République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université A/Mira de Béjaïa

Faculté des Sciences Exactes

Département d'Informatique

Mémoire de fin de cycle

En vue de l'obtention du diplôme de master recherche en Informatique

Option : Réseaux et Systèmes Distribués

Thème

Compression de données

Mémoire soutenu le 02/07/2016 par :

M^r ***TOUNSI*** **Billal**

M^r ***ZIDOUNE*** **Halim**

Devant le jury composé de :

Président :	M^{me} ***BATTAT*** **Nadia**	**MAA , U.A.M Béjaïa**
Examinateur :	M^r ***ALOUI*** **Abdelouhab**	**MCA , U.A.M Béjaïa**
Encadreur :	M^r ***AMROUN*** **Kamal**	**MCA , U.A.M Béjaïa**

Promotion 2015/2016

Remerciements

Tout d'abord, nous remercions Dieu le tout-puissant qui nous a donné le courage, la force et la volonté pour mener ce travail.

Un grand merci pour nos familles, surtout nos parents qui nous ont épaulés, soutenus et suivis tout au long de ce projet.

A nos chères amis qui ont toujours été présents et fidèles.

A notre encadreur ***Mr. AMROUN Kamal*** *pour tout le temps qu'il nous a consacré, pour ces précieux conseils et pour toute son aide et son appui durant la réalisation.*

Aussi à tous les enseignants et employés du département Informatique à qui on doit notre avancement.

Enfin, nous tenons aussi à remercier également tous les membres du jury pour avoir accepté d'évaluer notre travail.

Dédicaces

Je dédie ce modeste travail à mes tres chers parents qui ont toujours etaient la pour moi et qui m'ont setenu .

À mes sœurs et a mon frère .

À tous mes amis proches et a tous mes amis .

Billal TOUNSI

Je dédie ce modeste travail à mes aimables et respectueux parents qui ont toujours veillés sur mes études.

À mes chers frères et à ma sœurs .

À tous mes amis proches et à tous mes amis.

Halim ZIDOUNE

Table des matières

Table des figures

Liste des tableaux

Liste des Algorithmes

Liste des abréviations

RLE	**R**un **L**ength **E**ncoding
JPEG	**J**oin **P**hotographic **E**xperts **G**roup
MTF	**M**ove **T**o **F**ront
BWT	**B**urrows **W**heeler **T**ransform
LZ	**L**empel **Z**iv
LZW	**L**empel **Z**iv **W**elch
RGB	**R**ed **G**reen **B**lue
RVB	**R**ouge **V**ert **B**leu
YCbCr	**Y** représente la luminance **C**omposante bleu **C**omposante rouge
DCT	**D**iscrete **C**osines **T**ransform
API	**A**pplication **P**rogramming **I**nterface
POI	**P**oor **O**bfuscation **I**mplementation
HSSF	**H**orrible **S**preadSheet **F**ormat
OLE2	**O**bject **L**inking and **E**mbedding

Introduction générale

Dans le monde numérique et du partage d'information, la vitesse de transmission est primordiale. La compression de données est donc de plus en plus abordée. Elle vise à réduire le nombre de bits comportant l'information à transmettre ou à stocker.

Les méthodes de compression utilisées aujourd'hui sont le fruit d'une longue histoire d'évolution qui a connu la compression de données, la première approche définie la compression d'un message M comme l'algorithme le plus courts qui peut générer M. Pour M=111...11 de taille n son compressée est le code source suivant :

Comp (11...11)= " for (i=0 ; i< n ; i++) écrire 1 ".

Mais ce mécanisme ne fonctionne pas avec des messages de la forme 01001110101, ce qui a conduit à la recherche d'autres approches. Comme les méthodes a base de dictionnaire.

La première méthode basée sur le principe d'utilisation des dictionnaires est celle de Bigame, son principe est de créer un dictionnaire qui contient l'ensemble des mots de taille 2. Le fonctionnement de cette méthode est le suivant : soit k le nombre de bits nécessaire pour représenter une lettre d'un alphabet A (k=8 si on travaille avec le code ascii), et n le nombre de symbole utilisé pour représenter une séquence (sans calculer la redondance) tel que : 2puiss(k-1)<n<2puis(k). Nous avons donc 2puiss(k)-n symbole disponible pour utiliser k bits qui sont réservés pour les bigames[1]. Une autre méthode proposée par PIKE [2], et qui a proposé un schéma où :

- les symboles les plus fréquents tiens 4 bits.
- les symboles moins fréquents tiens 8 bits.
- les symboles les plus rares tiens 12 bits.

Une méthode similaire à la méthode de PIKE est proposée par TROPPER où les mots sont découpés en préfixes, radicaux, et suffixes communs [3].

Ces techniques ne sont pas adaptées à toutes les langues, ce qui a poussé les chercheurs à trouver d'autres solutions comme les codes de Huffman, Shannon que nous avons traités dans notre travail.

L'objectif de notre travail est l'étude de la compression de données en général,

celle sans perte en particulier et les différents algorithmes existants. Puis essayer d'effectuer une amélioration à un de ces algorithmes, notre choix c'est porté sur celui de compression RLE (Run-Length-Encoding). Le point fort de RLE réside dans le fait qu'il élimine la redondance successive d'un symbole dans une séquence de symbole. Ce dernier est appliqué dans la norme de compression d'image JPEG (Joint-Photographic-Expert-Group) basé sur la transformation d'image. Cette méthode est très populaire, en particulier sur internet où la compression d'un fichier permet de réduire le coût en bande passante, en plus elle permet d'enregistrer une image dans le format JPEG avec la majeur partie des appareils photo numériques et téléphones portables. Des tests comparatifs entre RLE et RLE+ ont été effectués dans ce travail sous java, le résultat obtenu montre que RLE+ gagne plus que RLE en termes d'espace mémoire surtout dans le cas où la qualité d'image est réservée.

Afin de réaliser ces objectifs, notre travail est structuré autour de quatre chapitres :

Le premier chapitre est réservé à des généralités sur la compression de données et les différents concepts utilisés dans le domaine de la compression.

Comme la compression de données a un grand rapport avec le codage nous allons vous présenter dans le deuxième chapitre quelques techniques de codage et algorithmes de compression existants ainsi que leurs avantages et leurs inconvénients.

Le troisième chapitre, traite l'amélioration apportée à l'algorithme RLE ainsi que les outils que nous avons développés afin d'implémenter l'algorithme RLE+ et de voir les résultats.

Le dernier chapitre, est consacré à une étude comparative entre l'algorithme de compression RLE et RLE+.

Chapitre 1

Généralités sur la compression de données

1.1 Définition de la compression

La compression des données, d'une manière générale c'est l'ensemble des méthodes et règles qui permettent de réduire le volume d'une donnée sans perdre les informations éssentielles. D'une manière plus simple, c'est une technique où on emploie une paire de fonctions, une d'elle a pour but de compresser les données et l'autre de les décompresser.

Le but de la compression est de représenter les informations avec une forme plus compacte que l'original où le résultat de la compression occupe moins d'espace que la donnée originale. Les données peuvent être compressées avec perte ou sans perte [4].

1.2 Type de compression

1.2.1 Compression sans pertes

Dans ce type de compression, on a chaque bit qui est soumis à la compression est restitué avec exactitude lors de l'étape de la décompression. Un ensemble de bits X a comme résultat de compression le compressé Y plus court que X, ce résultat est stocké ou transmis. Lors de la décompression du résultat Y, on récupèrera par exactitude l'ensemble de bits de départ X [4].

Schématiquement parlant on a :

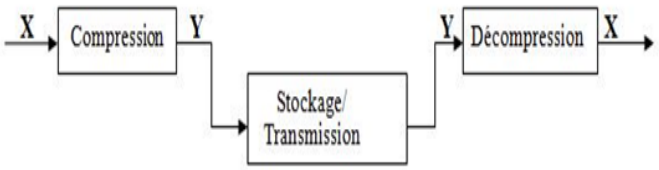


FIGURE 1.1 – Compression sans pertes

1.2.2 Compression avec pertes

Ce type de compression a un rôle tout à fait différent par rapport à celui de sans perte. Le principe de la compression avec perte est d'avoir une permission de supprimer quelques données (qui sont inutiles) de l'information pour avoir une meilleure compression, ce qui signifie qu'après une décompression du compressé on aura un résultat approximatif du fichier original. On a une information X qu'on veut compresser, son résultat de compression est Y, après décompression du compressé Y on a un résultat Z qui est une approximation de X (Z=X). On utilise souvent cette compression pour compresser les données comme image, son, vidéo,. . ., car pour ce type de données on peut enlever quelques données sans toucher aux informations essentielles qui sont contenues dans ce type de fichier [4].

Voici un schéma représentatif d'une compression avec perte :

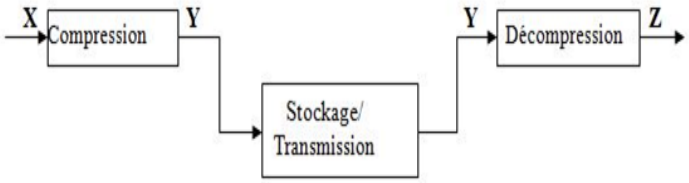


FIGURE 1.2 – compression avec pertes.

1.3 Compression d'image

Les méthodes de codage et de compression d'image cherchent à réduire le nombre de bits par pixel à stocker ou à transmettre, en exploitant la redondance informationnelle dans l'image. Comme les méthodes de compression déjà citées dans les sections précédantes, les méthodes de compression d'image ont des critères d'évaluation, les principaux critères sont [5] :

- La qualité de reconstitution de l'image.
- Le taux de compression.
- La rapidité du codeur.

