Compression de données

PEREIRA Vincent - LEPRETTE Franck - HACAULT Vincent

Décembre 2004

Table des matières

Chapitre 1

Introduction

La compression des données est un vaste sujet qui a fait l'objet de nombreux ouvrages et articles. Elle donne lieu aujourd'hui à de nombreuses recherches en raison des enjeux économiques. Elle est utilisée majoritairement dans les applications informatiques et elle est une des conditions d'existence du multimédia. L'utilisation de la compression et sa mise en pratique nécessitent des connaissances nombreuses et complexes tels que le calcul intégral, l'algèbre linéaire, la géométrie fractale, la théorie des probabilités $\cdots$ Dans cet exposé, nous allons ouvrir un champ de réflexion en indiquant tout d'abord quelles sont les méthodes générales utilisées aujourd'hui pour compresser les données puis dans un second temps, nous allons présenter deux méthodes très utilisées dans la compression des données.

Chapitre 2

Généralités sur la compression

2.1 Définition et intérêt de la compression

2.1.1 Définition

La compression consiste à réduire la taille physique de blocs d'informations. Elle est très utile pour plusieurs applications informatiques.
Les différents algorithmes de compression sont basés sur 3 critères :

- **Le taux de compression** : c'est le rapport de la taille du fichier compressé sur la taille du fichier initial.
- **La qualité de compression** : sans ou avec pertes (avec le pourcentage de perte).
- **La vitesse** de compression et de décompression.

Un compresseur utilise un algorithme qui sert à optimiser les données en fonction du type de données à compresser ; un décompresseur est donc nécessaire pour reconstruire les données grâce à l'algorithme dual de celui utilisé pour la compression.
La méthode de compression dépend du type de données à compresser car une image ou un fichier audio ne représentent pas le même type de données.

2.1.2 Intêret

De nos jours, la puissance des processeurs augmente plus vite que les capacités de stockage, et énormément plus vite que la bande passante des réseaux (car cela imposerait d'énormes changements dans les infrastructures de télécommunication).
Il y a donc un déséquilibre entre le volume des données qu'il est possible de traiter, de stocker, et de transférer.
Par conséquent, il faut donc **réduire la taille des données**. Pour cela, il faut exploiter la puissance des processeurs, pour pallier aux insuffisances des capacités de stockage en mémoire et des vitesses de transmission sur les réseaux.

Exemples

Choisissons une séquence vidéo avec les caractéristiques suivantes :
- 25 images par seconde,
- 16 millions de couleurs (soit 3 octets par pixel),
- Résolution de 640 x 480.

Sans compression, il faudrait un débit de 23 Mo/s (25 x 3 x 640 x 480), et pour donner un ordre d'idée, cela représente un débit 130 fois plus important que celui d'un lecteur de CD-ROM simple vitesse (150 Ko/s). De plus, si on souhaite stocker 2 heures de vidéo, il nous faudrait une unité de stockage de 162 Go (équivalent à 34 DVD de 4,7 Go).

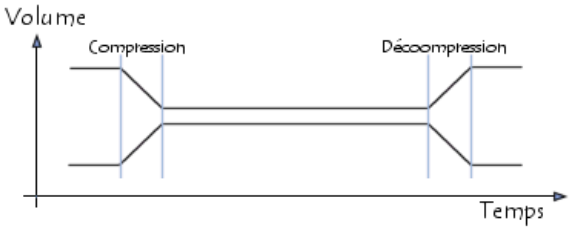
Un autre exemple très courant d'utilisation de la compression est pour le transport dans les réseaux cablés (minitel, internet), et sans fil (communication par satellite, téléphone portable).

Dans le domaine de la compression, il existe plusieurs façons de comparer les types de compression. Pour cette raison, nous allons voir comment classifier les algorithmes de compression.

2.2 Classification des algorithmes de compression

2.2.1 Compression symétrique / asymétrique

Dans le cas de la compression **symétrique**, la même méthode est utilisée pour compresser et décompresser l'information, il faut donc la même quantité de travail pour chacune de ces opérations. C'est ce type de compression qui est généralement utilisée dans les transmissions de données.
La compression **asymétrique** demande plus de travail pour l'une des deux opérations, la plupart des algorithmes requiert plus de temps de traitement pour la compression que pour la décompression. Des algorithmes plus rapides en compression qu'en décompression peuvent être nécessaires lorsque l'on archive des données auxquelles on accède peu souvent (pour des raisons de sécurité par exemple).



Compression de type symétrique.

2.2.2 Compression physique / logique

On considère généralement la compression comme un algorithme capable de comprimer des 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 se base sur la façon dont les données sont compressées ou plus précisément comment est-ce que les données sont réarrangées.
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.
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 est basée exclusivement sur l'information contenue à l'intérieur même des données.

