République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique

Université des Sciences et de la Technologie
HOUARI BOUMEDIENE

B. P. 32, El-Alia, 16111 Bab-Ezzouar, ALGER
Téléphone/Fax: +213 21 24 76 07

جامعة هواري بومدين
للعلوم والتكنولوجيا
U S T H B

الجمهورية الجزائرية الديمقراطية الشعبية

وزارة التعليم العالي
والبحث العلمي

جامعة هواري بومدين
للعلوم والتكنولوجيا

ص.ب 32، العالية، 16111، باب الزوار، الجزائر
الهاتف / الفاكس : 07 76 24 21 213+

Cours: SYSTEMES MULTIMEDIA

Master RSD, 2014/2015

Prof. Slimane Larabi

Chapitre 1. Compression de données sans perte

1.1 Intérêts de la compression

1.2 Critères des algorithmes de compression

1.3 Types de compression

1.4 Algorithmes de compression sans perte

1.1 Intérêts de la compression

1.2 Critères des algorithmes de compression

Les techniques de compression peuvent être comparées selon les critères suivants :

- **Efficacité (taux de compression) :** C'est le rapport entre la taille du fichier compressé et sa taille initiale. Pour une meilleure utilisation des algorithmes, il faut comprendre que la plupart d'entre eux sont plus ou moins efficace dans un type d'image que dans un autre.

- **Qualité de compression :** Compression avec une perte d'un certain pourcentage de la qualité ou compression sans perte.

- **Vitesse de compression/décompression**

- **Accessibilité :** Sous licence, ou libres de droits.

1.3 Types de compression

- Sans perte

- Avec perte

1.4 Algorithmes de compression sans perte

Ces méthodes sont basées généralement sur la recherche et le codage des données redondantes. Ce type de compression peut s'appliquer à n'importe quel type de données.

Algorithme RLE (*Run Length Encoding*) :

- C'est la méthode la plus simple et la plus utilisée. Son principe de base consiste à rechercher des données redondantes (pixels dans le cas des images) et en codant la longueur et la valeur.
- Ainsi à chaque répétition d'un pixel plus d'un nombre n précisé par l'utilisateur, cette suite de pixels est remplacée par un caractère spécial indiquant la compression suivi par le nombre de répétitions du pixel et en fin sa valeur.

Algorithmes de compression sans perte :

Caractéristiques :

- Algorithme de compression ou de décompression très simple à implémenter.
- Taux de compression relativement faible par rapport à d'autres algorithmes.
- Obtient des meilleurs résultats avec des images contenant des zones importantes de couleur contiguë (images monochromes).
- La compression des images en couleurs complexes (photos) ne donne pas des bons résultats.

La compression LZW (*Lempel Ziv Welch*) :

- Abraham Lempel et Jakob Ziv sont les créateurs du compresseur LZ77, inventé en 1977 (d'où son nom).
- Ce compresseur était alors utilisé pour l'archivage (les formats ZIP).
- Cet algorithme a été amélioré par *Terry Welch* de la société *Unisys* en 1984.
- Il est basé sur un dictionnaire (bibliothèque) construit au fur et à mesure de la lecture du fichier à coder.
- Les chaînes de caractères sont placées une par une dans la bibliothèque. Lorsqu'une chaîne est déjà présente dans la bibliothèque, son code de fréquence d'utilisation est incrémenté.
- Les chaînes de caractères ayant des codes de fréquences élevés sont remplacées par un " mot " ayant un nombre de caractères le plus petit possible et le code de correspondance est inscrit dans la bibliothèque.
- On obtient ainsi l'information encodée et sa bibliothèque.

La compression LZW (*Lempel Ziv Welch*) :

Algorithm
Begin
w = ∅
Repeat read a character **k**
if wk ∈ Dictionary
  then w = **wk**
else output (code (**w**))
        Dictionary ← **wk**
        w = **k**
Until EOF
End

La décompression LZW (*Lempel Ziv Welch*) :

Algorithm
Begin
read k;
output Dict[k];
w = Dict[k];
while (read k)
/* k peut être un caractère ou un code. */
{
output Dict[k];
add w Dict[k][0] to dictionary;
w = Dict[k];
}
End

La compression LZW (*Lempel Ziv Welch*) :

Caractéristiques :

- Cet algorithme est breveté par la société Unisys, il a été utilisé dans les formats TIFF et GIF, par contre l'algorithme LZ77 est libre de droit et a été utilisé dans le format PNG.

