrit sur la sortie et ajoute le nouveau caractère à son arbre de Huffman (le nouveau caractère est désormais associé à un code). Sinon, il s'agit du code d'un caractère déjà rencontré ; via son arbre de Huffman, il trouve les k bits du caractère associé au code, et écrit sur la sortie. Il incrémente alors de un le nombre d'occurrences du caractère qu'il vient d'écrire (et de @ si c'est un nouveau caractère) et met à jour l'arbre de Huffman. Cette méthode dynamique est peu moins efficace que la méthode statique pour estimer les fréquences. Il est probable que le message codé sera donc un peu long. Mais elle évite le stockage de l'arbre et de la table des fréquences, ce qui rend le résultat final souvent plus court. Ceci explique son utilisation en pratique dans les utilitaires courants.

1.5 Heuristiques de réduction d'entropie

En pratique, l'algorithme de Huffman (ou ses variantes) est utilisé en conjonction avec d'autres procédés de codage. Ces autres procédés ne sont pas toujours optimaux en théorie mais ils font des hypothèses raisonnables sur la forme des fichiers à compresser pour diminuer l'entropie ou accepter une destruction d'information supposée sans conséquence pour l'utilisation des données.

Nous présenterons trois exemples de réduction d'entropie. Le principe est de transformer un message en un autre, par une transformation réversible (fonction bijective), de façon à ce que le nouveau message ait une entropie plus faible, et soit donc compressible plus efficacement. Il s'agit donc d'un codage préalable à la compression, chargé de réduire l'entropie.

1.5.1 Run Length Encoding : RLE

Le codage statique tire parti des caractères apparaissant souvent, mais absolument pas leur position dans le texte. Si un même caractère apparaît souvent répété plusieurs fois d'affilée, il peut être utile de coder simplement le nombre de fois où il apparaît. Par exemple, pour transmettre une page par fax, le codage statique codera le 0 par un petit mot de code, mais chaque 0 sera écrit. Le codage *RLE* lit chaque caractère à la suite mais, lorsque au moins 3 caractères identiques successifs sont rencontrés, il affiche plutôt un code spécial de répétition suivi du caractère à répéter et du nombre de répétitions. Par exemple, avec & comme caractère de répétition, la chaîne « aabbbcccc&ee » est codée par « &a2&b3c&d5&&1&e2 » (il est nécessaire d'employer une petite astuce si le caractère spécial est rencontré pour le rendre le code instantané).

Pour ne pas avoir le problème du caractère spécial de répétition qui se code forcément comme une répétition de taille 1 les modems utilisent une variante de RLE appelé MNP5. L'idée est de que lorsque 3 ou plus caractères identiques sont rencontrés, ces trois caractères sont affichés suivis d'un compteur indiquant le nombre d'apparitions supplémentaires du caractère. Il faut bien sûr convenir d'un nombre de bits fixe à attribuer à ce compteur. Si le compteur dépasse son maximum, la suite sera codée par plusieurs blocs des 3 mêmes caractères suivi d'un compteur. Par exemple, la chaîne « aabbbcccc » est codée par « aabb0cccc2 ». Dans ce cas, si une chaîne de n octets contient m répétitions de longueur moyenne L , le gain de compression est $\frac{n-m(L-4)}{n}$ si le compteur est sur un octet. Ce type de codage est donc très utile par exemple pour des images noir et blanc ou des pixels de même couleur sont très souvent accolés. Enfin, il est clair qu'une compression sta-

tistique peut être effectuée ultérieurement sur le résultat d'un RLE. Nous obtenons l'algorithme 1.5.1.

Algorithme 1.6 Méthode RLE

```

repet := 0; compteur := 1
RefCar := entre[compteur]; CarCour := RefCar;
while compteur ≤ longueur(entre) do
  while RefCar = CarCour et compteur ≤ longueur(entre) do
    incrémente(compteur); incrémente(répète);
    CarCour := entre[compteur];
  end while
  if rpte < 4 then
    for i de 1 à rpte do
      écrire(RefCar);
    end for
  else
    écrire(&r,répète,RefCar);
  end if
  rpte := 0; RefCar := CarCour
end while

```

La décompression est immédiate : dès qu'on lit $\&nc$, qui indique n répétition du caractère c , on écrit n fois c .

1.5.2 Move-to-Front

Il est possible de procéder à un pré-calcul lors de la transformation d'un caractère ASCII en sa valeur entre 0 et 255 : en modifiant à la volée l'ordre de la table. Par exemple, dès qu'un caractère apparaît dans la source, il est d'abord codé par sa valeur, puis il passe en début de liste, et sera codé dorénavant par un 0, tous les autres caractères étant décalés d'une unité. Ce « move-to-front » permet d'avoir plus de codes proches de 0 que de 255. Ainsi, l'entropie est modifiée.