2.2.3 La compression statistique / numérique

On peut encore distinguer les algorithmes qui travaillent au niveau statistique et ceux qui opèrent au niveau numérique.
Pour les premiers, la valeur des motifs ne compte pas. Ce sont les probabilités qui comptent, et le résultat est inchangé par substitution des motifs tandis que pour les seconds, les valeurs des motifs influent sur la compression (par exemple JPEG), et les substitutions sont interdites.

Enfin le critère de classification le plus pertinent est basée sur la perte des données.

2.2.4 La compression sans / avec perte

Compression sans perte

Les algorithmes de compression **sans perte** (connu aussi sous le nom de non destructible, réversible, ou conservative) sont des techniques permettant une reconstitution exacte de l'information après le cycle de compression / décompression. Il existe 3 types d'algorithmes de compression sans perte :

- **Codage statistique** :
  Le but est de :
  - Réduire le nombre de bits utilisés pour le codage des caractères fréquents.
  - Augmenter ce nombre pour des caractères plus rares.

Exemple
Certaines informations sont plus souvent présentes que d'autres dans les données que l'on veut compresser.
Dans un fichier HTML par exemple, on trouvera beaucoup de signes $<$, / , et $>$. On va chercher à coder les données se répétant souvent sur moins de bits, et les données moins fréquentes sur plus de bits. 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 $\cdots$).

- **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 pour résoudre se problème : la compression avec perte.

Compression avec perte

Son principe est basé sur l'étude précise de l'oeil et de l'oreille humaine. Les signaux audio et vidéo contiennent une part importante de données que l'oeil et l'oreille ne peuvent pas percevoir et une part importante de données redondantes.
Les objectifs de la compression avec pertes sont d'éliminer les données non pertinentes pour ne transmettre que ce qui est perceptible et, comme pour la compression sans perte, d'éliminer l'information redondante.
Ce type de compression engendre une dégradation indiscernable à l'oeil (ou à l'oreille) ou suffisamment faible, en contrepartie d'un taux de compression très élevé.
Il existe des algorithmes de compression consacrés à des usages particuliers, dont en voici 3 :
- Compression du son (Audio MPEG, ADCPM ...).
- Compression des images fixes (JPEG,...).
- Compression des images animées (MPEG, ...).

2.2.5 Performances et efficacité

Nous allons nous intéresser à une série d'exemples afin de montrer le gain de compression dans des domaines informatiques.

Gain en temps sur les lectures / écritures :
- Temps pour écrire une image de 14Mo : 5s (soit 2,8Mo/s).
- Temps pour compresser une image de 14Mo en JPEG : 3,4s, soit 6,5Mo/s, résultat : 503Ko (soit 4% de l'original)

- Temps pour écrire une image JPEG de 503Ko : 0,2s
- Temps total compression + écriture : 3,6s, soit 1,4s de moins (gain de 24% en temps et 96% en volume stocké sur disque)

Gain sur les temps de transmission :
- Temps de transmission. 56Kbps/s d'un fichier HTML de 30Ko : 5,4s
- Temps de compression du fichier HTML au format« GZIP » : 0,05s (soit 600Ko/s), taille compressée : 15Ko (gain de 50% en volume)
- Temps de transmission de l' HTML compressé : 2,7s
- Temps total compression /transmission/ décompression : 2,8s, au lieu de 5,4s, soit un gain de près de 50% en temps.

Ainsi, nous avons pu voir que les méthodes de compression sont très utiles et apportent parfois des résultats spectaculaires. Sachant le fichier à compresser, chaque algorithme a ses avantages et ses inconvénients et il faut donc choisir l'algorithme en fonction du type de données.

Chapitre 3

Compression de type statistique : codage de Huffman

3.1 Introduction

Le codage de Huffman est un algorithme de compression des données basé sur les fréquences d'apparition des caractères apparaissant dans le document initial. Il a été développé par un étudiant de la MIT (*Massachusetts Institute of Technology*), **David A. Huffman** en 1952. Cette technique est largement utilisée car elle est très efficace et on observe selon le type de données des taux de compression allant de 20% à 90% mais plus généralement entre 30% et 60%. La dernière partie de cette section traitera des performances.
Le principe de compression est utilisé dans le codage d'image TIFF (Tagged Image Format File) spécifié par Microsoft Corporation et Aldus Corporation. La méthode JPEG (Join Photographic Experts Group) utilise aussi la compression de type Huffman pour coder les informations d'une image. (Elle utilise d'ailleurs des tables prédéfinies).
Ce procédé fait partie des méthodes de compression de type dites statistiques. Cela repose sur le principe sur l'attribution de codes plus courts pour des valeurs fréquentes et de codes plus longs pour les valeurs moins fréquentes. Cela est plus efficace que la représentation actuelle qui consiste, quant à elle, à utiliser une longueur fixe pour chaque symbole (exemple : un octet par caractère, code ASCII).

3.2 Principe de l'algorithme

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

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

3.2.1 Table de fréquence d'apparition

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

$$f_s = \frac{nombre\ d'occurrences\ de\ s}{nombre\ de\ symboles}$$

Ensuite on trie le tableau en fonction de la fréquence d'apparition (de façon croissante) puis suivant le symbole suivant.