Les méthodes de compression d'image suivent généralement le schéma suivant :

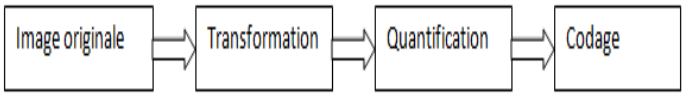


FIGURE 1.3 – Schéma d'un codeur d'image

Le résultat sera un fichier code, en appliquant les étapes dans le sens inverse on obtient une image presque identique à l'image originale.

Une méthode de compression d'image peut être avec ou sans perte. Avec une méthode sans perte, l'image obtenue après le décodage est identique à l'image originale, dans l'autre cas l'image obtenue rassemble l'image originale avec une distorsion qui ne doit pas dépasser une certaine limite.

1.4 Caractéristique d'un algorithme de compression

Dans les premières étapes de l'évolution de la théorie de l'information, on trouve que les méthodes de mesure et de comparaison entre les algorithmes se divisent en deux parties, l'une est la complexité de Kolmogorov qui juge un algorithme selon l'espace occupé par une fonction qui génère une séquence de symbole [4].

Exemple la séquence de 1 :111...11 n fois, peut être générée par une boucle :
For (i = 0 ; i < n ; i++) ecrire (1).

• Avec la complexité stochastique, la complexité et l'efficacité d'un algorithme se mesurent par :

– Taux de compression : qui est le rapport entre la taille du compressé et la taille de la séquence originale :

T=taille du compressé/taille originale.

Plus cette variante est plus grande, plus le programme est meilleur.

– Le temps de compression : est la durée nécessaire pour produire un compressé à partir d'un message originale en appliquant un algorithme de compression.

• Complexité de la méthode utilisée et de sa mise en œuvre : la complexité d'une méthode de compression peut être définie par le temps d'exécution ou par le nombre d'opération nécessaire à la compression d'une séquence de données[9].

1.5 Concepts généraux

• **Séquence :** une séquence est une suite ordonnée et finie de symboles tirés d'un alphabet de référence[6].

• **Longueur d'une séquence :** la longueur d'une séquence S est notée |S| , donne le nombre de symboles qui composent S[6].

• **Représentation des séquences :** soit B= {0,1} l'alphabet binaire et A alphabet, toute séquence formée à partir d'un alphabet A se présente par une séquence formé à partir de$B+$. Par déduction tout symbole de tout alphabet est représentable par une chaine de bits[6].

• **Information propre :** soit une séquence S composée d'un alphabet de M symboles S= Ai, L <= i <= M de probabilités P {(ai)=Pi, L <= i <= M}. Alors, l'information propre Li associée au ieme symbole est définie comme suite :
Li=log (L/ Pi)=-log (Pi) (bits)[6].

• **Code préfixe :** Un code a la propriété du préfixe si aucun mot de code n'est le préfixe d'un autre mot de code. Exemple : L'ensemble <0 ; 10 ; 11> est le produit

d'un code préfixe valide. L'ensemble <0 ; 1 ; 10 ; 11> n'est pas le produit d'un code préfixe valide, car les mots de code 1 et 10 sont tous deux préfixes de la séquence de bits 1011001100.., par exemple[6].

- **Entropie d'une séquence :** L'entropie d'une séquence est le nombre moyen de bits nécessaires pour coder les symboles de cette séquence. Supposons qu'une séquence contient M symboles de probabilité Pi et

$$\sum Pi = 1[6]. \tag{1.1}$$

Supposons aussi que l'occurrence d'un symbole est indépendante des autres symboles ce type appelé source sans mémoire. L'entropie d'une séquence est définie comme :

$$H = -\sum_{i=1}^{M} Pi * log_2(Pi)(bits/symbole)[6]. \tag{1.2}$$

Chapitre 2

Techniques de codage et algorithmes de compression

2.1 Définition du codage

Le codage est une étape essentielle dans la compression, c'est une méthode de transformation qui convertit la représentions d'une information en une autre [7]. Il existe plusieurs techniques de codage on citera shannon-fano, huffman,...

2.2 Techniques de codage

2.2.1 Code de Shannon-Fano

2.2.1.1 Définition

Le codage de Shannon-fanon a été proposé en 1948 par Claude Shannon dans l'article 'A Mathematical Theory of Communication'. Le principe de ce codage consiste à créer des codes préfix en se basant sur un ensemble de symboles et sur leurs probabilités respectives[8].

2.2.1.2 algorithme

L'algorithme pour produire un code de Shannon-Fano pour un alphabet A est le suivant [8] :

– Trier les symboles de l'alphabet A en ordre décroissant de fréquence d'apparition (du plus fréquent au moins fréquent).
– Diviser A en deux sous-ensembles A1 et A2 en respectant les deux contraintes suivantes :

  • A1 doit contenir les n plus fréquents symboles de A et A2 doit contenir les |A|-n moins fréquents symboles de A.

  • La déférence entre la somme d'apparition des symboles de A1 et la somme d'apparition du symbole de l'ensemble A2 doit être la plus petite possible. Formellement, nous cherchons les ensembles A1={s1, s2, ..., sj1} et A2= {sj, sj+1, ..., sn} tels que :

$$\sum_{i=0}^{i<j} Si - \sum_{i=j}^{i<=n} Si = 0 \tag{2.1}$$

– On Attribue aux symboles dans A1 le bit 0 comme premier bit de leurs mots de code et le bit 1 aux symboles dans A2
– Répéter l'algorithme récursivement, à partir de l'étape 2, sur chacun des ensembles jusqu'à ce qu'il reste qu'un symbole dans chaque ensemble. A chaque fois qu'un ensemble est divisé, les symboles appartenant à cet ensemble se voient attribuer un bit supplémentaire à leurs mots de code.

2.2.1.3 Exemple

Soit la figure suivante qui représente la fréquence d'apparition des symboles d'un alphabet A= A, B, C, D, E [7]. Les statistiques sont extraites à partir d'un texte :

Symbole	A	B	C	D	E
Apparitions	15	7	6	6	5
Fréquence	15 /39	7 /39	6 /39	6 /39	5 /39

TABLE 2.1 – tableau d'apparitions des symboles d'un texte

L'application de l'algorithme nous donne l'arbre suivant [7] :

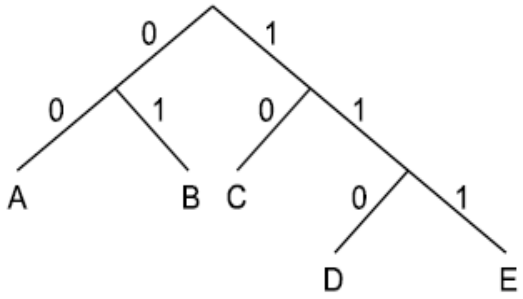


FIGURE 2.1 – application de l'algorithme de shannon-fano

A partir de l'arbre on peut extraire le code de chaque symbole. Ce qui nous donne le tableau suivant [7] :

Symbole	Mot de code
A	00
B	01
C	10
D	110
E	111

TABLE 2.2 – code de Shannon-Fano

2.2.1.4 Avantages et inconvénients

- **Avantages :**
  - Produit des codes préfix.
  - Réduction importante dans la taille du fichier a compresser.
- **Inconvénients :**
  - la taille des codes augmente avec le nombre de symboles qui existent dans l'alphabet utilisé.

2.2.2 Code de Huffman

2.2.2.1 Définition

La méthode de Huffman consiste à trouver le nombre d'occurrences de chaque caractère, puis d'attribuer un code plus court aux caractères les plus fréquents, et un code plus long pour les caractères les moins fréquents. Pour ce faire, un arbre de Huffman est utilisé (voir la prochaine section). Ensuite, pour que le fichier puisse être décompressé, on devra soit envoyer l'arbre de Huffman en en-tête du fichier compressé, soit envoyer chaque caractère avec son nombre d'occurrences, ce qui permettra de reconstruire l'arbre [9].

2.2.2.2 Principe de l'algorithme de Huffman

- Pour implémenter l'algorithme de Huffman, on aura besoin d'un arbre binaire et d'une file d'attente.
- Chaque nœud de l'arbre a une fréquence d'apparition dans une séquence de symbole.
- Les feuilles de l'arbre représentent des symboles avec leur fréquence d'apparition dans une séquence. La fréquence d'un nœud interne égale à la somme des fréquences de ces fils.

• **Algorithme**

- Ordonner les symboles par probabilité décroissante.
- En partant du bas, affecter '0' au symbole le moins probable et '1' au suivant.
- Supprimer ces symboles de la liste et les combiner en un symbole composite. La probabilité de ce symbole est la somme des probabilités initiales des symboles.
- Réordonner la liste
- répéter 2 - 4, tant qu'il reste au moins deux symboles dans la liste.

Une fois l'arbre construit, nous sommes en mesure d'obtenir le mot de code de chacun des symboles de l'alphabet. Pour ce faire, il suffit simplement d'accumuler les bits dans l'ordre racine $->$ feuille (chacune des feuilles représente un symbole). On peut donc retrouver le mot de code associé à un symbole en temps linéaire au nombre de bits qu'il contient[7].

Il existe une autre variante du codage de Huffman qui est le codage de Huffman adaptatif qui ne requiert pas de connaitre à l' avance les fréquences d'apparition des symboles.