- Il s'applique très bien sur les images de faibles profondeurs (nombre réduit de couleurs différentes) puisque les motifs différents doivent être relativement faibles pour être répétés.

- Il est l'un des plus répandus algorithmes, et est très rapide aussi bien en compression qu'en décompression.

La compression LZW (*Lempel Ziv Welch*) :

- Input : **^wed^we^wee^wed^** → 17 octets

Input: ^wed^we^wee^wed^

Sub string	Code

w	k	wk	∈ Dict?	Insert in Dict	Output

Input: ^wed^we^wee^wed^

Sub string	Code

w	k	wk	∈ Dict?	Insert in Dict	Output
ø	^	^	yes		

Input: ^wed^we^wee^wed^

Sub string	Code
^w	90

w	k	wk	∈ Dict?	Insert in Dict	Output
ø	^	^	yes		
^	w	^w	no	^w	^

Input: ^wed^we^wee^wed^

Sub string	Code
^w	90
we	91

w	k	wk	∈ Dict?	Insert in Dict	Output
ø	^	^	yes		
^	w	^w	no	^w	^
w	e	we	no	we	w

Input: **^wed**^we^wee^wed^

Sub string	Code
^w	90
we	91
ed	92

w	k	wk	∈ Dict?	Insert in Dict	Output
ø	^	^	yes		
^	w	^w	no	^w	^
w	e	we	no	we	w
e	d	ed	no	ed	e

Input: **^wed^**we^wee^wed^

Sub string	Code
^w	90
we	91
ed	92
d^	93

w	k	wk	∈ Dict?	Insert in Dict	Output
ø	^	^	yes		
^	w	^w	no	^w	^
w	e	we	no	we	w
e	d	ed	no	ed	e
d	^	d^	no	d^	d

Input: **^wed^w**e^wee^wed^

Sub string	Code
^w	90
we	91
ed	92
d^	93

w	k	wk	∈ Dict?	Insert in Dict	Output
ø	^	^	yes		
^	w	^w	no	^w	^
w	e	we	no	we	w
e	d	ed	no	ed	e
d	^	d^	no	d^	d
^	w	^w	yes		

Input: **^wed^we**^wee^wed^

Sub string	Code
^w	90
we	91
ed	92
d^	93
^we	95

w	k	wk	∈ Dict?	Insert in Dict	Output
ø	^	^	yes		
^	w	^w	no	^w	^
w	e	we	no	we	w
e	d	ed	no	ed	e
d	^	d^	no	d^	d
^	w	^w	yes		
^w	e	^we	no	^we	^w

Input: **^wed^we^**wee^wed^

Sub string	Code
^w	90
we	91
ed	92
d^	93
^we	95
e^	94

w	k	wk	∈ Dict?	Insert in Dict	Output
ø	^	^	yes		
^	w	^w	no	^w	^
w	e	we	no	we	w
e	d	ed	no	ed	e
d	^	d^	no	d^	d
^	w	^w	yes		
^w	e	^we	no	^we	^w
e	^	e^	no	e^	e

Input: **^wed^we^w**ee^wed^

Sub string	Code
^w	90
we	91
ed	92
d^	93
^we	94
e^	95

w	k	wk	∈ Dict?	Insert in Dict	Output
ø	^	^	yes		
^	w	^w	no	^w	^
w	e	we	no	we	w
e	d	ed	no	ed	e
d	^	d^	no	d^	d
^	w	^w	yes		
^w	e	^we	no	^we	^w
e	^	e^	no	e^	e
^	w	^w	yes		

Input: **^wed^we^wee**^wed^

Sub string	Code
^w	90
we	91
ed	92
d^	93
^we	94
e^	95

w	k	wk	∈ Dict?	Insert in Dict	Output
ø	^	^	yes		
^	w	^w	no	^w	^
w	e	we	no	we	w
e	d	ed	no	ed	e
d	^	d^	no	d^	d
^	w	^w	yes		
^w	e	^we	no	^we	^w
e	^	e^	no	e^	e
^	w	^w	yes		
^w	e	^we	yes		