Supposons par exemple que nous ayons le texte suivant composé des symboles pris dans le code ASCII :
"*L'algorithme de Huffman est une méthode qui permet de compresser les données*".

Pour chaque symbole on calcule tout d'abord le nombre d'occurrences de chacun puis sa fréquence d'apparition suivant la formule citée précédemment. La table de fréquences T triée est (*Afin de simplifier l'exemple, on a négligé la casse*) :

Symbole	Nombre d'occurrences	Fréquence d'apparition
c	1	$1/76 = 1.32\%$
g	1	$1/76 = 1.32\%$
q	1	$1/76 = 1.32\%$
’	1	$1/76 = 1.32\%$
a	2	$2/76 = 2.63\%$
f	2	$2/76 = 2.63\%$
i	2	$2/76 = 2.63\%$
p	2	$2/76 = 2.63\%$
u	3	$3/76 = 3.95\%$
h	3	$3/76 = 3.95\%$
l	3	$3/76 = 3.95\%$
d	4	$4/76 = 5.26\%$
n	4	$4/76 = 5.26\%$
o	4	$4/76 = 5,26\%$
r	4	$4/76 = 5.26\%$
t	4	$4/76 = 5.26\%$
m	5	$5/76 = 6.58\%$
s	5	$5/76 = 5.26\%$
	11	$11/76 = 14.47\%$
e	14	$14/76 = 18.42\%$

3.2.2 Construction de l'arbre de Huffman

L'arbre binaire de Huffman est la structure de données qui va nous permettre d'attribuer à chaque symbole une représentation binaire optimale. Afin de construire l'arbre, on utilise la table de fréquences précédemment construite qu'on appelle T et on applique l'algorithme suivant :

Algorithme 1: Construction de l'arbre

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

Résultat :
- A : L'arbre binaire résultant

begin

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

tant que *(Q non vide)* **faire**

Créer un nouveau noeud z dans A tel que :
gauche(z) = x = extraire-min (Q)
droite(z) = y = extraire-min (Q)
o(z) = o(x) + o(y)
Insérer (z,Q)

fin

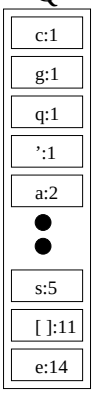
end

De façon informelle, on utilise une file d'attente Q dans laquelle on place les noeuds correspondants au couple [symbole : nombre d'occurrences du symbole] de tous les symboles. Ensuite on extrait de la file d'attente les 2 noeuds ayant la valeur minimale puis on crée un nouveau noeud dans l'arbre de Huffman ayant pour fils les 2 deux sélectionnés, on rajoute ensuite le noeud nouvellement crée dans la file d'attente , et on réitère jusqu'à ce que la file soit vide.

Reprenons l'exemple précédent cité et appliquons l'algorithme de construction de l'arbre étape par étape.