Par exemple, la chaîne « aaaafff » peut être modélisée par une source d'entropie 1, si la table est $(a, b, c, d, e, f, \dots)$. Elle est alors codée par « 00005555 ». L'entropie du code est 1. C'est aussi l'entropie de la source. Mais le code d'un Move-To-Front sera « 00005000 », d'entropie $H = 7/8 \log_2(8/7) + \log_2(8)/8 = 0,55$. Le code lui-même est alors compressible, c'est ce qu'on appelle la réduction d'entropie.

Le décodage est simple : à partir du même tableau initial, il suffit d'émettre le caractère correspondant à l'indice et de ranger le tableau en passant ce caractère en premier. Le tableau évolue exactement comme pendant la phase de codage.

D_0	c	o	m	p	r	e	s1	s2	e2
D_1	o	m	p	r	e1	s1	s2	e2	c
D_2	m	p	r	e1	s1	s2	e2	c	o
D_3	p	r	e1	s1	s2	e2	c	o	m
D_4	r	e1	s1	s2	e2	c	o	m	p
D_5	e1	s1	s2	e2	c	o	m	p	r
D_6	s1	s2	e2	c	o	m	p	r	e1
D_7	s2	e2	c	o	m	p	r	e1	s1
D_7	e2	c	o	m	p	r	e1	s1	s2

TAB. 1.3 – BWT sur le mot compressé

1.5.3 BWT : Transformation de Burrows-Wheeler

L'idée de l'heuristique BWT est de trier les caractères d'une chaîne afin que le Move-To-Front et le RLE soient les plus efficaces possibles. Le problème est bien sûr qu'il est impossible de retrouver la chaîne initiale à partir d'un tri de cette chaîne ! L'astuce est donc de trier la chaîne, mais d'envoyer plutôt une chaîne intermédiaire, de meilleure entropie que la chaîne initiale, et qui permettent cependant de retrouver la chaîne initiale. Considérons le mot « compressé ». Il s'agit de créer tous les décalages possibles comme indiqué par le tableau 1.5.3, puis de trier les lignes dans l'ordre alphabétique et lexicographique. La première colonne F de la nouvelle matrice des décalages est donc la chaîne triée de toutes les lettres du mot source. La dernière colonne s'appelle L . Seules la première et la dernière colonne sont écrites, car ce sont les seules importantes pour le code. Sur le tableau on différencie les occurrences d'un même caractère afin de simplifier la vision de la transformation mais n'interviennent pas dans l'algorithme de décodage : en effet l'ordre des caractères est forcément conservé entre L et F .

Bien sûr, pour calculer F (la première colonne) et L (la dernière colonne), il n'est pas nécessaire de stocker toute la matrice des décalages, un simple pointeur se déplaçant dans la chaîne est suffisant. Cette première phase est donc assez simple. Mais si seule la colonne F est envoyée, comment effectuer la réciproque ? La solution est d'envoyer la chaîne L au lieu de la chaîne F : si L n'est pas triée, elle est cependant issue du tri d'une chaîne quasiment semblable, simplement décalée d'une lettre. On peut donc espérer qu'elle conserve des propriétés issues du tri et que l'entropie en sera réduite.

Concernant le décodage, la connaissance de la chaîne, conjuguée à celle d'un index primaire (le numéro de la ligne contenant la chaîne originale, dans l'exemple précédent c est la ligne 4, en numérotant de 0 à 8) permet de récupérer la chaîne initiale. Il

	F	L
D_0	c	e2
D_8	e2	s2
D_5	e1	r
D_2	m	o
D_1	o	c
D_3	p	m
D_4	r	p
D_7	s2	s1
D_1	s1	e1

TAB. 1.4 – Les colonnes F et L après le tri

suffit de trier L pour récupérer F . Ensuite, on calcule un vecteur de transformation H contenant la correspondance des indices entre L et F .

Pour l'exemple, cela donne $H = [4, 0, 8, 5, 3, 6, 2, 1, 7]$, car C est en position 4, l'index primaire dans L , puis E_2 est en position 0 dans L puis E_1 est en position 8, ... Ensuite, il faut se rendre compte en plaçant l et F en colonnes, dans chaque ligne deux lettres se suivant doivent se suivre dans la chaîne initiale : en effet, par décalage, la dernière lettre devient la première et la première devient la deuxième. Ceci se traduit également par le fait que $\forall j, L[H[j]] = F[j]$. Il ne reste plus qu'à suivre cet enchaînement de lettres deux à deux pour retrouver la chaîne initiale, comme dans l'algorithme 1.5.3.

