

Année 2017 - 2018

Projet Math Info :
Compression de données

CHEAR Thierry
EAR Philippe

Enseignant encadrant : YUNÈS Jean-Baptiste

Contents

1	Huffman Statique	3
1.1	Fonctionnement	3
1.1.1	Compression	3
1.1.2	Décompression	4
1.2	Structure de données utilisée	4
1.3	Optimisation	4
1.4	Exemple	4
1.5	Spécification	6
2	Huffman Dynamique	7
2.1	Fonctionnement	7
2.1.1	Manipulation de l'arbre	7
2.1.2	Compression	8
2.1.3	Décompression	9
2.2	Structures de données utilisées	9
2.3	Optimisation	9
2.4	Exemple	10
2.5	Spécification	12
3	Pseudo-code	13
3.1	Huffman Statique Compression	13
3.2	Huffman Statique Décompression	14
3.3	Huffman Dynamique Compression	14
3.4	Huffman Dynamique Décompression	15
4	Performances	16
4.1	Fichiers texte	16
4.2	Fichiers image	16
4.3	Fichiers audio	17
4.4	Comparaison des taux de compression	17
5	Liste des outils utilisés	18

Introduction

La compression de données est une opération transformant une série de bits A , en une série de bits B , plus courte, mais transportant la même quantité d'information. Afin de retrouver la série A à partir de la série B , on y applique un algorithme de décompression. Une telle opération est essentielle aujourd'hui, car elle permet de réduire l'espace occupé par les différents fichiers se trouvant sur un espace de stockage, mais aussi le temps de transfert de ces fichiers d'un espace à un autre, au prix du temps de la décompression (très souvent bien plus court).

On trouve principalement deux types de compression :

1. La compression avec perte (lossy)

La compression avec perte ne concerne que les fichiers perceptibles (audio ou vidéo par exemple). Elle efface les données non perceptibles par l'humain, afin de gagner en taux de compression, et ne permet pas de retrouver les données originales après la décompression. Le format MP3, pour les fichiers audio, et JPEG pour les fichiers images utilisent des algorithmes de compression avec perte.

2. La compression sans perte (lossless)

La compression est dite sans perte si l'information après la décompression est exactement la même que celle avant la compression. On trouve par exemple la compression en FLAC pour les fichiers audio. Les deux algorithmes suivants sont des exemples d'algorithme de compression sans perte.

Le nombre de serveurs et de données stockés ne faisant qu'exploser ces dernières années, il nous faut trouver un moyen de réduire au maximum les coûts de stockage et la compression de données est un très bon moyen de palier ce problème. Nous allons donc nous intéresser à la capacité de compression ainsi qu'au temps d'encodage et de décodage de l'algorithme d'Huffman statique ainsi que celui d'Huffman dynamique, réputés pour être optimales en les décortiquant et les testant sur différents types de fichier. En premier temps, nous nous attaquerons l'algorithme d'Huffman statique, son fonctionnement et ses optimisations possibles suivis d'un exemple puis nous nous intéresserons à l'algorithme d'Huffman dynamique pour aussi comprendre son fonctionnement et ses optimisations possibles puis nous terminerons par une comparaison de leurs taux de compressions et temps d'exécution pour différents types de fichier.

L'ensemble du code est consultable sur GitHub, via le lien suivant : https://github.com/madpome/Comp_data_MI18

1 Huffman Statique

Le codage de Huffman Statique est un algorithme de compression sans perte, qui utilise un code à longueur variable pour coder les symboles utilisés. Le code de chaque symbole est déterminé par son taux d'apparition dans l'échantillon.

Si on note A l'alphabet utilisé dans un texte t , et A^* l'ensemble des mots finis formés sur A , alors on peut définir une fonction $f : A^* \rightarrow \{0, 1\}^*$, injective qui à chaque symbole, associe une suite de 0 et de 1 correspondant au code du symbole.

Cette fonction f vérifie deux propriétés :

Premièrement, c'est un morphisme, c'est-à-dire qu'il vérifie la propriété suivante : $\forall a, b \in A^*, f(ab) = f(a)f(b)$.

Deuxièmement, aucun code de f ne doit être préfixe d'un autre code. C'est-à-dire qu'il doit vérifier la propriété suivante : $\forall a, b \in A, f(a)$ n'est pas un préfixe de $f(b)$

De plus, on construit cette fonction f à l'aide d'un arbre binaire entier où chaque noeud contient une paire (c, p) , où c est un symbole (qui peut être vide), et p correspond au poids du noeud. Chaque feuille de cet arbre est le noeud d'un symbole, et chaque noeud qui n'est pas feuille contient le caractère vide. De plus, pour tout noeud N qui n'est pas une feuille, on a $N.p = \text{filsGauche}.p + \text{filsDroit}.p$ où filsGauche correspond au fils gauche de N et filsDroit à son fils droit.

L'algorithme nécessite alors deux lectures du texte. La première pour construire l'arbre, qui permet de connaître la fonction f , puis la seconde pour déterminer l'image du texte par f .