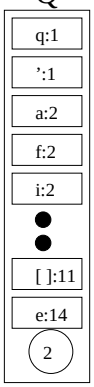
Initialisation

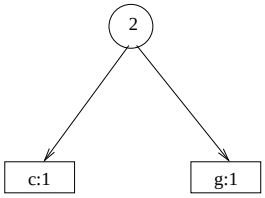
Q



Etape 1

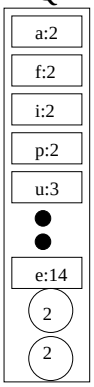
Q

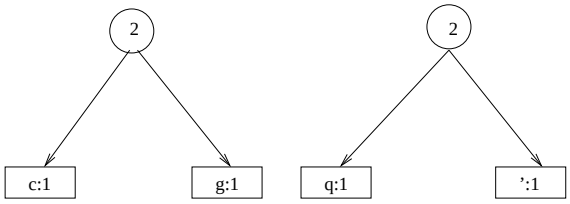


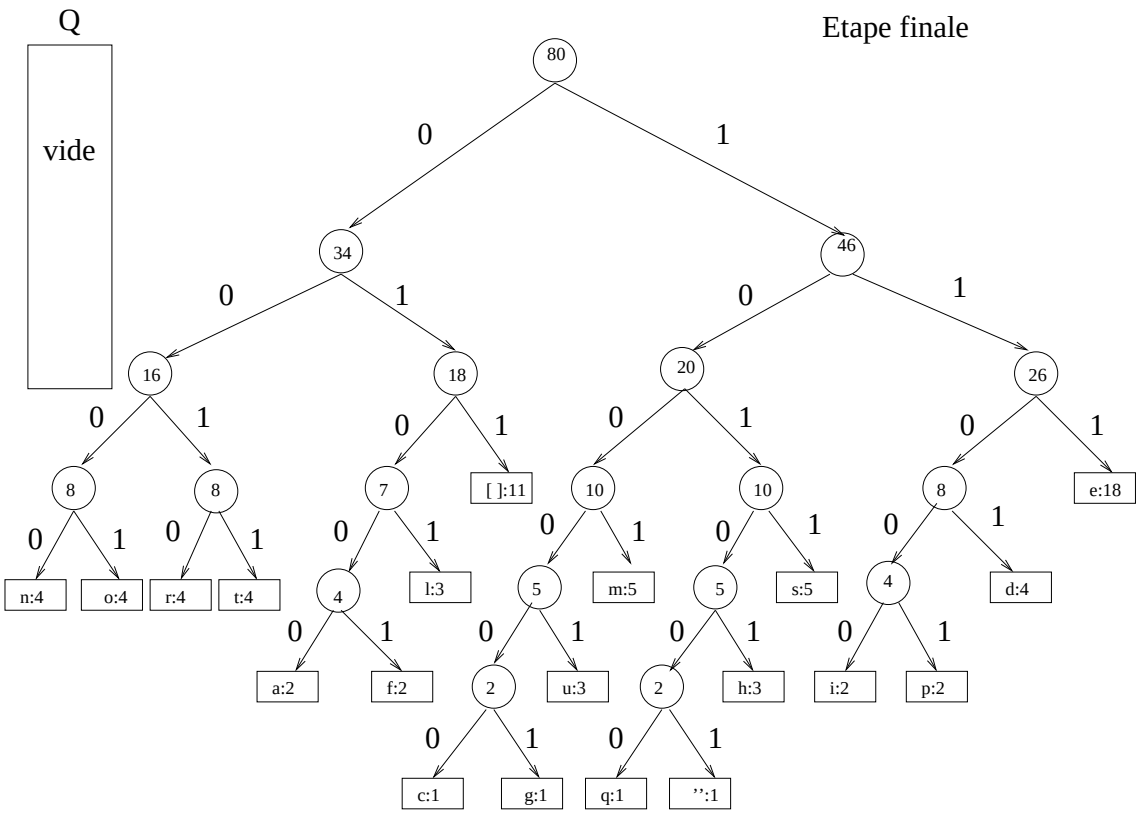


Etape 2

Q







L’arborescence de Huffman finalement construite, on étiquette les arcs de la façon suivante :

- Les arcs reliant un noeud à son fils gauche sont étiquetés par ’0’.
- Les arcs reliant un noeud a son fils droit sont étiquetés par ’1’.

De cette manière, chaque feuille représentant un symbole peut être redéfinie par un nombre binaire correspondant au chemin entre la racine et la feuille de l’arborescence. Ainsi, les symboles les plus utilisés ont une représentation binaire moins importante (en terme de taille) que les symboles les moins utilisés. Ceci permet de représenter chaque symbole de façon optimale et permet de réaliser une compression des données efficacement.

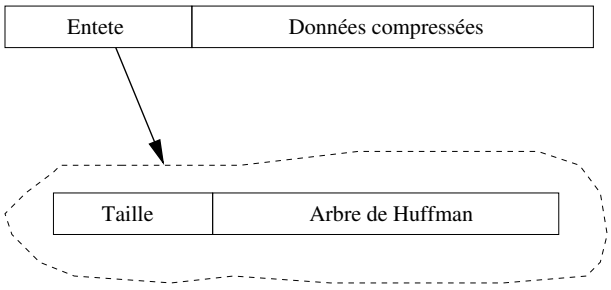
Si on reprend notre exemple, on obtient la table de correspondance suivante :

Symbole	Représentation binaire	Taille(en bits)	Gain(en bits)
c	100000	6	2
g	100001	6	2
q	101000	6	2
’	101001	6	2
a	01000	5	3
f	01001	5	3
i	11000	5	3
p	11001	5	3
u	10001	5	3
h	10101	5	3
l	0101	4	4
d	1101	4	4
n	0000	4	4
o	0001	4	4
r	0010	4	4
t	0011	4	4
m	1001	4	4
s	1011	4	4
	011	3	5
e	111	3	5

De cette manière, notre phrase exemple devient :
"*L’algorithme de Huffman est une méthode qui permet de compresser les données*"
l = 0101
’ = 101001
a = 01000
g = 100001
...
Message compressé = 0101101001010000101100001 etc...
(Á noter que le message compressé est une suite binaire de bits alors que le message original est une suite de symboles qui peut, comme nous l’avons vu en cours, être ramenée à une suite de bits).

3.2.3 Compression / Décompression

- **Compression** :
  Pour la compression des données, on applique l’algorithme de Huffman comme indiqué précédemment afin de calculer l’arbre binaire de Huffman. Ensuite, lorsqu’on désire transmettre des données à un destinataire, l’expéditeur émet un fichier contenant 2 parties :
  - Un entête : Celui ci contient la taille originale du fichier, l’arbre binaire de Huffman calculé précédemment (Cette structure de données peut être aisément représenter par un tableau simple).
  - Les données : Chaque symbole des données originales est substitué par sa représentation binaire correspondante.