Algorithme 1.7 Réciproque de la BWT

Soit l une chaîne de caractères et soit $index$

$F := \text{tri de } L$;

Calculer le vecteur de transformation H tel que $L[H[j]] = F[j], \forall j$;

for i de 0 à Taille de L **do**

 Afficher $L[index]$;

$index := H[index]$;

end for

En sortie de la transformation BWT, on obtient donc en général une chaîne d'entropie plus faible, bien adaptée à un Move-To-Front suivi d'un RLE. L'utilitaire de compression *gzip* utilise cette suite de réductions d'entropie avant d'effectuer un codage de Huffman.

1.6 Codes compresseurs usuels

Ces algorithmes de compression portent aussi le nom de substitution de facteurs. En effet, leur principe repose sur le remplacement de facteurs de l'entrée par des codes plus courts. Ces codes représentent les indices des facteurs dans un dictionnaire qui est construit dynamiquement, au fur et à mesure de la compression.

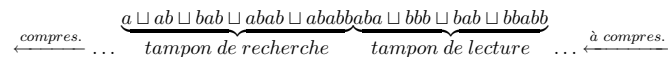
La plupart des programmes de compression dynamique utilisent une des deux méthodes proposées par Lempel et Ziv en 77 et en 78. Ces deux algorithmes parcourent l'entrée à comprimer de la gauche vers la droite. Ils remplacent les facteurs répétées par des pointeurs vers l'endroit où ils sont déjà apparus dans le texte. Bien que ces deux méthodes soient souvent confondues dans la littérature, elles se distinguent grandement par la construction du dictionnaire. Il existe un grand nombre de variantes de ces deux algorithmes. LZ77 est utilisé dans l'utilitaire `gzip` sous UNIX.

1.6.1 Algorithme de Lempel-Ziv 77

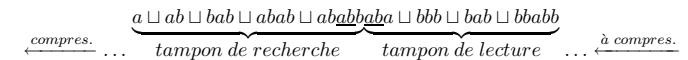
1.6.1.1 La compression

L'idée essentielle de cet algorithme est d'utiliser une partie de la donnée d'entrée comme dictionnaire. L'algorithme de compression fait glisser une fenêtre de N caractère sur la chaîne d'entrée de la gauche vers la droite. Cette fenêtre est composée de deux parties :

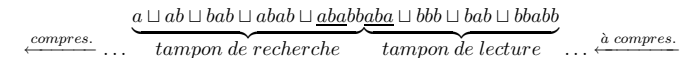
- à gauche le tampon de recherche de $N - F$ caractères qui constitue le dictionnaire courant des lettres qui ont été lues et comprimées récemment ;
- à droite, le tampon de lecture de F caractères dans lequel se trouvent les lettres en attente de compression



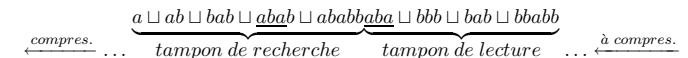
A la lecture du a à gauche du tampon de lecture, l'algorithme de compression parcourt le tampon de recherche de la droite vers la gauche pour trouver un a . Il en trouve un à une distance de 3 (décalage de 3). Il cherche ensuite à faire correspondre le plus de symboles possibles entre le préfixe du tampon de lecture et le facteur du tampon de recherche débutant au a trouvé précédemment. Il peut mettre en correspondance le facteur ab , de longueur 2 avec un décalage de 3.



Il continue ensuite sa recherche pour essayer de faire correspondre le plus de symboles possibles entre le préfixe du tampon de lecture et un facteur du tampon de recherche.



La meilleure correspondance est faite entre le facteur aba et le préfixe aba , avec un décalage de 5, d'une longueur de 3 caractères.



Une correspondance équivalente est trouvée avec un décalage de 10 et une longueur de 3.

L'algorithme choisit la plus longue correspondance et, s'il y en a plusieurs équivalents, celle qui est le plus à gauche (la dernière trouvée) dans le tampon de recherche. Dans notre cas, il sélectionne celle correspondant à un décalage de 10 et une longueur de 3 et il écrit le triplet de sortie indiquant le premier caractère qui n'est pas en correspondance dans le préfixe du tampon de lecture.