1.1 Fonctionnement

1.1.1 Compression

Soit T le texte à compresser. Soit Σ l'alphabet des symboles de T . Soit F un tableau de paires (c, p) tel que $F \subset \Sigma \times \mathbb{N}$. c correspond à un symbole de Σ , et p au nombre d'occurrences de c dans T . F est en réalité un ensemble

de nœuds qui sont tous les feuilles d'une forêt. La compression se déroule en 4 étapes :

1. Construction du tableau de fréquence
Lors de la première lecture, on va construire le tableau F , qui à chaque caractère de Σ associe sa fréquence dans T .
2. Construction de l'arbre à partir du tableau de fréquence
L'arbre est une transformation du tableau F , construit de la manière suivante: Tant que la longueur de F n'est pas 1 :
On trie F en fonction des poids, dans l'ordre croissant.
Soit n un nœud vide, n_1 le nœud de plus petit poids de F , et n_2 celui de $F \setminus \{n_1\}$. On initialise n avec n_1 comme fils gauche, n_2 comme fils droit, $n_1.p + n_2.p$ comme poids. Le caractère de n est le caractère vide.
On enlève n_1 et n_2 de F , et on y ajoute n .
3. Transmission de l'arbre de codage
Afin de permettre une décompression efficace (en temps), on transmet l'arbre de codage, avant d'écrire l'image du texte à compresser, en en-tête.
4. Calcul du texte compressé
On effectue la deuxième lecture du texte maintenant. Pour chaque caractère lu c , on parcourt l'arbre obtenu à l'étape précédente, pour chercher c , puis on écrit son code (qui correspond au chemin de la racine au nœud de c , avec 0 si on va dans le fils gauche, et 1 si on va dans le fils droit).

1.1.2 Décompression

Une fois la reconstruction de l'arbre effectuée, on procède à la décompression de la manière suivante : En partant de la racine, pour chaque bit lu, si le bit vaut 1, on descend dans le fils gauche, sinon, le bit vaut 0, et on descend dans le fils droit. Si on arrive dans une feuille, on lit le caractère du nœud, puis on revient à la racine.

1.2 Structure de données utilisée

L'algorithme de Huffman Statique utilisant principalement un arbre, une structure d'arbre binaire a naturellement été choisie. De plus, l'arbre binaire est rangé dans un tableau, car l'accès à chaque nœud est facilité. Chaque nœud contient l'indice du fils gauche, celui du fils droit, et le caractère qu'il contient si c'est une feuille.

1.3 Optimisation

Une première implémentation de l'algorithme nous faisait chercher le dernier caractère lu dans l'arbre, à chaque fois. Par exemple, si on lisait la lettre 'a', on allait chercher 'a' dans l'arbre pour avoir son code. Cependant cette méthode s'avérerait très peu efficace du fait que les caractères ne sont pas rangés dans un ordre précis dans l'arbre, mais en fonction du nombre d'occurrence, ce qui fait que la recherche pouvait se faire en $\theta(\text{Nombre de nœuds})$.

Cependant, comme l'arbre ne change au cours de la lecture (contrairement à l'algorithme suivant), le code des différents caractères ne change pas non plus. Ainsi, pour obtenir le code d'un symbole en $\theta(1)$, durant la deuxième lecture, il suffit de chercher le code de tous les caractères apparus, puis de les garder en mémoire dans un tableau liant un caractère lu, et son code.

1.4 Exemple

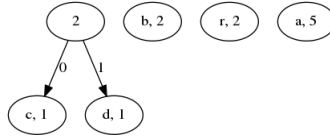
Supposons que l'on veuille compresser le texte **abracadabra**. Un premier passage nous donne le tableau de fréquences suivant :

symbole	d	c	r	b	a
fréquence	1	1	2	2	5

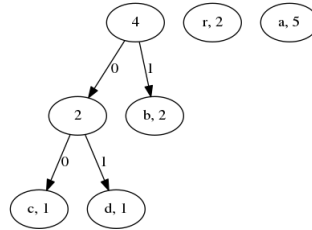
On présente ci-dessous les étapes de construction de l'arbre
Étape 1 :



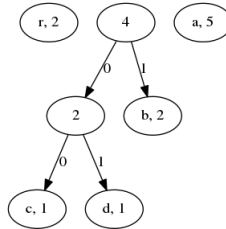
Étape 2 : Fusion des nœuds de c et d



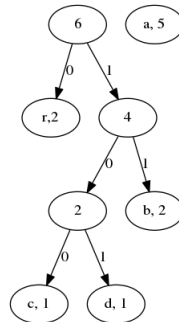
Étape 3 : Fusion des nœuds de b et de poids 2



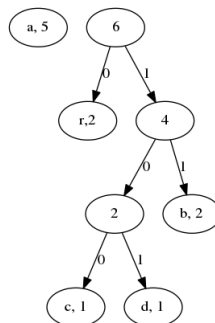
Étape 4 : Tri de la forêt dans l'ordre croissant des poids



Étape 5 : Fusion des nœuds de r et de poids 4



Étape 6 : Tri de la forêt dans l'ordre croissant des poids



Ainsi, l'arbre obtenu à la fin est le suivant (après fusion des nœuds de a et de poids 6) :

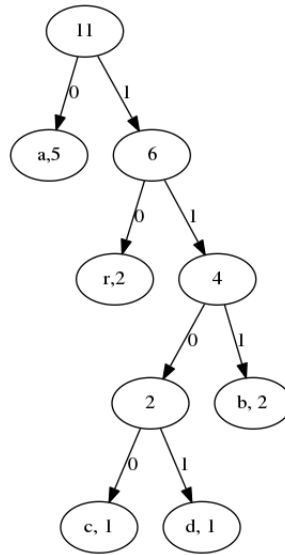


Figure 1: Arbre de Huffman Statique pour **abracadabra**

L'image de abracadabra par la fonction de compression est alors : 0 111 10 0 1100 0 1101 0 111 100 .