2.2.2.3 Exemple

Si on prend l'exemple de la section précédente. On obtient comme résultat de l'application du code de Huffman l'arbre suivante [7] :

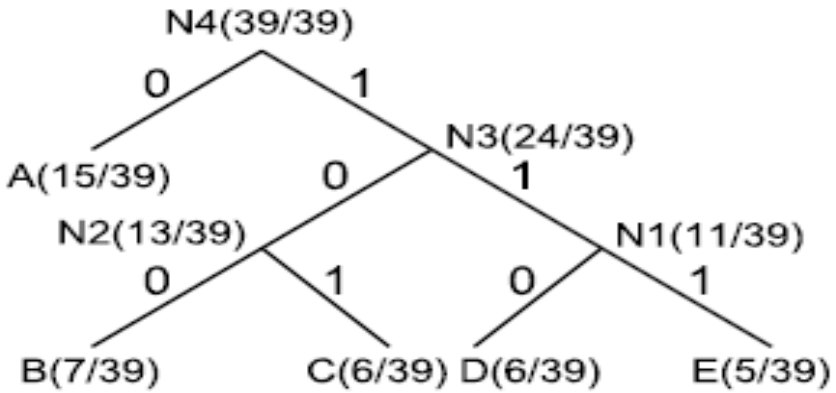


FIGURE 2.2 – Application de l'algorithme de Huffman

De ce fait voici un tableau représentant les mots de code pour chaque symbole et leurs probabilités d'apparitions [7] :

Symbole	Mot de code
A	0
B	100
C	101
D	110
E	111

TABLE 2.3 – Code de Huffman

Voici une comparaison entre le nombre de bits nécessaire pour coder le texte avec le code de Shannon-Fano et celui du code de Huffman d'après les résultats obtenus dans l'exemple précédant[7] :

Nombre de bits nécessaire pour coder le texte avec le codage Shannon-Fano :
P(A)*2+P(B)*2+P(C)*2+P(D)*3+P(E)*3=2.28 bits.
Avec le code Huffman :
P(A)*1+P(B)*3+P(C)*3+P(D)*3+P(E)*3=2.23 bits.
Où P(X) représente la probabilité du symbole X.

D'après les résultats obtenus on constate donc que le codage Huffman est plus optimal que celui de Shannon-Fano.

2.2.2.4 Avantages et inconvénients

- **Avantages :**
  - Le codage de Huffman offre une compression caractère par caractère optimale, c'est-à-dire qu'on ne peut pas obtenir des codes binaires plus courts pour les caractères. Il a aussi l'avantage d'être facile à implémenter par programmation, et le temps d'exécution est plutôt rapide.
- **Inconvénients :**
  - Certaines méthodes de compression qui encode des mots plutôt que des caractères peuvent offrir un taux de compression plus intéressant. Ainsi, le codage de Huffman est souvent utilisé en conjonction avec d'autres techniques de compression comme par exemple LZW.

2.3 Algorithmes de compression

2.3.1 RLE (Run-length-encoding)

2.3.1.1 Définition

L'encodage RLE consiste à remplacer toutes les suites de caractères ou bits pareils par un nombre représentant le nombre de répétitions du caractère ou bits suivit de celui-ci. Généralement, pour moins de gaspillage d'espace, on ne remplace une chaîne que si le nombre de répétition du caractère ou du bit est supérieur à deux [10].

2.3.1.2 Algorithme

Algorithm 1 Compression RLE

```
1: compteur ← 1
2: compteur_Occurrences ← 0
3: caractere_Ecrit ← 0
4: caractere_Séparation ← 127
5: i ← 1
6: Ouvrir fichier_Entree
7: Ouvrir fichier_Sortie
8: if (fichier_Entree <> null) and (fichier_Sortie <> null) then
9:     readln(tampon)
10:    caractere_Ecrit ← tampon
11:    while non fin fichierEntree do
12:        readln(tampon)
13:        if tampon <> caractere_Ecrit then
14:            if compteur > 4 then
15:                writeln(caractere_Séparation)
16:                writeln(compteur_Occurrences)
17:                writeln(caractere_Séparation)
18:                writeln(caractere_Ecrit)
19:            else
20:                for i = 1, compteur, i + + do
21:                    writeln(caractere_Ecrit)
22:                end for
23:            end if
24:            compteur ← 1
25:            caractere_Ecrit ← tampon
26:        else
27:            compteur ← 1 + compteur
28:        end if
29:    end while
30:    if compteur > 4 then
31:        writeln(caractere_Séparation)
32:        writeln(compteur_Occurrences)
33:        writeln(caractere_Séparation)
34:        writeln(caractere_Ecrit)
35:    else
36:        for  i = 1, compteur, i + + do
37:            writeln(caractere_Ecrit)
38:        end for
39:    end if
40:    Fermer fichier_Entree
41:    Fermer fichier_Sortie
42: else
43:    writeln(Erreur lors de l'ouverture des fichiers.)
44: end if
```

Algorithm 2 Décompression RLE

```
1: compteur ← 1
2: compteur_Occurrences ← 0
3: caractere_Ecrit ← 0
4: caractere_Séparation ← 127
5: i ← 1
6: est_Nombre ← FAUX
7: longueur_Tampon ← 0
8: Ouvrir fichier_Entree
9: Ouvrir fichier_Sortie
10: if fichier_Entree <> null and fichier_Sortie <> null then
11:     while Non Fin fichierEntree do
12:         readln(tampon)
13:         if estNombre = VRAI then
14:             if tampon = caractere_Séparation then
15:                 est_Nombre ← FAUX
16:                 compteur ← compteur_Ocurrences
17:                 compteur_Ocurrences ← 0
18:             else
19:                 longueur_Tampon ← longueur_Tampon + tampon
20:             end if
21:         else
22:             if tampon = caractere_Séparation then
23:                 est_Nombre ← VRAI
24:             else
25:                 for i = 1, compteur, i + + do
26:                     writeln(tampon)
27:                 end for
28:                 compteur ← 1
29:             end if
30:         end if
31:     end while
32:     Fermer fichier_Entree
33:     Fermer fichier_Sortie
34: else
35:     writeln(Erreur lors de l'ouverture des fichiers.)
36: end if
```

2.3.1.3 Exemple

Soit la séquence de caractères suivante : AAAAAAARRRRROOO qui est codée sur 15 octets. En appliquant la fonction de codage RLE on obtient la séquence suivante : 7A5R3O qui code sur 6 octets, de ce fait on obtient un gain de 9 octets.

2.3.1.4 Avantages et inconvénients

- **Avantages :**
  - Le principal avantage de l'encodage RLE est son immense simplicité. Comme illustré plus bas, son algorithme tient sur quelques lignes à peine autant au niveau de la compression que de la décompression. Il peut de plus être utilisé sur n'importe quel type de fichier ou après d'autres algorithmes de compression plus évolués pour gagner quelques octets de plus.
- **Inconvénients :**
  - L'encodage RLE possède un désavantage de taille. Le fichier ou texte à compresser doit contenir plusieurs chaînes de caractères ou bits répétés pour être d'une quelconque utilité. C'est pourquoi on le voit très souvent précédé d'une transformation comme BWT ou MTF qui " reclasse " le fichier afin d'augmenter les chances d'avoir plus de caractères ou bits répétés.
- Exemple

si on applique l'algorithme sur la phrase suivant : COMPRESSION on aura comme résultat le mot : 1C1O1M1P1R1E2S1I1O1N.qui est codé sur 20 octes.qui veut dire une perte de 9 octets.

2.3.2 BWT (Burrows-Wheeler transform)

2.3.2.1 Définition

Michael Burrows et David Wheeler ont produit un rapport de recherche en 1994 traitant du travail qu'ils ont effectué au " Digital Systems Research Center " à Palo Alto en Californie. Leur article, "A Block-sorting Lossless Data Compression Algorithm" présentait un algorithme de compression sur une transformation qui n'avait pas encore été rendue publique, découverte par Wheeler en 1983[10].

Le BWT est un algorithme qui prend un bloc de données et qui le réarrange en utilisant un tri. Le résultat contient exactement les mêmes caractères qu'à l'entrée. La seule différence est l'ordre dans lequel ils sont classés. Ce qui est intéressant dans cet algorithme par rapport aux autres algorithmes de tri, c'est qu'il est réversible. C'est-à-dire que l'on peut revenir au fichier original à partir du fichier trié.

2.3.2.2 Algorithme

Algorithm 3 Compression BWT

1: Fonction BWT (string S)
2: Créer une liste de toutes les rotations possibles de S ;
3: Chaque rotation est une ligne dans le tableau ;
4: Trier les lignes de la table en ordre alphabétique ;
5: Retourner la colonne la plus à droite de la table de même que le # de la ligne qui est le mot complet.

Algorithm 4 Décomression BWT

1: Fonction inverseBWT (string S)
2: Créer un tableau vide avec les dimensions carrées de la longueur de la chaîne ;
3: Répéter (longueur de la chaîne)fois
4: Insérer S comme une nouvelle colonne à gauche de la colonne précédente ;
5: Trier les lignes de la table en ordre alphabétique ;
6: Retourner la ligne au numéro indiqué dans le tri BWT.

2.3.2.3 Exemple

Soit le mot BANANA, avant d'appliquer il faut ajouter un signe de fin. Ici dans la figure suivante la fin est marquée par un point rouge[10].