Message à transmettre compressé

– **Decompression** :
Lors de la réception d'un message, le destinataire récupère dans l'entête l'arbre de Huffman et substitue chaque suite de bits correspondant à un chemin dans l'arborescence de Huffman par le symbole équivalent.On remarque qu'il ne peut pas avoir d'ambiguïté sur l'interprétation d'une séquence de bits à cause de la structure intrinsèque de l'arborescence.

3.2.4 Remarques

– Dans notre exemple nous avons appliqué l'algorithme sur un symbole mais on peut améliorer de façon significative l'efficacité en l'appliquant sur une suite de symboles (par exemple 2 ou 3 etc..). Par contre la taille de l'arbre binaire de Huffman ainsi que celle de l'entête pour transmettre des données sont significativement plus importantes. Il faut aussi considérer le temps de traitement qui est plus long.
– L'algorithme de Huffman n'est pas à proprement parler un algorithme de compression réservé aux réseaux (et aux couches basses en particulier) puisque ceci est réalisé au niveau applicatif ; néanmoins, du au fait de l'encapsulation des couches, la propagation du message compressé se réalise jusque dans les couches les plus basses.
– La méthode de Huffman possède l'inconvénient de devoir analyser statistiquement le message original ce qui peut prendre relativement du temps. De ce fait, on utilise parfois des tables statistiques prédéfinies ce qui évite d'analyser le texte original au préalable. C'est ce qu'on appelle des tables statiques alors que les autres sont habituellement dénommées dynamiques.

3.3 Performances

L'intérêt de la compression est de pouvoir réduire au maximum la taille des informations originales, comme nous l'avions indiqué en première partie. Cette mesure peut-être effectuée par le taux de compression qui est défini par la formule suivante :

$$\rho = 1 - \frac{Taille\ compressee}{Taille\ originale}$$

Par ailleurs, on peut aussi utiliser l'entropie permet de connaître le nombre minimum de bits nécessaires au codage d'un fichier. On rappelle que la formule de l'entropie est la suivante :

$$E = -\sum_{N}^{1}(P_k * log_2(P_k))$$

où P_k est la fréquence d'apparition du k-ième symbole parmi les n possibles. Analysons les performances de l'algorithme de Huffman grâce au taux de compression puis avec l'entropie sur notre exemple tout d'abord puis nous généraliserons.

3.3.1 Performances sur l'exemple

Taux de compression

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

Calculons la taille originale du message en bits : Le message contient 76 symboles et chaque symbole peut être représenté par le code ASCII équivalent sur 8 bits.

$$Taille\ originale = 76 * 8 = 608\ bits$$

Calculons à présent la taille du message compressé (il suffit de sommer la taille de la représentation binaire de chaque symbole), on obtient :

$$Taille\ compressee = 300\ bits$$

On peut donc a présent calculé le taux de compression et on obtient :

$$\rho = 1 - \frac{Taille\ compressee}{Taille\ originale} = \frac{300}{608} = 0,5066$$

Le taux de compression dans notre exemple est proche de 50% ce qui est relativement très bon.

Entropie

$$E = -\sum_{N}^{1}(P_k * log_2(P_k))$$

$$E = -\left[(\frac{1}{76} * log_2\frac{1}{76}) + (\frac{1}{76} * log_2\frac{1}{76}) + \cdots + (\frac{11}{76} * log_2\frac{11}{76}) + (\frac{14}{76} * log_2\frac{14}{76})\right]$$

$$E = 3.92$$

Ce qui signifie qu'il faut au minimum $3.92 * 76 = 297.92$ bits pour coder la chaîne. Avec 300 bits grâce à la méthode de Huffman on est très proche de l'optimalité.

3.3.2 Performances générales

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

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

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

3.4 Conclusion

L'algorithme de Huffman est un procédé largement répandu et qui se révèle être un algorithme performant en moyenne. Il est le plus représentatif des algorithmes de compression dit de type statistique : il en existe néanmoins d'autres tels que le RLE(*Run Length Encoding*), ou bien VLC (*Variable Length Code*) dénommé souvent codage entropique. Hormis tout ces avantages, Huffman présente l'inconvénient d'être relativement ancien et d'autres algorithmes plus récent reposant sur d'autres principes tels que les algorithmes de compression à dictionnaire (exemple : LZW) arrivent a des taux de compression supérieurs en moyenne à ceux que proposent l'encodage de Huffman.

Chapitre 4

Compression de type Dictionnaire

L'algorithme de Huffman a longtemps dominé le monde de la compression, et c'est en 1978 où sont apparues les méthodes de compression de Abraham Lempel et Jacob Ziv, méthodes dites de type dictionnaire aussi appelés algorithmes à substitution de facteurs.