Input: **^wed^we^wee**^wed^

Sub string	Code
^w	90
we	91
ed	92
d^	93
^we	94
e^	95
^wee	96

w	k	wk	∈ Dict?	Insert in Dict	Output
ø	^	^	yes		
^	w	^w	no	^w	^
w	e	we	no	we	w
e	d	ed	no	ed	e
d	^	d^	no	d^	d
^	w	^w	yes		
^w	e	^we	no	^we	^w
e	^	e^	no	e^	e
^	w	^w	yes		
^w	e	^we	yes		
^we	e	^wee	no	^wee	^we

Input: **^wed^we^wee^**wed^

Sub string	Code
^w	90
we	91
ed	92
d^	93
^we	94
e^	95
^wee	96

w	k	wk	∈ Dict?	Insert in Dict	Output
ø	^	^	yes		
^	w	^w	no	^w	^
w	e	we	no	we	w
e	d	ed	no	ed	e
d	^	d^	no	d^	d
^	w	^w	yes		
^w	e	^we	no	^we	^w
e	^	e^	no	e^	e
^	w	^w	yes		
^w	e	^we	yes		
^we	e	^wee	no	^wee	^we
e	^	e^	yes		

Input: **^wed^we^wee^w**ed^

Sub string	Code
^w	90
we	91
ed	92
d^	93
^we	94
e^	95
^wee	96
e^w	97

w	k	wk	∈ Dict?	Insert in Dict	Output
ø	^	^	yes		
^	w	^w	no	^w	^
w	e	we	no	we	w
e	d	ed	no	ed	e
d	^	d^	no	d^	d
^	w	^w	yes		
^w	e	^we	no	^we	^w
e	^	e^	no	e^	e
^	w	^w	yes		
^w	e	^we	yes		
^we	e	^wee	no	^wee	^we
e	^	e^	yes		
e^	w	e^w	no	e^w	e^

Input: **^wed^we^wee^we**d^

Sub string	Code
^w	90
we	91
ed	92
d^	93
^we	94
e^	95
^wee	96
e^w	97

w	k	wk	∈ Dict?	Insert in Dict	Output
ø	^	^	yes		
^	w	^w	no	^w	^
w	e	we	no	we	w
e	d	ed	no	ed	e
d	^	d^	no	d^	d
^	w	^w	yes		
^w	e	^we	no	^we	^w
e	^	e^	no	e^	e
^	w	^w	yes		
^w	e	^we	yes		
^we	e	^wee	no	^wee	^we
e	^	e^	yes		
e^	w	e^w	no	e^w	e^
w	e	we	yes		

Input: **^wed^we^wee^wed^**

Sub string	Code
^w	90
we	91
ed	92
d^	93
^we	94
e^	95
^wee	96
e^w	97
wed	98

w	k	wk	∈ Dict?	Insert in Dict	Output
ø	^	^	yes		
^	w	^w	no	^w	^
w	e	we	no	we	w
e	d	ed	no	ed	e
d	^	d^	no	d^	d
^	w	^w	yes		
^w	e	^we	no	^we	^w
e	^	e^	no	e^	e
^	w	^w	yes		
^w	e	^we	yes		
^we	e	^wee	no	^wee	^we
e	^	e^	yes		
e^	w	e^w	no	e^w	e^
w	e	we	yes		
we	d	wed	no	wed	we

Input: **^wed^we^wee^wed^**

Sub string	Code
^w	90
we	91
ed	92
d^	93
^we	94
e^	95
^wee	96
e^w	97
wed	98

w	k	wk	∈ Dict?	Insert in Dict	Output
ø	^	^	yes		
^	w	^w	no	^w	^
w	e	we	no	we	w
e	d	ed	no	ed	e
d	^	d^	no	d^	d
^	w	^w	yes		
^w	e	^we	no	^we	^w
e	^	e^	no	e^	e
^	w	^w	yes		
^w	e	^we	yes		
^we	e	^wee	no	^wee	^we
e	^	e^	yes		
e^	w	e^w	no	e^w	e^
w	e	we	yes		
we	d	wed	no	wed	we
d	^	d^	yes		