1.5 Spécification

Les fichiers compressés doivent avoir un format correct pour pouvoir être décompresser convenablement. Dans notre algorithme, un fichier compressé possède un en-tête qui contient 5 éléments :

1. Un nombre magique codé sur 4 octets, qui contient exactement la chaîne **HSMI**
2. Un nombre codé sur un octet, contenant le nombre de bits à lire sur le dernier octet. Ce nombre est naturellement compris entre 0 et 8.
3. Un nombre n correspondant au nombre de nœud de l'arbre, codé sur 2 octets, en big endian.
4. Un nombre codé sur 2 octets en big endian, correspondant à l'id de la racine de l'arbre.
5. On trouve enfin n blocs de 7 octets. Chaque bloc contient les données d'un nœud de l'arbre de codage. On trouve ces 4 éléments :
 - (a) l'id du nœud (compris entre 1 et n), codé sur 2 octets en big endian.
 - (b) la lettre contenue dans le nœud, codé sur 1 octet.
 - (c) l'id du fils gauche codé sur 2 octets en big endian (peut être égal à 0, si le nœud est une feuille)
 - (d) l'id du fils droit codé sur 2 octets en big endian (peut être égal à 0, si le nœud est une feuille)

À la suite de cet en-tête, on trouve une suite finie de bits contenant les chemins permettant la décompression du fichier.

L'algorithme de Huffman Statique comprend deux inconvénients majeurs, tout d'abord, le fichier d'entrée doit être lu deux fois, une fois pour déterminer l'arbre de codage, puis une seconde fois pour calculer l'image du texte de départ, par la fonction de compression. De plus, l'arbre de compression doit être transmis afin de procéder à la décompression. L'algorithme de Huffman Dynamique permet de palier à ces deux inconvénients.

2 Huffman Dynamique

L'algorithme de Huffman Dynamique est, tout comme le Huffman Statique, un algorithme de compression sans perte, mais qui contrairement au premier algorithme, utilise un code à longueur non variable (c'est-à-dire que chaque symbole est codé sur un nombre fixé de bits).

Tout d'abord, dans ce deuxième algorithme, l'arbre de codage, est modifié au cours du temps, et ne dépend que des symboles déjà lus, ainsi, contrairement à l'algorithme précédant, on peut l'appliquer sur un flux de données continu.

Ensuite, l'arbre de codage (qui est binaire entier), doit, à tout moment de l'algorithme, vérifier les deux propriétés suivantes :

Soit $n + 1$ le nombre de nœuds de l'arbre, $(c_i)_{i \in \llbracket 0, n \rrbracket}$ la suite de nœuds, et $p : \{c_i | i \in \llbracket 0, n \rrbracket\} \rightarrow \mathbb{N}$, qui à un nœud associe son poids.

1. La suite $(p(c_i))_{i \in \llbracket 0, n \rrbracket}$ est croissante.
2. Pour tout $i \in \llbracket 0, \frac{n}{2} \rrbracket$, c_{2i} et c_{2i+1} sont frères

Par exemple, l'arbre suivant est bien un arbre de Huffman :

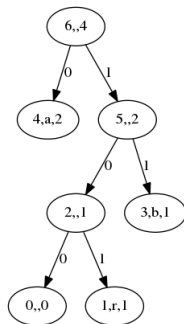


Figure 2: Exemple d'un arbre de Huffman

2.1 Fonctionnement

2.1.1 Manipulation de l'arbre

Les algorithmes de compression et de décompression utilisant le même arbre, les manipulations sont les mêmes dans les deux algorithmes. On trouve principalement 4 opérations sur ces arbres :

1. Initialisation de l'arbre

L'arbre est en premier lieu, initialisé avec un nœud vide, d'indice 0, et de poids 0.

2. Ajout d'un nouveau symbole

Soit A l'arbre courant.

Lors de la lecture d'un caractère k , s'il n'a jamais été rencontré auparavant, on l'ajoute dans l'arbre de la manière suivante :

Soit d_0 un nouveau nœud, qui est une feuille, de poids 0, et de caractère vide.

Soit d_1 un nouveau nœud, qui est une feuille, de poids 1, et de caractère k .

Soit d_2 un nouveau nœud, de poids 0, et de caractère vide, qui a pour fils gauche le nœud d_0 et pour fils droit le nœud d_1 .