4.1 Exemples de compressions intuitives de type dictionnaire

4.1.1 Compression de texte multi-dictionnaires

Cette méthode utilise un dictionnaire pour chaque taille de mot. L'idée est de remplacer un mot par la taille de celui ci et son index dans le dictionnaire. Une telle compression entraîne des gains très significatifs.
Mais cette technique n'est pas utilisée car elle nécessite des astuces pour repérer les conjugaisons, les pluriels ou encore les majuscules. De plus elle nécessite qu'un dictionnaire soit présent pour compresser et pour décompresser, ce qui représente une contrainte en espace mémoire.

4.1.2 Compression de texte multi-langages

Le principe d'une méthode de compression de texte multi-langages est de remplacer certains mots du dictionnaire par leur traduction dans une autre langue, de façon à diminuer la longueur de la séquence de mots dans une phrase. On peut ainsi obtenir de bons taux de compression.
En effet si nous prenons une phrase française telle que 'nous mangeons', elle pourrait être remplacée par 'we eat', qui est une traduction franco-anglaise et qui permet à la fois un décodage simple et un gain en taille(7 caractères dans ce cas).
Néanmoins l'utilisation d'une telle méthode ne peut fonctionner que s'il n'existe aucune hésitation sur l'interprétation, telle que les phrases à interdépendance syntaxesémantique (par exemple *La petite brise la glace*).

4.2 L'algorithme LZW

4.2.1 Historique des algorithmes LZ**

C'est en 1977 que Jacob Ziv et Abraham Lempel fournissent une technique de compression différente de l'algorithme de Huffman, et capable de donner de meilleurs taux de compression. Ils mettent ainsi en place l'algorithme LZ77.
Puis vient LZSS, version amélioré de LZ77 par Storer et Szymanski puisque la recherche des séquences dans le dictionnaire est réduite logarithmiquement.
Enfin vient l'algorithme LZ78, plus connu sous le nom LZW, amélioration faite par Terry Welch en 1984 de LZSS de par le fait que les séquences sont rangées dans une arborescence. Il porte le nom de ses 3 inventeurs : Lempel, Ziv et Welch.

4.2.2 Le principe

Le principe est fondé sur le fait qu'une séquence de caractères peut apparaître plusieurs fois dans un fichier.
L'algorithme LZW de compression consiste à émettre à la place des séquences, les adresses des de ces séquences d'un dictionnaire généré à la volée.
C'est un algorithme de compression nettement plus performant en moyenne que les algorithmes statistiques puisqu'il permet d'obtenir des gains plus élevés sur la majorité des fichiers.
L'algorithme LZW se distingue des méthodes statistiques pour plusieurs raisons :

- Le fichier comprimé ne stocke pas le dictionnaire, ce dernier est automatiquement généré lors de la décompression. Il n'existe donc pas de table d'entête.
- Contrairement aux méthodes statistiques qui utilisent la probabilité de présence sur un ensemble de taille fixe de symboles, l'algorithme LZW représente un algorithme d'apprentissage, puisque les séquences répétitives de symboles sont dans un premier temps détectées puis compressées seulement lors de leurs prochaines occurrences. Le taux de compression est dépendant de la taille du fichier. Plus la taille est importante, et plus le taux de compression l'est aussi.
- Il permet le compactage à la volée, puisqu'il n'y a pas à lire le fichier au préalable, il compresse les séquences de symboles au fur et à mesure.

4.2.3 L'algorithme de compression

Le principe de l'algorithme LZW de compression

L'objectif de l'algorithme de compression LZW est de construire un dictionnaire où chaque séquence sera désignée par une adresse dans celui ci. De plus chaque séquence ne s'y trouvant pas y est rajoutée. Au final, on se retrouve avec une suite d'entiers, des adresses pointant vers une séquence contenue dans le dictionnaire.
Le dictionnaire est donc un tableau dans lequel sont rangées des séquences de symboles de taille variable, repérées par leurs adresses (leur position dans le tableau). La taille de ce dictionnaire n'est pas fixe et les premières adresses de 0 à 255 du dictionnaire contiennent les codes ASCII. Les séquences ont donc des

adresses supérieures à 255.

Voici l'algorithme de compression :

Algorithme 2: L'algorithme LZW de compression

Données :
- Le dictionnaire des symboles rencontrés : Dico
- Le fichier à compresser : Fichier

Résultat :
- Le fichier compresser : Fichier

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 d'utilisation de l'algorithme LZW de compression

A l'aide de l'algorithme LZW, nous voulons compresser la séquence suivante : 'SISI-ET-ISIS'
On lit donc le premier caractère soit 'S'. Puis on lit le suivant : 'I'. On forme la séquence u=s+t soit 'SI', puisque celle ci n'est pas présente dans le dictionnaire, on l'ajoute dans le dictionnaire. Cette séquence aura l'adresse 256, première séquence située dans le dictionnaire après la table ASCII. Enfin on attribue l'adresse de s (valeur ASCII de 'S') dans le fichier soit **83** ou (1010011 en binaire). On continue ensuite avec le caractère suivant.
La compression effective a lieu lorsque qu'on rencontrera pour la seconde fois une paire ('SI' par exemple) déjà présente dans le dictionnaire. Dans ce cas on émettra l'adresse de la séquence 'SI' et non l'adresse de la séquence 'S' et de la séquence 'I', on ajoutera par la suite dans le dictionnaire la séquence de 3 caractères (soit 'SIS' dans notre exemple).