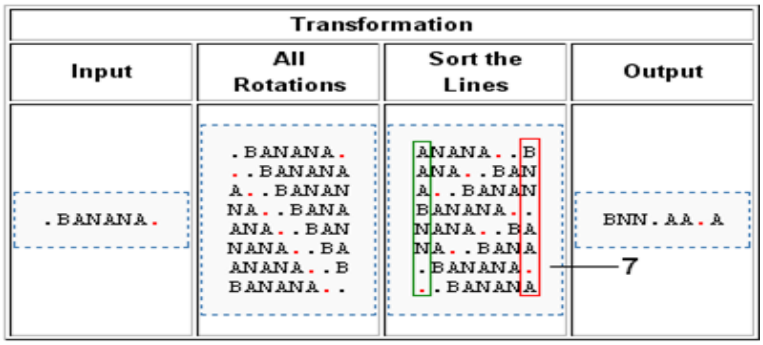


FIGURE 2.3 – Codage BWT

voici le déroulement de la phase de decodage [10] :

inverse transformation
input
BNN.AA.A

Add1	Sort1	Add2	Sort2
B	A	BA	AN
N	A	NA	AN
N	A	NA	A.
.	B	.B	BA
A	N	AN	NA
A	N	AN	NA
.	.	..	.B
A	.	A.	..
Add3	**Sort3**	**Add4**	**Sort4**
BAN	ANA	BANA	ANAN
NAN	ANA	NANA	ANA.
NA.	A..	NA..	A..B
.BA	BAN	.BAN	BANA
ANA	NAN	ANAN	NANA
ANA	NA.	ANA.	NA..
..B	.BA	..BA	.BAN
A..	..B	A..B	..BA
Add5	**Sort5**	**Add6**	**Sort6**
BANAN	ANANA	BANANA	ANANA.
NANA.	ANA..	NANA..	ANA..B
NA..B	A..BA	NA..BA	A..BAN
.BANA	BANAN	.BANAN	BANANA
ANANA	NANA.	ANANA.	NANA..
ANA..	NA..B	ANA..B	NA..BA
..BAN	.BANA	..BANA	.BANAN
A..BA	..BAN	A..BAN	..BANA

Add7	Sort7	Add8	Sort8
BANANA.	ANANA..	BANANA..	ANANA..B
NANA..B	ANA..BA	NANA..BA	ANA..BAN
NA..BAN	A..BANA	NA..BANA	A..BANAN
BANANA.	BANANA.	.BANANA.	BANANA..
ANANA..	NANA..B	ANANA..B	NANA..BA
ANA..BA	NA..BAN	ANA..BAN	NA..BANA
..BANAN	.BANANA	..BANANA	.BANANA.
A..BANA	..BANAN	A..BANAN	..BANANA

Output
.BANANA.

TABLE 2.4 – decodage BWT

2.3.2.4 Avantages et inconvénients

– Le plus grand avantage de cette méthode est qu'elle regroupe les caractères identiques, ce qui rend la compression avec d'autres algorithmes encore plus efficace car la compression profite de la redondance des données.

2.3.3 MTF (Move to Front)

2.3.3.1 Définition

L'algorithme MTF (Move to Front) consiste à remplacer chaque caractère par un indice, donné par un tableau évoluant dynamiquement[10].

Cette technique est utilisable en conjonction avec la transformée de Burrows-Wheeler (BWT)[10].

2.3.3.2 Algorithme avec un exemple

Le tableau est tout d'abord initialisé en rangeant les caractères utilisés pour le codage comme ceci :

Indice	0	1	2	3	4	5	6	...	25
Caractère	A	B	C	D	E	F	G	...	Z

TABLE 2.5 – tableau de position de lettres d'alphabet

Lorsqu'un caractère est lu, son indice est émis, puis ce caractère est placé en première position et tous les autres caractères décalés (d'où le nom de Move to Front). Par exemple si le premier caractère à coder est un E, le tableau deviendrait :

Indice	0	1	2	3	4	5	6	...	25
Caractère	E	A	B	C	D	F	G	...	Z

TABLE 2.6 – Déroulement de l'algorithme MTF

Ainsi, lorsque des caractères semblables se suivent (comme dans BWT), le flux émis contiendra beaucoup de 0, ce qui dans une compression statistique (comme Huffman) augmentera considérablement l'efficacité de compression. On note que dans ce cas, l'émission d'un 0 laisse le tableau identique, et que dans les autres cas, le réarrangement ne concerne que les premiers éléments du tableau.

Par exemple, la séquence "EEEEEA" serait transformée en la suite "400001" ; le tableau évoluerait comme suit :

Indice	0	1	2	3	4	5	6	...	25
État initial	A	B	C	D	E	F	G	...	Z
Tableau modifié par le premier "E"	E	A	B	C	D	F	G	...	Z
Tableau conservé 4 par les 4 "E" suivants								...	
Tableau modifié par le "A"	A	E	B	C	D	F	G	...	Z

TABLE 2.7 – Exemple de déroulement de MTF

Le décodage est tout aussi 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.

2.3.3.3 Avantages et inconvénients

– Cette méthode permet de compresser davantage et est peu coûteuse en temps. Ce qui fait qu'elle peut être efficacement jointe à d'autres méthodes.

2.3.4 Lempel -Ziv 77

2.3.4.1 Définition

LZ 77 est un algorithme de compression sans perte introduit en 1977 par Jacob Ziv et Abraham Lempel. La technique LZ77 c'est de faire une compression d'une séquence en exploitant les répétitions de sous-séquences qui peuvent exister dans cette séquence, pour cela l'algorithme fait un parcourt de la séquence de gauche à droite. La partie de gauche, déjà parcourue, est censée être déjà compressée. La partie droite est celle qui reste à compresser. Le pointeur courant pointe au premier symbole de la partie non compressée. On va chercher dans la partie de gauche la plus longue sous-chaîne qui correspond au début de la partie non compressée. Lorsqu'une concordance est trouvée, on encode sa position (relative au pointeur courant), sa longueur (la concordance) et le premier symbole qui n'a pas concordé dans la partie de droite. Cela nous donne des tuples de la forme ¡P, L, Q¿, ou P représente la position, L représente la longueur de la concordance et Q le caractère prochain [1].

Pour la décompression, on fait une lecture tuple par tuple, et on ajoute à la séquence décompressé le segment désigné.

2.3.4.2 Algorithme[7]

Algorithm 5 Compression LZ77

while not end of input **do**
  $< L, D > \leftarrow find\ best\ match$
  if $L < Lmin$ **then**
    emit C1([next input byte])
  else
    emit C1(¡L,.¿)
    emit C1(¡D¿)
  end if
end while
Where
Procedure emit w :
$a \leftarrow a * w$
Return

Algorithm 6 Décompression LZ77

```
while end not signaled do
    msg ← receive C1
    if msg = [c] then
        store c
    else
        let ¡L,.¿ = msg
        D ← receive C2
        store copy of ¡L,D¿
    end if
end while
where
procedure receive C :
let msg, b s.t. C(msg) * b = a
a ← b
return msg
```

La figure suivante illustre la manière dont une chaine de caractères est compressée, à la fin de l'exécution du programme on aura comme resultat une liste de tuples[11].

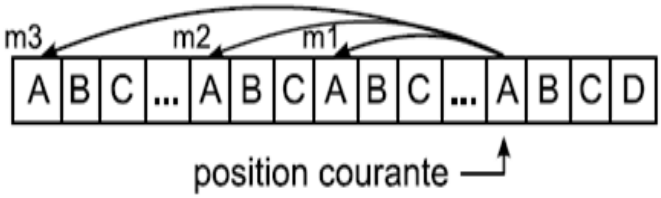


FIGURE 2.4 – compression d'une séquence avec LZ77

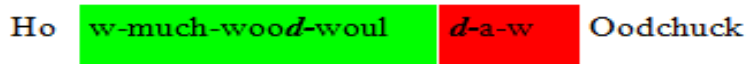
2.3.4.3 Exemple

Soit le texte à coder " how-much-wood-would-a-woodchuck " en entrée.

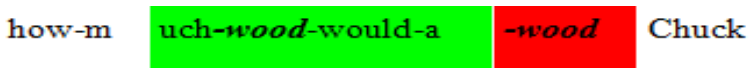
On choisit une fenêtre de 21 caractères. Cette fenêtre sera donc coupée en deux parties :

La première représente la partie lue qui sera composé de 16 caractères et la seconde étant alors de 5 caractères représente le buffer de codage.

Pour des raisons de rapidité, nous avancerons déjà la fenêtre sur le texte. Nous prendrons donc au départ :

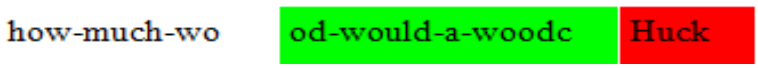


Nous remarquons que la chaîne 'd-' se retrouve déjà dans la fenêtre de pré-lecture. Elle sera donc codée puis remplacée par son code dans le fichier de sortie. Ce code sera donc le suivant : (6, 2, a). 6 représente la position du début de l'équivalence en partant de la fin de la fenêtre de pré-lecture (partie verte). 2 est la longueur de la chaîne équivalente, et 'a' est le caractère suivant dans le buffer qui n'a pas pu être compressé. On déplace ensuite la fenêtre de 3 caractères. Ce qui donne alors :



On remarque que toute la chaîne contenue dans le buffer a une équivalence dans la fenêtre de pré-lecture. On a alors le code suivant qui remplacera la chaîne dans le fichier de sortie : (13, 5, c). Nous voyons ainsi que la longueur maximale de l'équivalence sera donc la taille du buffer. Cela implique aussi que lorsque l'équivalence sera maximale, on devra chercher le caractère suivant qui ne se trouve pas dans ce buffer.