Algorithme 1.8 Algorithme de compression pour LZ77

Mettre un pointeur de codage au début de l'entrée

while non vide (tampon de lecture) **do**

Trouver la plus longue correspondance entre le tampon de lecture et celui de recherche

Ecrire (p, l, c) où

- p mesure la distance décalage
- l longueur de la correspondance
- c premier caractère de l'entrée qui n'est pas dans le dictionnaire

Déplacer le pointeur de codage de $l + 1$ positions vers la droite

end while

le mage dit abracadabra	(0, 0, l)
e mage dit abracadabra	(0, 0, e)
mage dit abracadabra	(3, 1, m)
le mage dit abracadabra	(0, 0, a)
le mage dit abracadabra	(0, 0, g)
le mage dit abracadabra	(5, 2, d)
mage dit abracadabra	(0, 0, i)
mage dit abracadabra	(0, 0, t)
mage dit abracadabra	(4, 1, a)
mage dit abracadabra	(0, 0, b)
mage dit abracadabra	(0, 0, r)
mage dit abracadabra	(3, 1, c)
mage dit abracadabra	(5, 1, d)
mage dit abracadabra	(4, 1, b)
mage dit abracadabra	(0, 0, r)
mage dit abracadabra	(5, 1, "r")

TAB. 1.5 – Compression par LZ77

A chaque instant, l’algorithme va rechercher dans les $N - F$ premiers caractères du tampon de recherche le plus long facteur qui se répète au début du tampon de lecture. Il doit être de taille maximale F . Cette répétition sera codée (p, l, c) où

- p est la distance entre le début du tampon de lecture et la position de répétition dans le dictionnaire ;
- l est la longueur de la répétition ;
- c est le premier caractère du tampon de lecture différent du caractère correspondant dans le tampon de recherche.

La répétition peut chevaucher le dictionnaire et le tampon de lecture, comme on le verra plus loin. Après avoir codé la répétition, la fenêtre glisse de $l + 1$ caractères vers la droite. Le codage du caractère c ayant provoqué cette différence est indispensable dans le cas où aucune répétition n’est trouvée dans le dictionnaire. On écrira alors sur la sortie $(0, 0, c)$.

Exemple Nous considérons en entrée le mage dit abracadabra et d’une fenêtre de taille $N = 11$ dont le tampon de lecture est de $F = 5$ caractères. Nous obtenons la suite de triplets engendré par l’algorithme :

Chacun des triplets est codé en binaire en associant un nombre fixe de bits pour chacune des valeurs. Choisir la dernière correspondance trouvée au lieu de l’apre-

mière simplifie le programme. Il n’est pas nécessaire de mémoriser la correspondances précédente. Cependant, le fait de choisir la première correspondance, au prix d’un accroissement de la complexité du programme, présente un avantage. Dans ce cas, le décalage est plus petit. Cela peut sembler inutile car le codage des lexèmes est fait de telle sorte que la longueur $(10, 3, \sqcup)$ et de $(5, 3, \sqcup)$ est identique (les valeurs numériques sont écrites avec une taille fixe paramétrée par la taille du tampon de recherche). Mais si on ajoute un codage de Huffman à l’issue de LZ77, on attribuera aux décalages les plus courts des codes plus courts. Cette méthode proposée par B. Herd porte le nom de LZH et repose sur le principe suivant : si on dispose d’un grand nombre de petits décalages, on améliore la compression en utilisant LZH.

En pratique pour obtenir un codage efficace $N - F = 2^{e_1}$ et $F = 2^{e_2}$. On a donc besoin de e_1 bits pour coder p , la position dans le dictionnaire et de e_2 bits pour coder l , la longueur de répétition.

Plus le répété sera long, plus le codage sera efficace. Pour avoir des chances d’obtenir de longues répétitions, il est nécessaire que le dictionnaire soit de taille suffisante (quelques milliers de caractères).

L’algorithme effectue sa compression au fur et à mesure du déplacement de la fenêtre. Il n’utilise que le dictionnaire compris dans le tampon de recherche et certains facteurs vus précédemment sont de ce fait oubliés. Si la fenêtre est trop grande, il faudra trop de bits pour coder le triplet de sortie et l’algorithme n’effectuera pas une bonne compression. De plus, la recherche de motifs deviendra de plus en plus coûteuse en temps (de l’ordre de $(N - F) \times F$ opérations). Il faut donc trouver un compromis pour obtenir une bonne compression en fonction de tous ces paramètres. En pratique, cette méthode fonctionne assez bien pour une taille de fenêtre de l’ordre de $N \leq 8192$ pour des raisons suivantes :

- Beaucoup de mots et de fragments de mots sont suffisamment courants pour apparaître souvent dans une fenêtre. C’est le cas pour « . . . sion », « . . . que », « la » . . .
- les mots rares ou techniques ont tendance à être répétés à des positions très proches. C’est par exemple le cas le mot « compression ».
- Si un caractère ou une suite de caractères sont répétés un certain nombre de fois consécutivement, le codage peut être très économe parce qu’il autorise la répétition à chevaucher les deux tampons.