Voici un tableau résumant les opérations effectuées sur l'exemple lors du déroulement de l'algorithme LZW de compression :

	Phase 1	Phase 2	Phase 3	phase 4	Phase 5	Phase 6
Valeur de s	S	I	S	SI	-	E
Valeur de t	I	S	I	-	E	T
Valeur de u	SI	IS	SI	SI-	-E	ET
u appartient au Dico ?	Non	Non	**Oui**	Non	Non	Non
Ajout dans le Dico de	SI	IS		SI-	-E	ET
Ecrire adresse de	S soit **83**	I soit **73**		SI soit **256**	- soit **45**	E soit **69**

	Phase 7	Phase 8	Phase 9	phase 10	Phase 11	Phase 12
Valeur de s	T	-	I	IS	I	IS
Valeur de t	-	I	S	I	S	
Valeur de u	T-	-I	IS	ISI	IS	
u appartient au Dico ?	Non	Non	**Oui**	Non	**Oui**	
Ajout dans le Dico de	T-	-I		ISI		
Ecrire adresse de	T soit **84**	- soit **45**		IS soit **257**		IS soit **257**

Voici le dictionnaire résultant de l'algorithme :

Adresse	Séquence
0..255	ASCII
256	SI
257	IS
258	SI-
259	-E
260	ET
261	T-
262	-I
263	ISI

Enfin voici la suite d'adresse contenu dans le fichier compressé :

83.73.256.45.69.84.45.257.257

On peut ajouter que les adresses ne sont pas de taille maximale dès le départ de l'algorithme. Elles sont incrémentées d'1 bit dès que l'on atteint une nouvelle puissance de 2.
Dans notre exemple lorsqu'on obtient une adresse à écrire de taille plus grande que 255, les adresses sont incrémentées d'1 bit.
Plusieurs méthodes sont possibles pour permettre de prévenir le décompresseur du changement de la taille des adresses.
Premièrement, l'utilisation d'un code spécial est nécessaire qui a pour adresse par exemple 255 et signifie que les adresses seront incrémentées de 1 bits à partir de sa lecture. Ainsi le décompresseur est prévenu lors de la lecture d'un telle

code et sait alors que le nombre de bits à lire est incrémenté (dans ce cas ci soit 9 bits). Ce processus fonctionne si l'on condamne certaines adresses du dictionnaire (255, 511,1023 ...,) en n'y plaçant pas de séquences d'octets.
Remarque : Un problème se pose, c'est la façon dont on peut différencier la vraie valeur '255' de l'indicateur d'augmentation de bits. Il suffit pour résoudre ce problème d'émettre l'adresse 255 sur 8 bits suivi immédiatement du code '255' mais sur 9 bits soit 011111111.
Une autre méthode, moins élégante, qui utilise une entête pour informer l'algorithme de décompression des positions où dans le fichier l'on à augmenter la taille des adresses.
Le code 'SP' peut ne pas être le seul code de communication avec l'algorithme de décompression. D'autres codes peuvent par exemple servir à vider des dictionnaires périodiquement, à créer d'autres dictionnaires : il n'y a pas de limites.

4.2.4 L'algorithme de décompression

Le principe de l'algorithme LZW de décompression

L'algorithme de décompression LZW reconstruit le dictionnaire au fur et à mesure de la lecture du fichier compressé. Son processus est très proche de l'algorithme de compression.
On commence par lire les 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 correspondant à l'adresse dans le fichier décompresser.
Voici l'algorithme de décompression :

Algorithme 3: L'algorithme LZW de décompression

Données :
- Le dictionnaire des symboles rencontrés : Dico
- Le fichier à décompresser : Fichier

Résultat :
- Le fichier décompresser : Fichier

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 d'utilisation de l'algorithme LZW de décompression

A l'aide de l'algorithme LZW, nous voulons décompresser la séquence suivante : '83.73.256.45.69.84.45.257.257'
On lit donc la première adresse soit 83. On écrit la séquence associée à cette adresse du dictionnaire. Ensuite on lit l'adresse suivante. On s'assure que l'adresse lue appartient au dictionnaire et que ce n'est pas un code spécial de contrôle, ensuite on écrit la séquence associée à cette adresse. On ajoute dans le dictionnaire la concaténation de l'ancienne adresse lue et la première lettre de la séquence. On continue ensuite avec les adresses suivantes jusqu'à la fin du fichier.
Remarque : nous ne nous préoccuperons pas de la taille des adresses inscrites dans le fichier à décompresser.
Voici un tableau résumant les opérations effectuées sur l'exemple lors du déroulement de l'algorithme LZW de compression :