On déplace encore la fenêtre de la taille de l'équivalence + 1 et on tombe alors sur le cas suivant :



On tombe donc sur le cas où il n'y a pas d'équivalence entre le buffer et la fenêtre de pré-lecture. On remarque alors une des plus grosses faiblesses de cet algorithme : comme il n'y a pas d'équivalence, on code le premier caractère du buffer par le triplet (0, 0, h). On décale ensuite la fenêtre d'un caractère. La faiblesse est donc que lorsqu'on ne trouve pas d'équivalence, un caractère prend plus de place qu'un octet !

On continue ensuite l'algorithme jusqu'à ce que l'on n'ait plus aucun caractère dans le buffer.

2.3.4.4 Avantages et inconvénients

- **Avantages :**
  - LZ77 est notamment la base d'algorithmes répandus comme Deflate (ZIP, gzip) ou LZMA (7-Zip).
- **Inconvénients :**
  - La recherche de la concordance est une tâche ardue, et nécessite beaucoup de temps.

2.3.5 LZW (Lempel-Ziv-Welch)

2.3.5.1 Définition

L'algorithme de classe LZ77 utilise un horizon pour trouver les concordances qui aideront à la compression. Mais cette méthode a été ameliorée avec l'arrivée de l'algorithme LZ78 qui utilise une structure de données auxiliaire pour stocker la séquence répétée, cette structure appelée un dictionnaire. Cette fois pour représenter une séquence répétée on utilise l'index de cette séquence dans le dictionnaire mais pas la positions. On aura alors des tuples sous forme <index, & > ou & représente le caractère qui suit la séquence répétée.

LZW est une autre amélioration de l'algorithme LZ78, celui-ci a éliminé l'utilisation des tuples, il utilise seulement les index des séquences dans le dictionnaire [1].

2.3.5.2 Algorithme

Algorithm 7 Compression LZW

```
1: w ← Nul
2: while (lecture d'un caractère c) do
3:     if (w + c) ∈ dictionnaire then
4:         w ← w + c
5:     else
6:         Ajouter(w + c) au dictionnaire
7:         Écrire le code de w
8:         W ← c
9:     end if
10: end while
    Écrire le code de w
```

Algorithm 8 Décomression LZW

```
1: readln (c){lecture d'un caractère c}
2: writeln(c){ajouter à la suite d'oubli}
3: W ← c
4: while (lecture d'un caractère c) do
5:     if c > 255 and c ∈ dictionnaire then
6:         Entrée ← l'entrée du dictionnaire de c
7:     else
8:         if c > 255 and c ∉ dictionnaire then
9:             entrée ← w + w[0]
10:        else
11:            Entrée ← c
12:        end if
13:    end if
14:    Sortie : écrire entrée
15:    Ajouter rw+entrée[0] au dictionnaire
16:    w ← entrée
17: end while
```

2.3.5.3 Exemple

Voici un déroulement de l'algorithme LZW sur une séquence de symbole, on prend comme séquence S=cagtaagagaa [12].

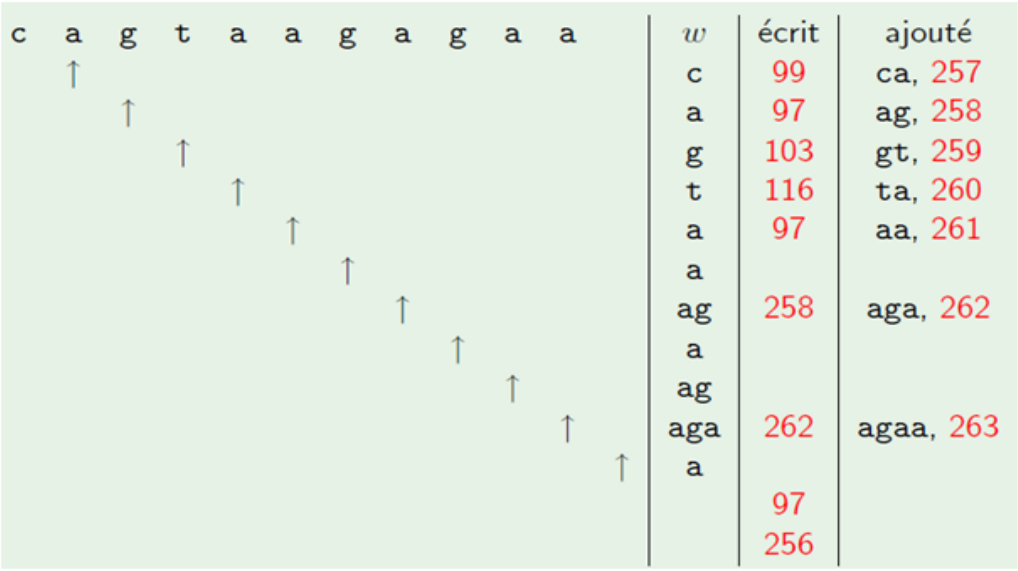


FIGURE 2.5 – compression avec LZW

La sortie de cet algorithme sera un fichier composé de symboles et d'indices. Le dictionnaire n'est pas enregistré, car il est reconstruit à la phase décompression.

2.3.5.4 Avantages et inconvénients

- **Avantages :**
  - Méthode excellente lorsque les données comportent des répétitions variées comme texte en français ou en anglais.
  - Le dictionnaire est reconstruit chez le récepteur son l'envoyer.
  - Taux de compression d'au moins 50%.
- **Inconvénients :**
  - Introduit des éléments supplémentaires dans le dictionnaire qui peuvent être peu utilisés.
  - Le dictionnaire peut être dans l'état plein.
  - La recherche dans le dictionnaire consomme du temps.

Chapitre 3

Analyse et amélioration de l'algorithme de compression RlE

3.1 Encodage RLE

Comme déjà vu, dans le chapitre précédent, le principe de l'algorithme RLE consiste à remplacer toutes les suites de caractères ou bits pareils par un nombre représentant le nombre de répétitions du caractère ou bits suivit de celui-ci.
Généralement, pour moins de gaspillage d'espace, on ne remplace une chaîne que si le nombre de répétitions du caractère ou du bit est supérieur à deux.

- Exemple : soit la suite S=ABBBBBCCCKLM.

L'application de codage RLE pour S nos donne :

```
RLE(S) = A#5B #3CKLM.
Taille originale =12 cotes.
Taille du compressé =10 octets.
Le gain = 2 cotes.
Tel que # est un séparateur qui indique au décodeur que les deux
caractères suivants sont des paramètres du compressé suivant.
```

3.2 Domaines d'application

Les taux de compression obtenus par RLE ne sont pas toujours acceptables. Dans plusieurs cas, la chaine d'entrée est la même que la chaine de sortie.

• Exemple : RLE(ABDFHG)= ABDFHG.

On remarque que la fonction n'a aucun gain dans l'espace mémoire mais au contraire une perte dans le temps. Pour avoir un taux élevé la chaine doit contenir plusieurs chaines de caractères ou bits répétés pour être d'une quelconque utilité. Ce désavantage a réduit l'utilisation d'un tel algorithme dans des cas limités. Cependant jusqu'à présent aucun algorithme ne peut remplacer RLE pour compresser une longue suite de caractère répété consécutivement. RLE est utilisé dans la compression d'image BMP,JPEG...

3.3 RLE+

RLE compare le caractère actuel avec son précédent, s'ils sont égaux alors il incrémente le nombre d'occurrences. L'amélioration qu'on a apportée a ce codage consiste à calculer la différence entre le caractère actuel avec son précédent :
différence = code ASCII (caractère actuel) - code ASCII (caractère précédent)
Si cette différence est trouvée au moins 3 fois consécutivement dans une sous-séquence qui appartient à la séquence S alors cette sous-séquence est remplacé par le code suivant :
Concaténer (premier caractère de la sous séquence, séparateur, taille de la sous-séquence, différence).

3.3.1 Exemple

soit la chaine suivante : S= LYZABCDEF.

L'algorithme commence à lire caractère par caractère en calculant la différance entre un caractère avec son précédent comme suite : Y-L. Z-Y.A-Z.B-A.C-B.D-C.E-D.F-E 13 1 25 1 1 1 1 1.

On remarque que la différence =1 est répété 5 fois consécutivement, à partir de la lettre A d'où on a le compressé suivant :

```
RLE+(S) = LYZA#51.
RLE(S) = S.
```

```
Taille originale=9.
Taille compresse avec RLE = 9.
Taille compresse avec RLE+ = 7.
```

On remarque que RLE+ a un gain dans un cas où RLE n'a aucun gain.

Dans le premier prototype, on a choisi les cas où la différence est supérieure à 0, car dans le cas où la différence est nulle on le considère comme étant un cas de RLE alors on lui applique l'encodage simple de RLE, ou-bien on utilise un autre séparateur (separateur2) qui est différent du séparateur utilisé dans RLE. De ce fait, on peut avoir dans une même chaine codée le cas où une sous-séquence est remplacée par un code RLE suivi d'une autre sous-séquence remplacée par un code RLE+.

Le décodeur détecte cette différence grâce aux deux séparateurs utilisés qui sont différents entre eux.

```
RLE utilise le séparateur "#"
RLE+ utilise le séparateur "&"
```

3.3.2 Exemple de codage multiple

S=ZAGGGGHIJKLM

RLEAM(S)=ZA#4GH&51.

La sous-chaine GGGG est remplacée par #4G, et la sous-chaine HIJKLM est remplacée par H&51.