1.6.1.2 La décompression

Celle-ci est simple et rapide. A partir de la suite de triplets, le décodage se fait en faisant coulisser la fenêtre de manière analogue au codage. Le dictionnaire est

Algorithme 1.10 Algorithme de compression LZ78

```

mot := ε ;
dictionnaire[0] := ε
indice := 1
repeat
  lire S le premier caractère du texte T restant à comprimer (on retire également S de T)
  if mot.S ∈ dictionnaire then
    mot := mot.S
    emis := faux,
  else
    émettre(indice de mot dans le dictionnaire, S)
    affecter(mot, S) à l'entrée indice du dictionnaire
    indice := indice + 1
    mot := ε
    emis := vrai
  end if
until fin du texte à compresser
if emis = faux then
  émettre(indice de mot dans le dictionnaire, S)
end if

```

Dictionnaire	lexème
0	<i>null</i>
1	<i>a</i> (0, <i>a</i>)
2	<i>ab</i> (1, <i>b</i>)
3	<i>b</i> (0, <i>b</i>)
4	<i>aba</i> (2, <i>a</i>)
5	<i>ba</i> (3, <i>a</i>)
6	<i>bb</i> (3, <i>b</i>)
7	<i>bba</i> (6, <i>a</i>)
8	<i>bbb</i> (6, <i>b</i>)
9	<i>abb</i> (2, <i>b</i>)

TAB. 1.7 – Compression par LZ78

l'entrée 1 du dictionnaire suivi de la lettre *b*. On code donc le facteur *ab* par (1, *b*) et on ajoute *ab* en position 2 du dictionnaire ...

1.6.2.2 Algorithme de décompression

La décompression est simple : il suffit de reconstruire le dictionnaire au fur et à mesure du décodage. Les numéros de facteur seront les mêmes que pour le codage et les facteurs pourront être interprétés sans problème.

Algorithme 1.11 Algorithme de décompression LZ78

```

mot := ε ;
dictionnaire[0] := ε
indice := 1
repeat
  lire suivant (ind, S) dans le texte comprimé
  if ind = 0 then
    émettre(S)
    dictionnaire[indice] := S
    indice := indice + 1
  else
    facteur := concatene(dictionnaire[ind], S)
    émettre(S)
    dictionnaire[indice] := facteur
    indice := indice + 1
  end if
until fin du texte à décompresser

```

Une propriété remarquable de cette méthode est que l'algorithme de compression et celui de décompression utilisent le même dictionnaire sans que celui-ci soit transmis. Il est entièrement reconstruit au cours de la décompression.

Exemple La données (0, *a*)(1, *b*)(0, *b*)(2, *a*)(3, *a*)(3, *b*)(6, *a*)(6, *b*)(2, *b*) à décompresser. Nous obtenons la suite d'émissions suivante :

a ab b aba ba bb bba bbb abb

1.7 Conclusion

Sommaire

2.1	Introduction	36
2.2	Codes compresseurs usuels	36
2.2.1	Formats GIF et PNG	36
2.3	Compression avec perte	36
2.3.1	Dégradation de l'information	36
2.3.2	Transformation des informations audiovisuelles	37
2.3.3	Le format JPEG	38
2.3.4	Le format MPEG	40
2.4	Conclusion	42

Résumé

Nous nous intéressons au problème de la compression d'images et les techniques pour résoudre ce problème.

2.1 Introduction

2.2 Codes compresseurs usuels

2.2.1 Formats GIF et PNG

Les formats de données compacts usuels tiennent compte de la nature des données à coder. Il sont différents s'il s'agit d'un son, d'une image ou d'un texte. Parmi les formats d'images, le format GIF (Graphic Interchange Format) est un format de fichier graphique bitmap proposé par la société CompuServe. C'est un format de compression pour des images pixellisée, c'est-à-dire décrite comme une suite de points (pixels) contenus dans un tableau ; chaque pixel a une valeur qui décrit sa couleur.