	Phase 1	Phase 2	Phase 3	phase 4
Valeur de a	S=*Séquence*(83)	I=*Séquence*(73)	SI=*Séquence*(256)	-=*Séquence*(45)
Valeur de b	73	256	45	69
b appartient au Dico ?	Oui	Oui	Oui	Oui
Valeur de s	I=*Séquence*(73)	SI=*Séquence*(256)	-=*Séquence*(45)	E=*Séquence*(69)
Valeur de t	I	S	-	E
Ajout dans le Dico de	Si	IS	SI-	-E
Ecrire séquence	**S** *et* **I**	**SI**	-	**E**

	Phase 5	Phase 6	Phase 7	phase 8
Valeur de a	S=*Séquence*(69)	I=*Séquence*(84)	SI=*Séquence*(45)	-=*Séquence*(257)
Valeur de b	84	45	257	257
b appartient au Dico ?	Oui	Oui	Oui	Oui
Valeur de s	T=*Séquence*(84)	-=*Séquence*(45)	-=*Séquence*(45)	IS=*Séquence*(257)
Valeur de t	T	-	I	I
Ajout dans le Dico de	ET	T-	-I	ISI
Ecrire séquence	**T**	**-**	**IS**	**IS**

Le dictionnaire généré par l'algorithme de décompression est bien évidemment le même que celui généré par l'algorithme de compression. (cf le dictionnaire produit précédemment par l'algorithme de compression.) Nous retrouvons donc notre phrase de départ ('SISI-ET-ISIS'), et nous pouvons aussi remarquer que l'algorithme de décompression est nettement plus rapide que l'algorithme de compression. Dans notre exemple seulement 8 phases ont été nécessaires pour décompresser le fichier alors que la compression a en nécessité 12. L'algorithme de décompression est en moyenne 3 à 10 fois plus rapide que celui de la compression.

4.2.5 Efficacité de l'algorithme LZW

La phrase 'SISI-ET-ISIS' est codée sur 96 bits (12***8** bits (dans le code ASCII)). Après utilisation de l'algorithme LZW de compression, le fichier compressé

possède une taille de 79 bits (2*8+7*9), donc on a ici un taux de compression de l'ordre de

$$\rho = 1 - \frac{79}{96} \approx 0.18 \approx 18\%.$$

En général lorsque la taille du fichier n'est pas trop petite, le taux de compression est supérieur à celui obtenu par une méthode statistique. On peut rajouter que l'algorithme LZW est nettement plus rapide qu'un algorithme de type statistique en terme de traitement.
On peut noter qu'il est possible de compresser un fichier déjà compressé auparavant par un algorithme de type statistique ou l'inverse.

Variante de l'algorithme LZW

Il existe des variantes des compressions LZ :
- LZP : Algorithme qui se base sur la répétition de phrases entières.
- LHA, LZS, LZX, LZH : Algorithmes quasiment identiques, encore dérivés du LZ77. Il n'est employé que pour l'utilitaire Lharc, très répandu au Japon, mais de moins en moins utilisé dans le Monde.

4.2.6 Conclusion

L'algorithme LZW est aujourd'hui considéré comme la méthode de compression la plus efficace et une des plus connues. Elle est relativemnt rapide ce qui a rendu l'utilisation de la compression possible sur les disques durs de façon transparente. Cette méthode est aussi utilisée dans le format *.gif*, mais encore dans les compresseurs tels que *ZIP*, *ARJ*.
Néanmoins elle était peu utilisée car elle était brevetée par la société Unisys jusqu'en juillet 2004 (8 Juillet 2004) où le brevet a expiré. Par conséquent, cette dernière ne peut plus à présent réclamer des droits sur l'utilisation du format gif par exemple car celui ci reposait sur l'algorithme LZW.

Chapitre 5

Conclusion

La compression des données est appelée à prendre un rôle encore plus important en raison du développement des réseaux et du multimédia. Son importance est surtout due au décalage qui existe entre les possibilités matérielles des dispositifs que nous utilisons (débits sur Internet, sur Numéris et sur les divers câbles, capacité des mémoires de masse $\cdots$) et les besoins qu'expriment les utilisateurs (visiophonie, vidéo plein écran, transfert de quantités d'information toujours plus importantes dans des délais toujours plus brefs). Quand ce décalage n'existe pas, ce qui est rare, la compression permet de toutes façons des économies. Les méthodes déjà utilisées couramment sont efficaces et sophistiquées (Huffman, LZW, JPEG) et utilisent des théories assez complexes, les méthodes émergentes sont prometteuses (fractales, ondelettes) mais nous sommes loin d'avoir épuisé toutes les pistes de recherche. Les méthodes du futur sauront sans doute s'adapter à la nature des données à compresser et utiliseront l'intelligence artificielle mais on est dans le droit de se poser la question suivante : La compression sera-t-elle utile dans le futur alors que les contraintes sur les ressources deviennent de moins en moins importantes ?