H représente le premier caractère de la sous-séquence, & le séparateur, 5 la taille de la sous-séquence, et 1 c'est la différence à appliquer pour former les 5 caractères suivants.

3.3.3 Algorithme

L'algorithme est composé de deux parties, une parie compression et une autre décompression.

Algorithm 9 Algorithme de compression RLE+

```
compteur ← 1 Résultat ← "" i ← 0
while i < (source * taille − 1) do
    Courant ← source * charAt(i)
    Suivant ← source * charAt(i + 1)
    Differance ← codeASCII(suivant) − codeASCII(courant)
    j ← i + 1
    Occurrence ← 1
    Deferent ← faux
    while j < source.length − 1 and déférent = false do
        courant ← source * charAt(j)
        suivant ← source * charAt(j + 1)
        Defference2 ← codeASCII(suivant) − codeASCII(courant)
        if defference2 = Differance —— Differance ≥ 0 then
            Occurrence ← (Occurrence + 1)
            j ← j + 1
        else
            Deferent ← vrai
        end if
    end while
    if Occurrence > 1 then
        if Differance = 0 then
            if Occurrence > 2 then
                Résultat ← Résultat + "#" + (Occurrence + 1) + courant
            else
                for k = 1, k = Occurrence, k + + do
                    Resultats ← Résultat + courant
                end for
            end if
        end if
    else
        if Differance > 0 then
            if Occurrence > 2 then
                Resultat ← source.charAt(i) + "&" + Occurrence + différence
            else
                for k = i, Occurrence + i, k + + do
                    Resultats ← résultat + source.charAt(k)
                end for
            end if
        end if
    end if
    i ← j + 1
end while
if i < source.taille then
    Résultat ← résultat + source.charAt(i).
end if
Retourner résultats.
```

Algorithm 10 Algorithme de décompression RLE+

```
1: for i = 1 ; source.taille ; i++ do
2:     Courant ← source.charAt(i)
3:     if courant = 127 then
4:         i ← i + 1
5:         nombre ← source ∗ charAt(i)
6:         i ← i + 1
7:         caractère ←source*charAt(i)
8:         for i = 1 ; nombre ;i++ do
9:             Resultat ← résultat + caractère ;
10:        end for
11:    else
12:        if courant = 128 then
13:            i ← i + 1
14:            nombre ← source ∗ charAt(i)
15:            i ← i + 1
16:            différence ← source ∗ charAt(i)
17:            Dernier ← resultat ∗ charAt(resultat ∗ taille − 1)
18:        end if
19:    else
20:        Resultat ← Result + courant
21:    end if
22: end forRetourner résultat
```

3.4 Role de RLE dans JPEG

Afin de voir l'importance de la modification apportée à RLE, on a implémenté un algorithme de compression très populaire, et qui est utilisé dans la compression d'image tel que RLE est une phase presque finale dans cet algorithme, d'où le choix de l'algorithme JPEG (voir la section suivante). Le langage choisis est le langage JAVA, vue que son apprentissage est facile, et il offre pas mal de fonctions de traitement d'image grâce à ces API.

Afin de voir la différence, nous étions forcés à étudier un algorithme de compression d'image qui est illustré dans la section suivante.

3.5 JPEG (Join Photographic Expert Group)

JPEG est l'abréviation de (Joint-Photographic-Expert-Group), qui représente le comité d'experte qui édite des normes de compression pour l'image fixe, et qui a réussi entre l'année 1991 et 1992 de développer un algorithme de compression d'image qui propose deux modes de compression, le premier avec perte et le second sans perte. Cet algorithme reste jusqu'à nos jours une norme standard pour le codage d'image surtout celles prisent avec une appareille photo[13].

Le principe de cette norme repose sur le principe que l'œil humain est moins sensible aux changements des couleurs brusques qu'aux changements de couleurs lentes.

C'est pour cette raison que l'algorithme fait d'abord une transformation de couleur de RGB vers YCbCr (voir la section suivante).

3.5.1 Algorithme JPEG

Dans ce qui suit nous présentons les étapes de déroulement de l'algorithme JPEG :

3.5.1.1 Transformation de l'image de format RGB vers YCbCr

Dans cette phase l'algorithme applique une fonction de transformation pour extraire les couleurs lentes de l'image.

La formule de passage de RGB ou RVB (rouge vert bleu) vers YCbCr est la suivante [13] :

$$\begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.144 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{pmatrix} \times \begin{pmatrix} R \\ V \\ B \end{pmatrix} + \begin{pmatrix} 0 \\ 128 \\ 128 \end{pmatrix}$$

La formule inverse :

$$\begin{pmatrix} R \\ V \\ B \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1.402 \\ 1 & -0.344 & -0.714 \\ 1 & 1.772 & 0 \end{pmatrix} \begin{pmatrix} Y \\ (Cb - 128) \\ (Cr - 128) \end{pmatrix}$$

FIGURE 3.1 – formule de passage de RGB à YCbCr

3.5.1.2 Découpage de l'image en bloc

Pour faciliter les calculs, une division pour chaque bloc (Y, Cb, Cr) doit être appliquée pour chaque sous-bloc Y, Cr, Cb, chaque bloc contient 8*8 case. Les opérations qui suivent sont appliquées pour chaque bloc indépendamment des autres[13].

3.5.1.3 Application de DCT (transformation de cosinus discret)

Pour chaque bloc on applique une transformation de cosinus discret, chaque valeur du résultat obtenue reflète la rapidité, l'écart et l'importance d'un changement d'un pixel a un autre.
Pour un bloc de taille N*N le DCT s'exprime mathématiquement [13] : La transfor-

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

FIGURE 3.2 – Equation de DCT.

mation inverse s'exprime par [13] :

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

FIGURE 3.3 – DCT inverse.

avec[13]

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

3.5.1.4 La quantification

La quantification s'applique à chaque bloc obtenu à partir du DCT. La quantification consiste à diviser le bloc de 8*8 case par une matrice de quantification

déjà choisis par le codeur, le but est d'atténuer les hautes fréquences auquel l'œil est très peu sensible. C'est à cette phase qu'on perd l'information d'origine, car la quantification est une fonction à sens unique (on ne peut pas revenir en arrière)[13].

Généralement la matrice choisis par le codeur est de cette forme [13] :

$$\begin{pmatrix} A & B & B & C & C & D & D & D \\ B & B & C & C & D & D & D & D \\ B & C & C & D & D & D & D & D \\ C & C & D & D & D & D & D & D \\ C & D & D & D & D & D & D & D \\ D & D & D & D & D & D & D & D \\ D & D & D & D & D & D & D & D \\ D & D & D & D & D & D & D & D \end{pmatrix}$$

FIGURE 3.4 – modele de matrice de quantification

A, B, D sont en ordre croissant, avec ce genre de matrice on veut préserver les pixels qui se situent dans le coté haut à gauche.

3.5.1.5 Codage

La dernière phase consiste à extraire de chaque bloc les valeurs obtenues après la quantification. Pour cela nous devons faire un parcours de la matrice qui s'effectue en zigzags comme dans la figure suivante [13] :

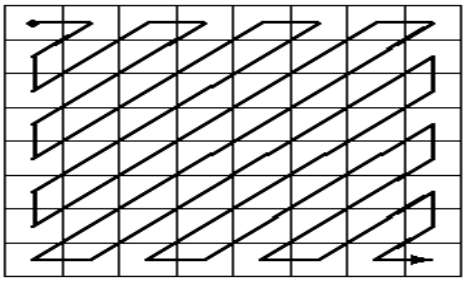


FIGURE 3.5 – le parcoure d'une matrice lors de la phase de codage

Donc on obtient une chaine de 64 valeurs. Ensuite cette chaine sera codée avec RLE, et huffman consécutivement. Le schéma général de cet algorithme est illustré dans la figure suivante [13] :

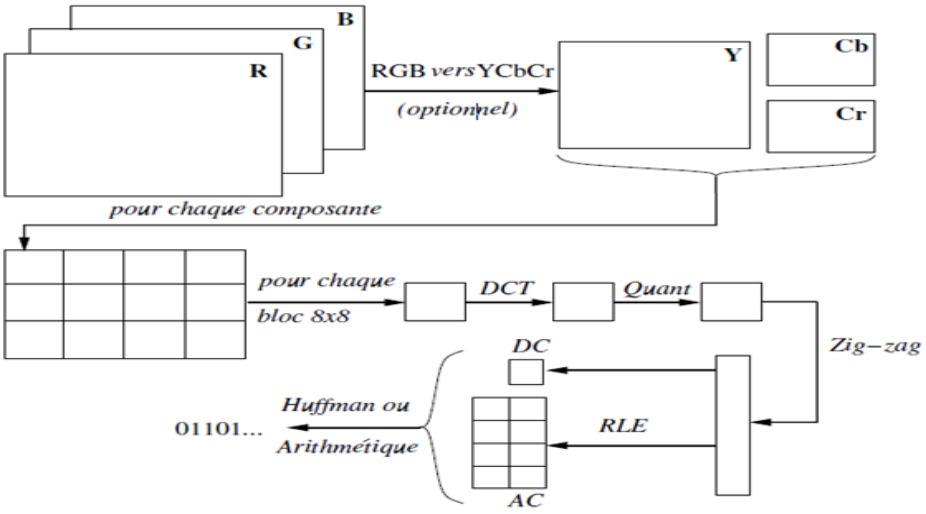


FIGURE 3.6 – le deroulement de l'algorithme JPEG

3.5.1.6 La décompression

Pour obtenir l'image à partir du compressé, on doit faire le chemin inverse de l'algorithme. La quantification inverse consiste à multiplier un bloc par la matrice de quantification.