On remplace alors le nœud c_0 par le nœud d_2 dans l'arbre A . La suite $(c_i)_{i \in \llbracket 0, n \rrbracket}$ est alors actualisée (on augmente l'indice de tout les nœuds de 2). On procède ensuite à un rééquilibrage à partir du nœud d_2 .

L'illustration suivante est un exemple d'ajout :

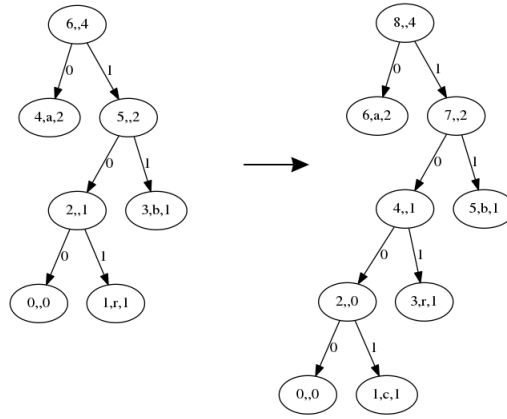


Figure 3: Ajout du caractère c dans l'arbre de gauche

3. Cas d'un caractère déjà lu

Si le caractère lu est déjà apparu dans le texte auparavant, alors il se trouve dans l'arbre, on lance alors une procédure de rééquilibrage à partir de son nœud.

4. Rééquilibrage

Un rééquilibrage sert à conserver la première propriété des arbres de Huffman. On effectue un rééquilibrage uniquement après avoir ajouté un caractère, ou lorsque l'on veut incrémenter le poids d'un nœud. Pour cela, on effectue la procédure suivante :

Soit c_i le nœud depuis lequel on commence le rééquilibrage.

Tant que c_i n'est pas la racine de l'arbre :

On cherche si il existe un $j \geq i$, le plus grand possible, tel que le $p(c_i) \leq p(c_j)$, et tel que c_j n'est pas le père de c_i .

Si un tel j existe, on échange les valeurs de c_j et de c_i .

On incrémente alors la valeur du poids de c_i .

Puis on passe au père de c_i .

L'exemple suivant montre un rééquilibrage :

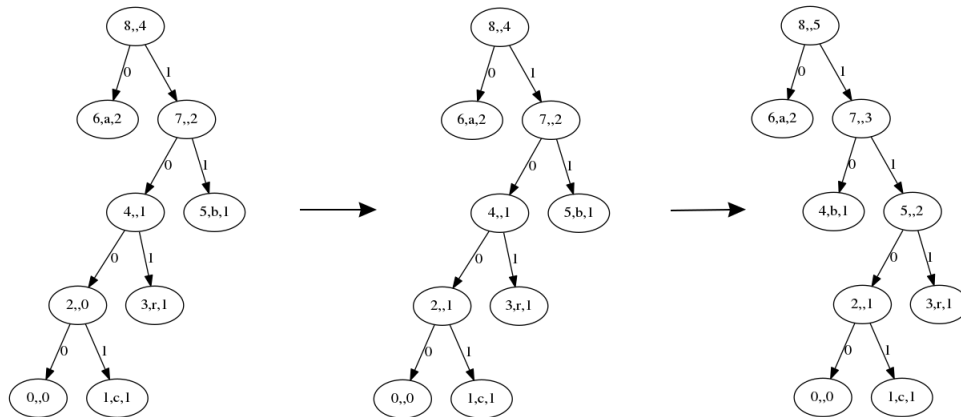


Figure 4: Exemple de rééquilibrage après ajout de c

2.1.2 Compression

À chaque lecture de caractère, si le caractère est un nouveau symbole, on écrit le chemin de la racine, au nœud c_0 , puis on écrit le caractère lu. Si le caractère a déjà été rencontré auparavant, on écrit le chemin de la racine au

nœud du caractère.

2.1.3 Décompression

La décompression se déroule de la manière suivante :

On se place à la racine de l'arbre.

Tant qu'on a des bits à lire :

On lit un bit, si il vaut 1, on descend dans le fils gauche, sinon on descend dans le fils droit.

Si on n'est pas dans une feuille, on continue la lecture et l'avancée dans le parcours de l'arbre. Sinon, on est dans une feuille, et alors :

Si on est dans la feuille vide (on est dans c_0), on lit un caractère, pour obtenir le caractère codé, et on le rajoute à l'arbre, puis on procède à un rééquilibrage, par rapport au nœud c_2 (le nouveau père du nœud c_0).

Sinon, on est dans une feuille qui contient un caractère, on lance un rééquilibrage sur cette feuille, et on écrit le caractère.

2.2 Structures de données utilisées

L'algorithme de Huffman Dynamique utilisant principalement un arbre binaire, une structure d'arbre binaire a naturellement été choisie. De plus, l'arbre binaire est rangé dans un tableau, car l'accès à chaque nœud est facilité. Chaque nœud contient un caractère si c'est une feuille, les distances entre l'indice du nœud courant, et l'indice du fils gauche, du fils droit, et du père, un poids (qui correspond au nombre de caractère du nœud déjà lu). Le rangement dans un tableau facilite aussi la gestion des indices pour chaque nœud, et la recherche d'un nœud de poids égale, d'indice plus grand. Garder la distance nous permet d'éviter de mettre à jour les pointeurs vers les fils gauches, et droits, des nœuds fils d'un nœud qui a été échangé avec un autre durant un rééquilibrage.