Le principe de compression est en deux étapes : tout d'abord, les couleurs pour les pixels (initialement, il y a 16,8 millions de couleurs codées sur 24 bits *RGB*) sont limités à une palette de 2 à 256 couleurs (2, 4, 8, 16, 32,64, 128 ou 256 qui est le défaut). La couleur de chaque pixel est donc approchée par la couleur la plus proche figurant dans la palette. Tout en gardant un nombre important de couleurs différentes avec une palette de 256, ceci permet d'avoir un facteur trois de compression. Puis, la séquence de couleurs des pixels est compressé par l'algorithme de compression dynamique *LZW* (Lempel-Ziv-Welch). Il existe deux versions de ce format développées respectivement en 1987 et 1989 :

- *GIF 87a* permet un affichage progressif (par entrelacement) et la possibilité d'avoir des images animés (les *GIFs* animés) en stockant plusieurs images au sein du même fichier.
- *GIF 89a* permet en plus de définir une couleur transparente dans la palette et de préciser le délai pour les animations.

Comme l'algorithme de décompression *LZW* a été breveté par Unisys, tous les éditeurs de logiciel manipulant des images *GIF* auraient pu payer une redevance à Unisys. C'est une des raisons pour lesquelles le format *PNG* est de plus en plus plébiscité, au détriment du format *GIF*. Le format *PNG* format analogue ; mais l'algorithme de compression utilisé est *LZ77*.

2.3 Compression avec perte

2.3.1 Dégradation de l'information

La valeur de l'entropie est une borne inférieure pour laquelle le taux de compression ne peut descendre au dessous de ce seuil. De plus, les heuristiques de réduc-

tion d'entropie permettent d'abaisser ce seuil en changeant de modèle de la source. Cependant aucune de ces méthodes ne prenait en compte le type de fichier à compresser et son utilisation. Considérons une image numérique. Les compresseurs sans perte transmettent le fichier sans utiliser l'information qu'il s'agit d'une image, et le fichier pourra toujours être restitué dans son intégralité. Mais il peut exister des formats beaucoup plus efficaces pour lesquels, si une partie de l'information est perdue, la différence ne soit jamais visible. Les fichiers ne pourront donc pas être restitués dans leur état initial, mais leur état permettra de visualiser des images d'au moins bonne qualité. Ce qui veut dire que la qualité qui était codée dans le fichier de départ était superflue puisqu'elle n'induit aucune différence visuelle.

Les compresseurs avec perte sont donc spécifiques du type de données à transmettre ou à stocker (images, son et vidéo pour la plupart), et utilisent cette information en codant le résultat visuel du fichier plutôt que le fichier lui-même. Ce type de compression est obligatoire pour les formats vidéo, par exemple où la taille des données et les calculs sont très importants.

Les compresseurs avec perte vont permettre de dépasser le seuil de l'entropie et obtenir des taux de compression très importants. Ils tiennent compte non plus seulement de l'information brute, mais aussi de la manière dont elle est perçue par des êtres humains, ce qui nécessite une modélisation de la persistance rétinienne ou auditive par exemple.

On pourra mesurer l'efficacité de ces compresseurs avec le taux de compression, mais la qualité des images produites n'est évaluée que grâce à l'expérience des utilisateurs. Nous présentons donc des formats standards, largement éprouvés. En outre, pour les informations audiovisuelles, un autre critère important viendra se greffer au taux de compression : la rapidité de la décompression. En particulier pour le son et la vidéo, la visualisation nécessite des calculs extrêmement rapides pour lire les fichiers compressés. La complexité algorithmique des décompresseurs est donc une contrainte aussi importante que la taille des messages compressés lorsqu'on traite des informations audiovisuelles.

2.3.2 Transformation des informations audiovisuelles

Les formats compacts de stockage d'images, de sons ou de vidéo utilisent toujours des sources étendus. Ils codent non pas pixel par pixel, mais prennent en compte un pixel et son entourage, pour observer des paramètres globaux, comme l'apparition des contours, les régularités, ou l'intensité des variations. Ces données seront traitées sous la forme de signaux pseudo-périodiques. Par exemple, si un morceau d'une image est un dégradé doux, la fréquence du signal sera faible, tandis que la fréquence d'une partie dont les couleurs, les textures varient beaucoup sera élevée.

Cela permettra un codage très efficace des zones de base fréquence sera souvent invisible ou inaudible, donc parfaitement acceptable.