Input: **^wed^we^wee^wed^**

Sub string	Code
^w	90
we	91
ed	92
d^	93
^we	94
e^	95
^wee	96
e^w	97
wed	98

w	k	wk	∈ Dict?	Insert in Dict	Output
ø	^	^	yes		
^	w	^w	no	^w	^
w	e	we	no	we	w
e	d	ed	no	ed	e
d	^	d^	no	d^	d
^	w	^w	yes		
^w	e	^we	no	^we	^w
e	^	e^	no	e^	e
^	w	^w	yes		
^w	e	^we	yes		
^we	e	^wee	no	^wee	^we
e	^	e^	yes		
e^	w	e^w	no	e^w	e^
w	e	we	yes		
we	d	wed	no	wed	we
d	^	d^	yes		
d^	ø	d^			d^

La compression LZW (*Lempel Ziv Welch*) :

- Input : **^wed^we^wee^wed^** → 17 octets
- Output: ^ w e d ^w e ^we e^ we d^ → 10 octets

La compression LZW (*Lempel Ziv Welch*) :

Décodage

```
string entry;
char ch; int code;
int prevcode, currcode;

prevcode = read code;
decode/output prevcode;
while (there is still data to read)
{
   curcode = read code;
   output Dict(curcode);
   ch = first char of entry;
   add Dict(prevcode)+ch to Dict;
   prevcode = currcode;
}
```

La compression LZW (*Lempel Ziv Welch*) :

Décodage

Exemple:

Dict={0->a, 1->b)

Chaine de codes: 0 1 2 4 3 6

Algorithm

prevcode = 0;

output ‘a’;

while (there is still data to read)

{

curcode=1; output ‘b’; ch=‘b’; Dict(2)=‘ab’; prevcode=‘b’;

curcode=2; output ‘ab’; ch=‘a’; Dict(3)=‘ba’; prevcode=‘ab’;

curcode=4; output ‘ab..a’; ch=‘a’; Dict(4)=‘aba’; prevcode=‘aba’;

curcode=3; output ‘ba’; ch=‘b’; Dict(5)=‘abab’; prevcode=‘ba’;

curcode=6; output ‘ba..b’; ch=‘b’; Dict(6)=‘bab’; prevcode=‘bab’;

}

La compression LZW (*Lempel Ziv Welch*) :

Décodage

Exemple:

Dict={0->a, 1->b, 2->c)

Chaine de codes: 0 3 1 2

Algorithm

prevcode = 0;

output ‘a’;

curcode=3; output ‘a..a’; ch=‘a’; Dict(3)=‘aa’; prevcode=‘aa’;

curcode=1; output ‘b’; ch=‘b’; Dict(4)=‘aab’; prevcode=‘b’;

curcode=2; output ‘c’; ch=‘b’; Dict(5)=‘bc’; prevcode=‘c’;

}

- **Le codage de Huffman :**

- Cet algorithme est développé en 1952 par David Huffman,

- il est l'un des algorithmes les plus anciens, son codage est basé sur la fréquence d'apparition d'un caractère : plus le caractère apparaît souvent plus son code sera court et vice-versa.

- Pour permettre un décodage unique les codes attribués aux différents caractères doivent être préfixés, c'est-à-dire qu'aucun caractère n'est un préfixe d'un autre.

- C'est pourquoi on appelle aussi ce codage un ***VLC*** préfixé (*Variable Length Code*, code à taille variable).

- **<u>Le codage de Huffman :</u>**

Début

- Chercher la fréquence d'apparition de chaque caractère.

Répéter:

- Trier les caractères par ordre décroissant de fréquence (poids).
- Construire l'arbre binaire comme suit :
    - Relier les deux caractères de fréquences les plus basses et
    - Affecter à ce nœud la somme des fréquences des caractères.
- **Jusqu'à ce que tous les nœuds soient reliés**

L'arbre étant construit, on met un 1 sur la branche à droite du nœud et un 0 sur celle de gauche.

Fin

- **Le codage de Huffman :**

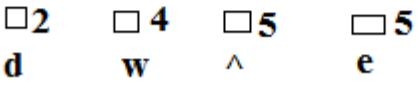
- Parcourir l’arbre de la racine vers chacune des feuilles pour tirer le code de chaque caractère.