Une deuxième structure de données utilisée est un tableau d'indice inverse, de taille 256 (une case pour chaque valeur ASCII possible), qui a un caractère, associe deux attributs :

1. l'indice du nœud de l'arbre, qui contient le caractère, utilisé pour accéder au nœud en temps constant, et pour savoir si le caractère est un nouveau caractère ou non.
2. le chemin de la racine, au nœud précédemment cité. On conserve un tel chemin, afin d'éviter de le recalculer à chaque fois qu'on rencontre un caractère.

2.3 Optimisation


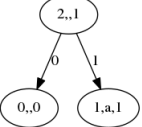
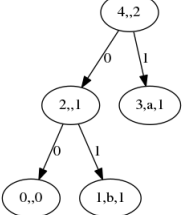
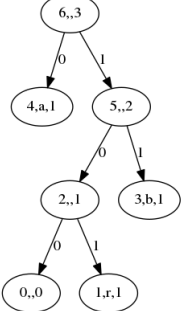
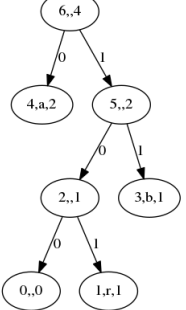
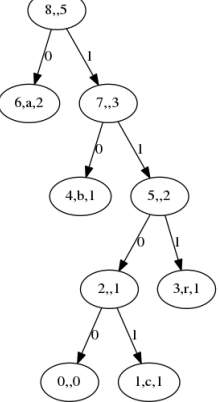
Dans une première implémentation de l'algorithme, un tableau contenant les caractères déjà rencontrés était maintenu à jour. Cependant, pour pouvoir utiliser ce tableau pour un caractère c , il fallait effectuer une recherche linéaire, donc en $\theta(\text{Nombre de caractères différents})$. De plus, si on a déjà rencontré le caractère auparavant, on doit écrire le chemin de la racine, au nœud contenant ce caractère, et lancer un rééquilibrage, ce qui demande alors de parcourir l'arbre, jusqu'à trouver le bon nœud, et donc une seconde recherche dans l'arbre, en $\theta(\text{Nombre de nœud de l'arbre})$.

Une optimisation possible est de créer un tableau associatif, de 256 cases (le nombre de caractères ASCII possibles), qui a un char c , associe une valeur particulière, si c n'a jamais été vu, ou un pointeur sur le nœud et le chemin demandé sinon, on économise alors les deux recherches, passant alors d'un $\theta(\text{Nombre de nœud de l'arbre} + \text{Nombre de caractères différents})$ à un $\theta(1)$. Ce tableau est alors mis à jour à chaque rééquilibrage, en cas de besoin.

De plus, la recherche d'un père de poids égale au poids du nœud courant (qui est la recherche la plus effectuée) s'effectuait en temps linéaire avant, pour ensuite passer à une recherche dichotomique (Les poids étant triés dans l'ordre croissant).

2.4 Exemple

Les étapes du codage de **abracadabra** sont les suivantes :

Étape	Code généré	Arbre	Description
1			Initialisation
2	a		Ajout de a
3	0b		Ajout de b
4	00r		Ajout de r
5	0		Incrémentation de a
6	100c		Ajout de c

7	0		Incrémentation de a
8	1100d		Ajout de d
9	0		Incrémentation de a
10	111		Incrémentation de b

11	110		Incrémentation de r
12	0		Incrémentation de a

2.5 Spécification

Tout comme pour l'algorithme précédant, les fichiers compressés doivent respecter un certain format pour pouvoir être décompressé. Pour cet algorithme, chaque fichier possède un en-tête, contenant la chaîne **HDMI** codé sur 4 octets. À la suite de cet en-tête, on trouve une suite finis de bits, correspondant au codage du fichier original.

L'algorithme devant fonctionner sur un flux de données, la création d'un caractère de fin de fichier est nécessaire. Cependant, un fichier quelconque peut contenir tout les caractères codables sur 8 bits (soit 1 octet). C'est pour cela qu'au lieu de coder nos caractères sur 1 octet, on le code sur 2. Le premier octet vaut **11111111** si le caractère correspond au caractère de fin de fichier, et **00000000** sinon. De plus, pour savoir qu'il faut lire un caractère (donc 2 octets) lors de la décompression, on a besoin du chemin de la racine, au nœud de poids 0. C'est pour cela qu'une fois que la compression du fichier d'origine est terminée, on écrit le chemin de la racine de l'arbre de codage, au nœud de poids 0, puis le caractère **11111111 00000000**.