2.3.3 Le format JPEG

Le codage au format JPEG (pour Joint Photographic Group) compresse des images fixes avec perte d'information. L'algorithme de codage est complexe, et se déroule en plusieurs étapes. Le principe de base est que les couleurs des pixels voisins dans une image diffèrent peu en général. De plus, une image est un signal : plutôt que des valeurs des pixels, on calcule des fréquences (transformée de Fourier DFT ou transformée en cosinus discrète DCT dans le cas JPEG). En effet, on peut considérer qu'une image est une matrice de couleurs des pixels ; et il se trouve que les images numériques sont principalement composées de basses fréquences DDCT quand on regarde la luminance des couleurs (dans le codage d'une image en mode luminance/Chrominance, YUV par exemple) plutôt qu'en mode RGB. Il existe plusieurs formules pour passer du mode RGB au mode YUV selon les modèles physiques de la lumière. Une formule classique pour trouver la luminance est celle-ci : $Y = 0,299 \times R + 0,587 \times G + 0,114 \times B$

L'idée est donc d'appliquer la DCT à transformer cette matrice en une matrice contenant les hautes fréquences en haut à droite et les basses fréquences en bas à gauche.

Considérons la matrice suivante donnant la luminance :

$$L = \begin{bmatrix} 76 & 76 & 76 & 255 & 255 & 255 & 255 & 255 \\ 255 & 76 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 76 & 255 & 150 & 150 & 255 & 255 & 255 \\ 255 & 76 & 255 & 150 & 255 & 150 & 255 & 255 \\ 255 & 255 & 255 & 150 & 255 & 150 & 29 & 29 \\ 255 & 255 & 255 & 150 & 255 & 150 & 29 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 29 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 29 & 29 \end{bmatrix}$$

$$D = \begin{bmatrix} 1631 & 71 & -83 & 49 & 94 & -36 & 201 & 32 \\ 6 & -348 & 49 & 2 & 26 & 185 & -60 & 86 \\ 39 & -81 & -123 & 27 & 28 & 0 & -43 & -130 \\ -49 & 3 & -10 & -25 & 19 & -126 & -33 & -92 \\ -101 & 43 & -111 & 55 & -4 & -3 & -120 & 50 \\ 25 & -166 & 62 & -50 & 50 & -2 & -9 & -7 \\ -64 & -34 & 54 & 11 & -69 & 0 & 26 & -54 \\ -48 & 73 & -48 & 28 & -27 & -33 & -23 & -29 \end{bmatrix}$$

Enfin, l'œil humain donne plus d'importance à la luminosité qu'aux couleurs. En ne gardant que les premiers termes de la décompression en luminance DCT (les plus importants), on perd un peu d'information mais l'image reste visible. Cette opération est appelée quantification et c'est la seule étape de tout codage JPEG qui soit avec perte d'information. L'opération consiste à arrondir la valeur $C(i, j) = \frac{DCT(i, j)}{Q(i, j)}$ où Q est une matrice de quantification

$$Q = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 150 & 29 & 77 \\ 24 & 35 & 55 & 64 & 81 & 150 & 29 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

$$C = \begin{bmatrix} 102 & 6 & -8 & 3 & 4 & -1 & 41 & 1 \\ 1 & -29 & 4 & 0 & 1 & 3 & -1 & 2 \\ 3 & -6 & -8 & 1 & 1 & 0 & -1 & -2 \\ -4 & 0 & 0 & -1 & 0 & -1 & 0 & -1 \\ -6 & 2 & -3 & 1 & 0 & 0 & -1 & 1 \\ 1 & -5 & 1 & -1 & 1 & 0 & 0 & 0 \\ -1 & -1 & 1 & 0 & -1 & 0 & 0 & -1 \\ -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

On peut remarquer sur l'exemple, qu'après quantification, beaucoup de valeurs sont identiques et proches de zéro. Le choix de la matrice de quantification respectivement avec des valeurs proches de 1 ou loin de 1 permet d'augmenter ou diminuer le niveau de détails. L'image est finalement codée comme une suite de nombres (une valeur quantifiée de DCT suivie du nombre de pixels ayant cette valeur qui sont consécutifs selon le balayage en zigzag (voir figure 2.1). Les faibles valeurs obtenues permettent d'espérer que l'entropie obtenue après ce RLE a été considérablement réduite. L'algorithme JPEG se termine alors par un codage statistique par exemple Huffman. Enfin, il est à noter que le standard JPEG-2000 où la transformation en cosinus et la quantification sont remplacées par une transformée en ondelettes dans laquelle ne sont conservés que certains niveaux. Le nombre de niveaux ou bandes conservés influe directement sur la taux de compression désiré.