Chapitre 4

Implémentation de RLE+

4.1 Présentation de l'application

Afin d'évaluer notre algorithme et de voir l'importance de l'amélioration qu'on a apporté à RLE, nous avons créé une application qui a les fonctionnalités suivantes :

- Charger un fichier image.
- Appliquer l'algorithme JPEG sur cette image.
- Enregistrer l'image résultante en utilisant un codage RLE en ignorant le codage de Huffman.
- Enregistrer l'image résultante en appliquant le codage RLE+.
- Calcul pour chaque type de codage les paramètres suivants :

  Taille du fichier initial.

  Taille du fichier compressé.

  Gain= Taille fichier initial-Taille de fichier compressé.

  Taux=taille finale/taille initiale.
- Exporter les résultats vers un fichier EXCEL pour dessiner un graphe.
- Afficher le fichier EXCEL pour déduire à partir des résultats le meilleur algorithme.

4.2 Langage utilisé

JAVA est le langage le plus utilisé durant notre cycle de formation, il propose pas mal de fonctions pour la gestion de l'interface graphique et gestion de déférents types et structures de données[14]. Pour le traitement et compilation du code java notre choix c'est porté sur l'IDE éclipse.

Pour faire la comparaison entre les résultats obtenus par les deux algorithmes, on a choisis Excel pour construire les tableaux et les graphes de comparaison.

4.2.1 Structure d'un fichier Excel

Un fichier Excel est composé de plusieurs feuilles.

Chaque feuille est composée de plusieurs lignes.

Chaque ligne est compose de plusieurs cellules.

Chaque cellule a un type (NUMIRIQUE, STRING, FORMIULE, BOOLEAN, EREUR...).

4.2.2 L'API POI-HSSF

HSSF permet la manipulation de document EXCEL de la version 97 a la version 2007 uniquement pour le format OLE2(fichier avec l'extension .XLS) le format OOXML d'Excel n'est pas encore supporté(fichier avec l'extension .XLSX).
Notre application utilise cette API pour faire l'interaction avec le fichier EXCEL[14].
Parmi les nombreuses fonctionnalités proposées par cette API, il y a :

- Lecture et écriture de document.
- Création et modification des différentes entités qui composent un document (document, feuille, cellule...).
- Formatage des cellules (alignement, police, couleur...).
- Support graphique : dessin de primitives, d'image...).

4.3 Interface de l'application

La figure suivante représente une capture d'écran à notre application :

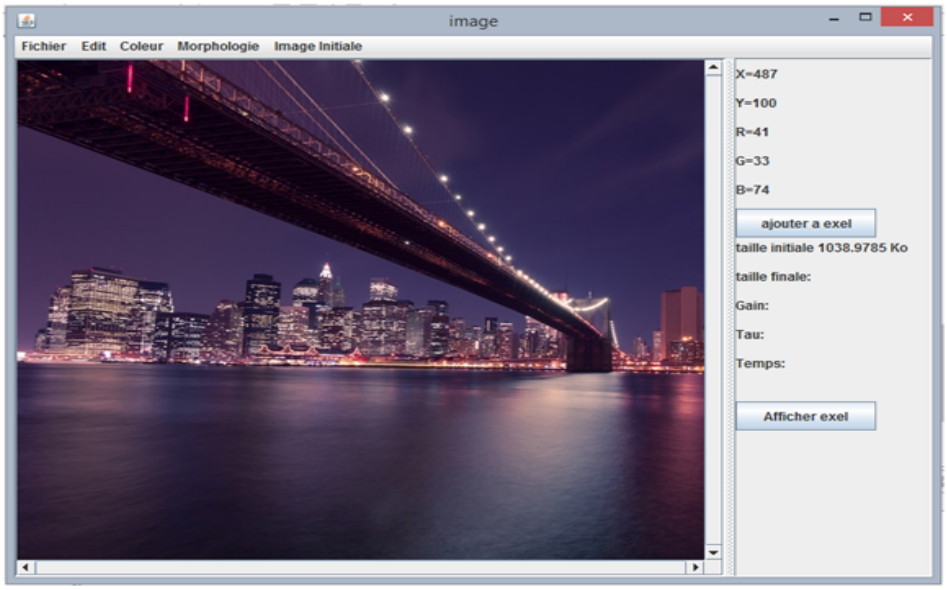


FIGURE 4.1 – capture d'écran à l'application

L'interface offre un ensemble de composants pour l'édition d'une image, elle permet aussi d'appliquer certaines fonctions qui sont développées durant la recherche. Après la compression d'une image on peut voir sur l'écran le résultat qu'on obtiendra après la décompression. Les statistiques des résultats sont affichées dans le panneau à droite.

Le bouton sur le panneau des statistiques avec le titre " ajouter a Excel " permet d'exporter les statistiques actuelles vers un fichier Excel, ce dernier peut être affiché en cliquant sur le bouton titré " afficher EXCEL ".

Image obtenue en appliquant l'algorithme JPEG :

FIGURE 4.2 – resultat de la compression

On remarque que l'image résultante est presque identique à l'image initiale sauf l'effet de mosaïque qui a été produit, d'ailleurs c'est à cause de cet inconvénient que la société JPEG a continué ces recherches pour régler ce problème ce qui a produit la naissance de JPEG 2000. Ces détails sont hors de notre recherche car le but d'implémenter JPEG est d'évaluer l'algorithme RLE.

4.4 Statistiques

Afin de comparer l'algorithme de RLE avec celui de RLE+, nous avons appliqué l'algorithme JPEG jusqu'à la phase codage RLE, et on lui a appliqué l'amélioration qu'on a apporté. On a ignoré la phase du codage de huffman, car le but est d'évaluer notre amélioration.

Les résultats suivants sont obtenus grâce à l'exécution consécutive de programme sur une série d'images choisis par nous-mêmes. L'unitée utiliser est le kilo octets.

• Les paramètres de simulation sont les suivants :

- Temps de compression.
- Taux de compression = taille de compressé / taille du fichier original
- Gain en kilo octet= taille de fichier original - taille de compressé

Un ensemble de tableaux et de graphes ont été générés grâce à l'api d'EXCEL, sur l'exécution des deux algorithmes sur 5 différentes images (taille et contenus).

Pour les graphes, on a utilisé des histogrammes groupés afin de voir la différence entre les deux algorithmes pour chaque image.

4.4.1 Résultat 1 :

Le premier tableau obtenu a pour but de voir le gain en kilo octets pour chaque image avec les deux algorithmes.

taille originale	48,05273438	577,5996	1038,979	1516,693	1654,6
GAIN RLE	2,052734375	175,5996	708,9785	994,6934	1269,6
GAIN RLE+	2,052734375	188,5996	727,9785	1033,693	1297,6

TABLE 4.1 – tableau des résultats des deux algorithmes

Le graphe obtenu de ces résultats est le suivant :

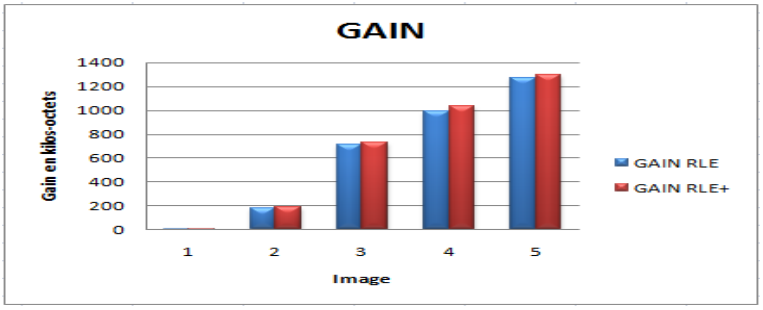


FIGURE 4.3 – Graphe de gain obtenu.

On remarque à partir des résultats obtenus des deux algorithmes que le gain obtenu avec RLE+ est supérieur ou égale à celui obtenu avec RLE.

4.4.2 Résultat 2 :

Le tableau qui suit représente les taux de compression obtenus des deux algorithmes :

taille originale	48,05273438	577,5996	1038,979	1516,693	1654,6
taux RLE	0,957281632	0,695984	0,31762	0,34417	0,232685
taux RLE+	0.957281632	0,673477	0,299332	0,318456	0,215762

TABLE 4.2 – Tableau de taux de compression

Son graphe est le suivant :

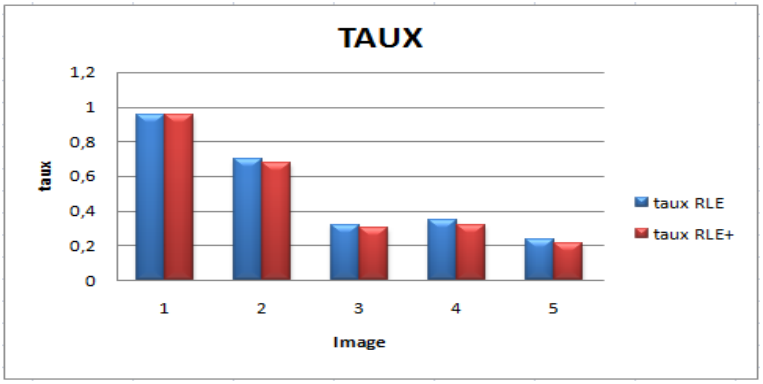


FIGURE 4.4 – graphe de taux de compression

Le taux de compression aide à déduire lequel des algorithmes est le meilleur. Pour déduire lequel des deux algorithmes est meilleur, on regarde lequel des deux taux se raproche le plus de 0, donc selon les resultats obtenus on remarque que RLE+ est meilleur que RLE à ce niveau.