3 Pseudo-code

3.1 Huffman Statique Compression

Algorithm 1 Compression Huffman Statique Construction de l'arbre

Require:

T : le texte à compresser
 F : le tableau des fréquences (vide)

```
while  $X \neq EOF$  do
   $X \leftarrow litUnCaractere(T)$ 
  if  $F$  contient  $X$  then
     $F[X].p \leftarrow F[X].p + 1$ 
  else
     $F \leftarrow F \cup \{(X, 1)\}$ 
  end if
end while
while longueur de  $F \neq 1$  do
   $F \leftarrow tri(F)$ 
   $n \leftarrow nouveauNoeud$ 
   $n.p \leftarrow F[0].p + F[1].p$ 
   $n.fg \leftarrow F[0]$ 
   $n.fd \leftarrow F[1]$ 
   $F \leftarrow (F \setminus \{F[0], F[1]\}) \cup \{n\}$ 
end while
```

Algorithm 2 Compression Huffman Statique Codage du texte

Require:

T : le texte à compresser
 F : l'arbre de codage

```
while  $X \neq EOF$  do
   $X \leftarrow litUnCaractere(T)$ 
   $C \leftarrow codeDe(X)$  dans  $F$ 
  Écrit  $C$ 
end while
```

3.2 Huffman Statique Décompression

Algorithm 3 Décompression Huffman Statique

Require:

F Arbre des fréquences

T suite finie de bits

$N \leftarrow F.root$

while Il reste des bits à lire **do**

$X \leftarrow litUnBit(T)$

if $X = 0$ **then**

$N \leftarrow N.fg$

else

$N \leftarrow N.fd$

end if

if N est une feuille **then**

 Écrit $N.c$

$N \leftarrow F.root$

end if

end while

3.3 Huffman Dynamique Compression

Algorithm 4 Compression Huffman Dynamique

//la fonction reequilibre renvoie à la fonction définie en dessous

Require:

T texte à compresser

$A \leftarrow initArbre()$

while $X \neq EOF$ **do**

$X \leftarrow litUnCaractere(T)$

if $dejaVu(A, X)$ **then**

$Nx \leftarrow getNoeud(A, X)$

 ÉcritLeCheminDeLaRacineA(Nx)

 reequilibre(Nx, A)

else

$c_0 \leftarrow getNoeudDePoids0(A)$

 ÉcritLeCheminDeLaRacineA(c_0)

 Écrit(X)

 ajouteDans(A, X) //Peut être vide, ajouteDans rajoute le caractère X dans l'arbre, et renvoie le père du nœud de poids 0

 reequilibre(c_0, A) // c_0 qui était l'ancien nœud de poids 0, est maintenant son père après l'ajout du caractère X

end if

end while

Algorithm 5 Algorithme de rééquilibrage de l'arbre

Require:

n nœud de départ pour le rééquilibrage

A arbre

while $n \neq \text{root}(A)$ **do**

if Il y a un nœud c_k d'indice plus haut que n && de poids égal && qui n'est pas son père **then**

$\text{echange}(X, c_k)$

$\text{incrementPoids}(n)$

$n \leftarrow \text{Pere}(n)$

end if

end while

$\text{incrementPoids}(n)$

3.4 Huffman Dynamique Décompression

Algorithm 6 Décompression Huffman Dynamique

Require:

T texte à décompresser

$A \leftarrow \text{initArbre}()$

$N \leftarrow \text{racine}(A)$

while Il reste des bits à lire **do**

$X \leftarrow \text{litUnBit}(T)$

if $X = 0$ **then**

$N \leftarrow N.fg$

else

$N \leftarrow N.fd$

end if

if $\text{isLeaf}(N)$ **then**

if $N.c = \emptyset$ **then**

$c \leftarrow \text{litUnCaractere}(T)$

$\text{ajouteDans}(A, c)$

$\text{ecrit}(c)$

else

$\text{ecrit}(N.c)$

end if

$\text{reequilibre}(N, A)$

$N \leftarrow \text{racine}(A)$

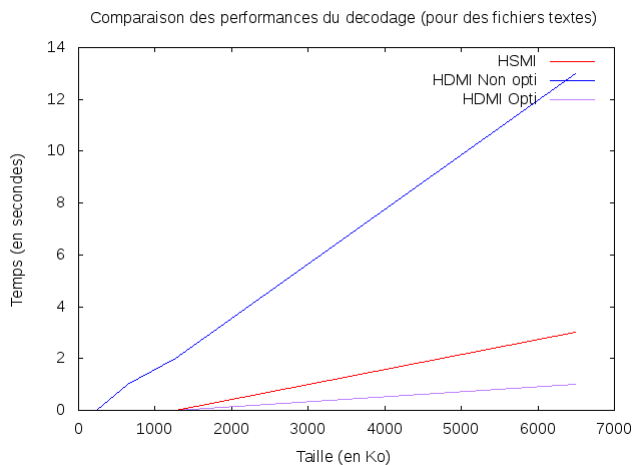
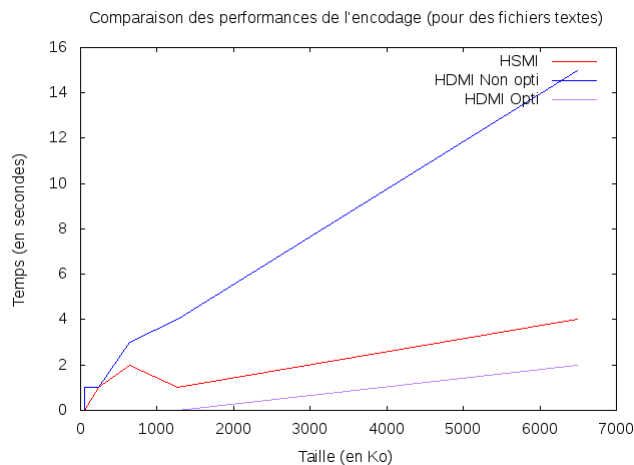
end if