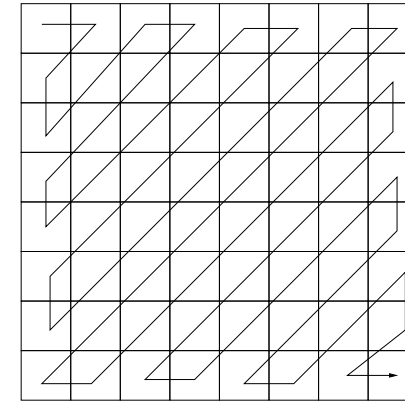


FIG. 2.1 – Balayage en zigzag de JPEG

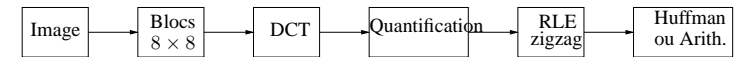


FIG. 2.2 – Compression JPEG

2.3.4 Le format MPEG

Le format MPEG (pour Motion Picture Experts Group) définit la compression d'images animées. L'algorithme de codage utilise JPEG pour coder une image mais prend en compte le fait que deux images consécutives d'une séquence vidéo sont très voisines. Une des particularités de normes MPEG est de chercher à compenser le mouvement (zoom par exemple ...) d'une image à la suivante. Une sortie MPEG-1 contient quatre sortes d'images : des images au format I (JPEG), des images P codées par différences avec l'image précédente (par compensation du mouvement deux images consécutives sont très voisines), des images bidirectionnelles B codées par différence avec l'image précédente et la suivante (en cas de différences moins importantes avec une image future qu'avec une image précédente), enfin des images basse résolution utilisées pour l'avance rapide sur un magnétoscope.

En pratique, avec les images P et B , il suffit d'une nouvelle image complète I toutes les 10 à 15 images. Un film vidéo étant composé d'images et de son, il faut également compresser le son. Le groupe MPEG a défini trois formats : les MPEG-1 Audio Layer I , II et III , le troisième est le fameux MP3. Le MPEG-1 est la combinaison de compressions d'images et de son, associées à une synchronisation par marquage temporel et une horloge de référence du systèmes comme indiqué sur la figure 2.3

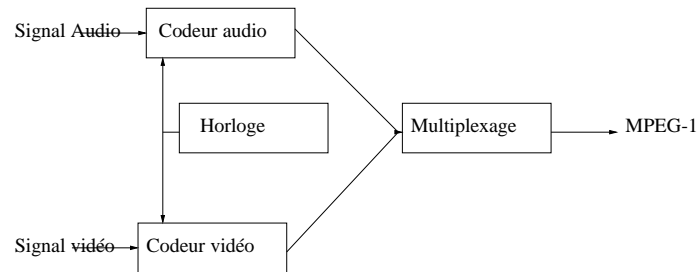


FIG. 2.3 – Compression MPEG-1

Depuis 1992, le format MPEG a évolué passant de MPEG-1 (320×240 pixels et une débit de $1,5\text{Mbits/s}$), à MPEG-2 en 1996 (avec quatre résolutions de 352×288 à 1920×1152 pixels plus un flux de sous-titres et de langues multiplexés, celui-ci est l'actuel format des DVD) ou encore à MPEG-4 en 1999 (format des vidéos conférences et du DivX où la vidéo est devenue une scène orientée objet). Enfin le son est décomposé en trois couches, MP1, MP2 et MP3, suivant les taux de compression (et donc les pertes) désirés. En pratique, les taux de compression par rapport à l'analogique sont d'un facteur 4 pour le MP1, de 6 à 8 pour le MP2, utilisé par exemple dans les vidéos CD et de 10 à 12 pour le MP3. Nous donnons une idée de la transformation MP3 sur la figure 2.4 : le son analogique est d'abord filtré et modifié par les transformées de Fourier (avec perte) et simultanément en cosinus (DCT).

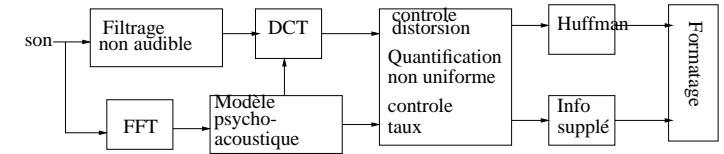


FIG. 2.4 – Compression du son MP3

Ensuite, la combinaison de ces deux transformations permet de faire une quantification et de contrôler le résultat obtenu pour ne conserver que la partie audible principale. Au final, une compression statistique est bien sûr appliquée, pour réduire le fichier au niveau de son entropie finale.

2.4 Conclusion

Ce chapitre clôt la partie sur la compression de données.

CHAPITRE

3

Compression du son

3.1 Introduction

3.2 Conclusion

Sommaire

3.1 Introduction	44
3.2 Conclusion	44

Résumé

Nous nous intéressons au problème de la compression du son et les techniques pour résoudre ce problème.