4.4.3 Résultat 3 :

Pour le tableau suivant, il représente le temps qu'un algorithme prend pour effectuer une compression d'une image :

taille originale	48,05273438	577,5996	1038,979	1516,693	1654,6
temps RLE	11	92	72	129	109
temps RLE+	25	107	134	207	178

TABLE 4.3 – Tableau représentatif du temps de compression

Le graphe résultant du tableau est le suivant :

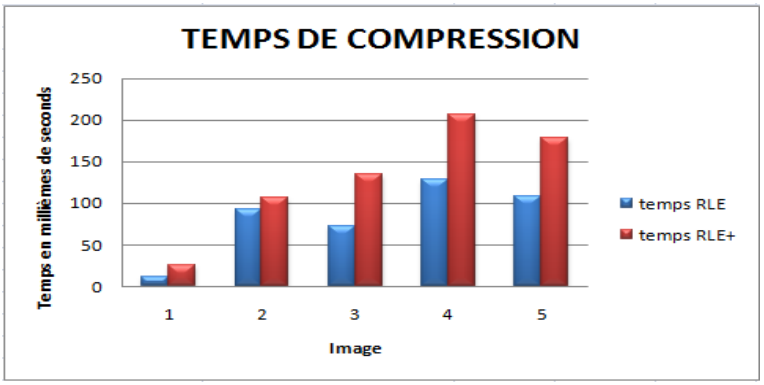


FIGURE 4.5 – Graphe des temps de compression.

D'après les resultats obtenus, on remarque que le temps d'execution de RLE+ est superieur que celui de RLE, cela est justifié par l'opération de calcul de la différence entre les symboles faite par RLE+.

4.4.4 Résultat 4 :

Afin de bien voir l'importance de l'amélioration effectué à RLE, nous avons calculé la différence entre les gains obtenus par les deux algorithmes (gain RLE+ -

gain RLE).

Taille de fichier	48,05273438	577,5996	1038,979	1516,693	1654,6
Gain RLE-RLE+	0	13	19	39	28

TABLE 4.4 – tableau de différence de gain

On remarque du tableau que les résultats obtenus sont toujours positif ou nul, cela signifie que RLE+ est toujours meilleur que RLE dans le gain.

Ces résultats sont représentés dans le graphe suivant :

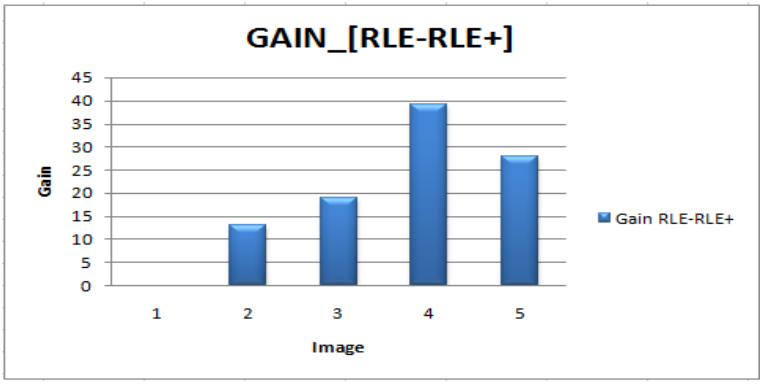


FIGURE 4.6 – la différence de gain entre RLE et RLE+

On remarque que la différence de gain augmente quand la taille de l'image est grande.

Par exemple, on a dans la 4eme image, un gain d'amélioration qui est égal ou presque à 40 kilo octet, cet espace mémoire récupéré peut stocker une image en couleur de dimension 116*116.

D'aprés les resultats obtenus, on remarque que l'amélioration qu'on apportée a l'algorithme RLE est efficace, du fait que les statistiques obtenues avec cette amélioration sont bien meilleurs que celles obtenues avec RLE, même si il y a une perte au niveau du temps d'exécution de la compression du fait qu'on a privilégié l'espace memoire sur le temps.

Conclusion générale

Dans ce travail, nous avons mis en œuvre une amélioration à une méthode de compression sans perte afin d'élever le taux de compression obtenue par cette dernière, et qui est utilisée dans la compression d'image.

Dans un premier temps nous avons procédé à une étude comparative entre les différents algorithmes de compression sans perte existants, l'approche utilisée dans notre méthode est la même que RLE.

Dans le chapitre 3, on a étudié l'algorithme JPEG, en illustrant la place de RLE dans ce dernier, une implémentation a été faite sous JAVA. Avec le travail réalisé on remarque l'importance d'une telle amélioration sur une compression JPEG.

A travers les testes et les résultats que nous avons présentés dans le chapitre 4, nous avons démontré que RLE+ est meilleur dans le taux de compression et gain d'octets dans la plupart des cas par rapport a l'algorithme originale RLE.

Mais l'inconvénient de cette amélioration est les temps de compression obtenu et qui montre que RLE+ consomme un temps supérieur au temps obtenu avec RLE. Ce point peut être jugé comme négligeable dans le cas où la compression a été utilisé pour l'archivage des données, et avec une amélioration dans le code source de l'algorithme on peut obtenir un temps inferieur.

Les problèmes rencontrés lors de la validation de l'algorithme RLE+, ne doit pas nous pousser à ignorer l'amélioration affecté à RLE, mais au contraire il faut chercher les cas ou RLE+ obtiens des taux de compression meilleurs que les autres algorithmes existants. Comme par sauvegarder une base de données en commençons par le champ Identifient et qui est utilisé généralement dans tous les types de base de données et qui incrémente à chaque fois. Cette incrémentation sera compressée grâce à RLE+.

• Perspective

La compression des données est appelée à prendre un rôle encore plus important en raison du développement des réseaux et du multimédia. Son importance est surtout due au décalage qui existe entre les possibilités matérielles des dispositifs

que nous utilisons (débits sur Internet, sur Numéris et sur les divers câbles, capacité des mémoires de masse,...) et les besoins qu'expriment les utilisateurs (visiophonie, vidéo plein écran, transfert de quantités d'information toujours plus importantes dans des délais toujours plus brefs). Quand ce décalage n'existe pas, ce qui est rare, la compression permet de toute façon des économies.

La quantification appliquée dans l'algorithme JPEG sert à éliminer la partie bas à droite pour produire une suite de 0 qui est compressée grâce à RLE, cette opération conduit à la perte d'information car la fonction inverse de quantification ne produit pas la même matrice (multiplication par 0 donne un 0). Avec l'amélioration apportée on peut imaginer une possibilité d'utiliser une autre forme de table de quantification, tel que l'existence d'une séquence de 0 n'est pas obligatoire pour atteindre un bon taux de compression, car RLE+ compresse aussi les suites de la forme 12345..n ou bien 0246. . .2n.

Donc la suite compressée n'est pas obligatoirement nulle. Avec un tel résultat on peut estimer la reconstruction de la table initiale, en utilisant une table de quantification différente de la table utilisée avant. Le but de cette dernière est de produire une suite qui a une forme compressible avec RLE+.

Ou bien d'éviter carrément la transformation DCT suivi d'une quantification et de remplacer ces deux dernières par une autre transformation qui produit des séquences qui aide RLE+ à compresser.

Bibliographie

[1] Bookstein (Abraham) et Fouty (Gary). - A Mathematical Model For Estimating The Effectiveness of Bigram Coding. Information Processing and Management, vol. 12, 1976, pp. 111-116.

[2] Pike (J.). - Text Compression Using a 4-bit Coding System. Computer Journal, vol. 24, no° 4, 1981, pp. 324-330.

[3] Tropper (Richard). - Binary-Coded Text : A text Compression Method. Byte, vol. 7, no° 4, 1982, pp. 398-413.

[4] PIGEON, Steven. Contribution à la compression de données. Thèse de Doctorat : Informatique. Montreal, 2001.

[5] ZITOUNI, Athmane. Ondelettes et techniques de compression d'images numérique, These de Doctorat : Sciences en Electronique. Biskra, 2013.

[6] Jean-Guillaume Dumas, Jean-Louis Roch, Éric Tannier, Sébastien Varrette :theorie des codes. Compression. Dunod, Paris, ISBN 9-78-210-050692-7, 2007.

[7] BEAUDOIN, Vincent. Développement de nouvelles techniques de compression de données sans perte. Mémoire de Maitres en Sciences (M.Sc.), Science et Génie. Québec, 2008.

[8] C. E. Shannon. A mathematical theory of communication. Bell System Technical Journal, 27, July, October 1948.

[9] D. A. Huffman. A method for the construction of minimum-redundancy codes. In Proceedings of the Institute of Radio Engineers, volume 40, pages 1098-1101, sep 1952.

[10] http ://www2.cegep-ste-foy.qc.ca/departements/freesite-informatique/ProjetsRechercheH2006/Gr4628/H06-620-Equipe7/rapport.html, visité le 20/03/2016

[11] http ://d.nouchi.free.fr/TER/c6.html, visité le 20/03/2016.

[12] http ://www-igm.univ-mlv.fr/ lecroq/cours, visité le 25/03/2016.

[13] http ://www-igm.univ-mlv.fr/ dr/XPOSE2013/La compression de donnees/j-peg.html, visité le 20/05/2016.

[14] Jean-Michel DOUDOUX. Développons en Java. Http ://www.jmdoudoux.fr, visité le 01/06/2016.