end while

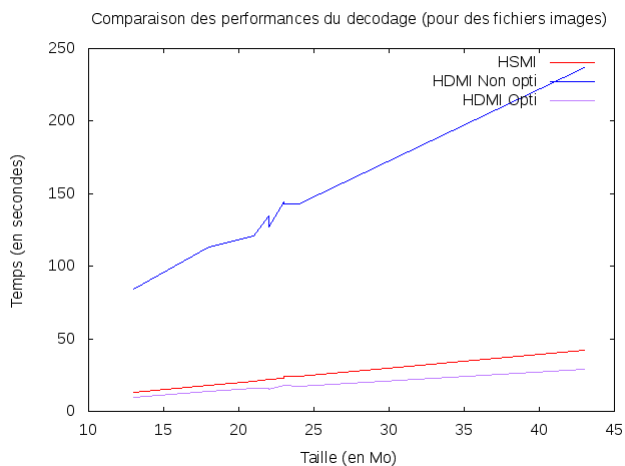
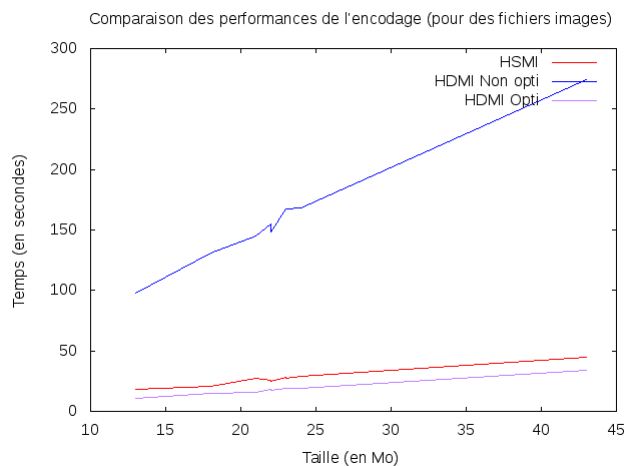
4 Performances

Pour l'ensemble des graphiques suivants, **HSMI** désigne l'algorithme Huffman Statique, **HDMI Non opti** désigne l'algorithme Huffman Dynamique sans optimisation, et enfin, **HDMI Opti** renvoie à l'algorithme Huffman Dynamique, après optimisation.

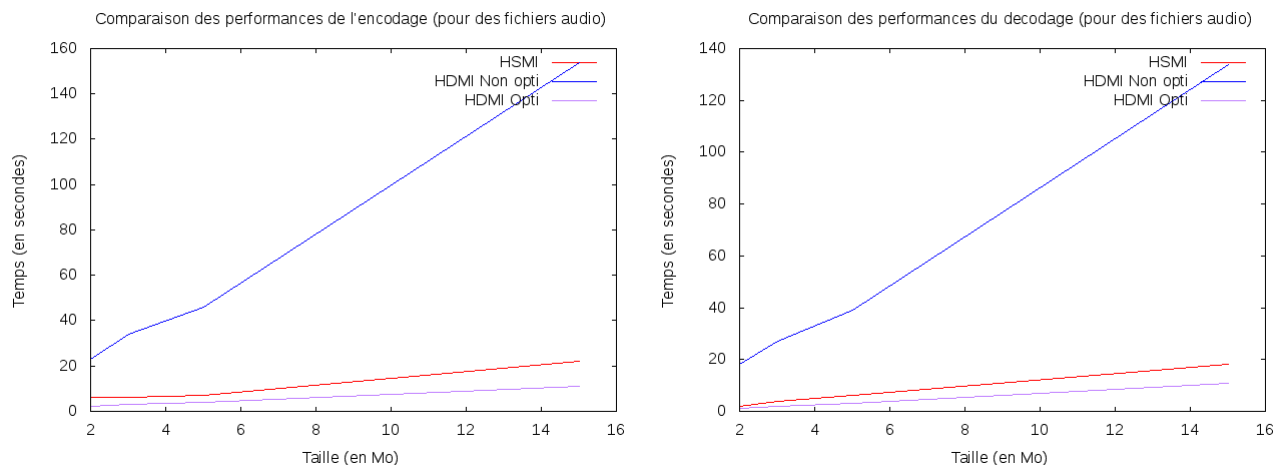
4.1 Fichiers texte



4.2 Fichiers image

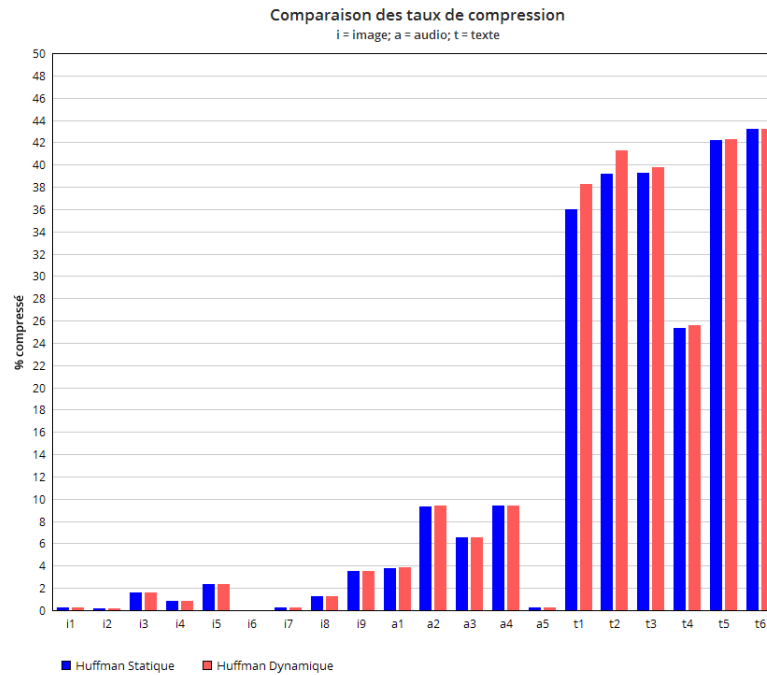


4.3 Fichiers audio



4.4 Comparaison des taux de compression

Nom du fichier	Type du fichier	Taille du fichier (Ko)
i_1 : <i>pinngd_mlc</i>	nef	13956
i_2 : <i>rnmyic</i>	dng	18514
i_3 : <i>uzkoci_raspberry</i>	nef	21236
i_4 : <i>juriart</i>	cr2	22512
i_5 : <i>stsci</i>	tif	22612
i_6 : <i>emdde7</i>	raf	23305
i_7 : <i>elbebe1991</i>	cr2	23811
i_8 : <i>mr.wolf</i>	dng	24810
i_9 : <i>stci2</i>	tif	43460
a_1 : <i>muscle</i>	wav	2228
a_2 : <i>airplane</i>	wav	2494
a_3 : <i>swimming</i>	wav	3700
a_4 : <i>car</i>	wav	5594
a_5 : <i>rain</i>	wav	15245
t_1 : <i>amerikasta</i>	txt	46
t_2 : <i>ourfriendthedog</i>	txt	49
t_3 : <i>florante</i>	txt	237
t_4 : <i>chinois</i>	txt	641
t_5 : <i>mobydick</i>	txt	1270
t_6 : <i>sherlock</i>	txt	6488



Sur ce graphique, on voit que les algorithmes sont très peu efficace pour les fichier image et audio, car les images prises en test ont toutes des formats de fichier compressés. Enfin, on voit que la compression sur des fichiers textes est d'environ 40% sauf pour le 4ème fichier. Cette différence s'explique par le fait que ce fichier est un texte en chinois, codé en UTF-8. Or l'UTF-8 utilise un code à longueur variable pour encoder les caractères. De plus, les symboles encodables sur 1 octet en UTF-8 correspondent exactement à ceux de la table ASCII. Donc tout les caractères usuels sont codés sur 1 octet, contrairement aux caractères chinois, qui sont codés sur plusieurs octets. Une solution pour obtenir un bon taux de compression serait de lire le bon nombre d'octet à chaque lecture de caractère.

5 Liste des outils utilisés

Les graphes d'arbres ont été générés à l'aide du logiciel GraphViz.
Les différents graphiques ont été conçus grâce à l'outil gnuplot

Conclusion

Comme nous avons pu le constater, les deux versions de l'algorithme de Huffman sont équivalentes dans leurs versions optimisées, aussi bien en temps qu'en taux de compression. Les algorithmes fonctionnent en temps linéaire et la complexité ne change pas ou peu en fonction du type de fichier.

Néanmoins, les taux de compressions ne sont correctes que pour des fichiers textes qui sont de l'ordre de 40% ce qui s'explique par le fait que les fichiers audio et image soient compressés. Ces algorithmes ne sont donc que très peu efficaces sur des fichiers quelconques, et ne sont pas adaptés à une utilisation à grande échelle.

Bibliographie

L'ensemble des fichiers textes ont été pris sur le site : <http://www.gutenberg.org/>.

Les fichiers audio proviennent de : <http://soundbible.com/tags-raw.html>

Les images utilisés pour les tests proviennent de :

1. <https://www.wesaturate.com/>

2. http://hubblesite.org/get_involved/hubble_image_processors/

<http://www2.ift.ulaval.ca/~dadub100/cours/H16/4003/02Acetates2.pdf>

<http://utbm2004.free.fr/L042/codage%20de%20huffman%20-%20%E9nonc%E9.pdf>

https://en.wikipedia.org/wiki/Huffman_coding

C.Choffrut, Université Paris 7, Codage de Huffman