- **Le codage de Huffman :**

- Exemple: “**^wed^we^wee^wed^**”:
- e: 5
- ^: 5
- w: 4
- d: 2

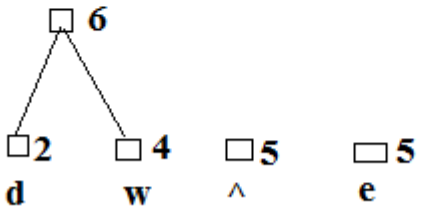
- **<u>Le codage de Huffman :</u>**

- Exemple: “**^wed^we^wee^wed^**”:
- e: 5
- ^: 5
- w: 4
- d: 2



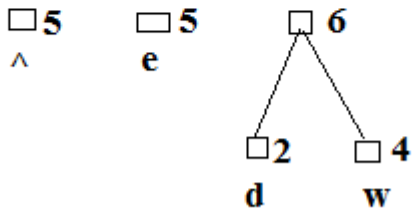
- **<u>Le codage de Huffman :</u>**

- Exemple: “**^wed^we^wee^wed^**”:
- e: 5
- ^: 5
- w: 4
- d: 2



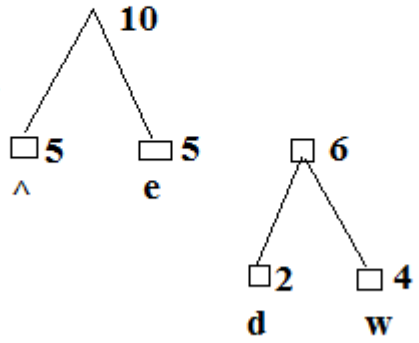
- **<u>Le codage de Huffman :</u>**

- Exemple: “**^wed^we^wee^wed^**”:
- e: 5
- ^: 5
- w: 4
- d: 2



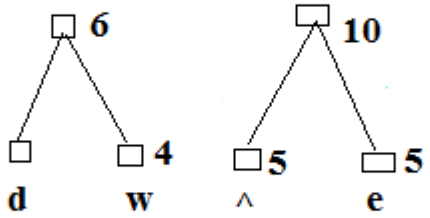
- **<u>Le codage de Huffman :</u>**

- Exemple: “**^wed^we^wee^wed^**”:
- e: 5
- ^: 5
- w: 4
- d: 2



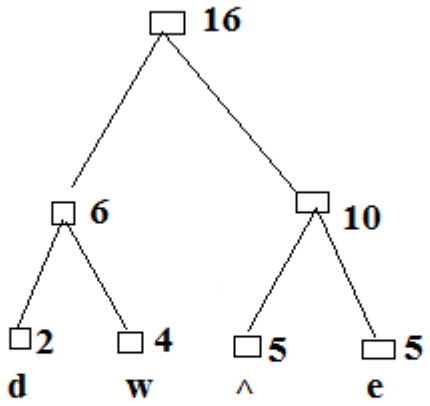
- **<u>Le codage de Huffman :</u>**

- Exemple: “**^wed^we^wee^wed^**”:
- e: 5
- ^: 5
- w: 4
- d: 2



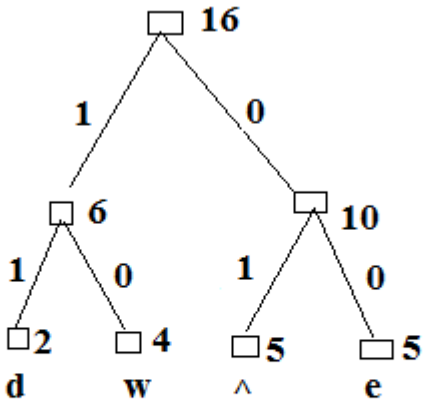
- **<u>Le codage de Huffman :</u>**

- Exemple: “**^wed^we^wee^wed^**”:
- e: 5
- ^: 5
- w: 4
- d: 2



- **<u>Le codage de Huffman :</u>**

- Exemple: “**^wed^we^wee^wed^**”:
- e: 5
- ^: 5
- w: 4
- d: 2



d:11
w: 10
^: 01
e: 00

01 10 00 11 01 10 00 01 10 00 00 01 10 00 11 01
16x2 bits, soit 4 octets au lieu de 16 octets.

- **<u>Le codage de Huffman :</u>**

  - **Caractéristiques :**
    - Cet algorithme permet d'avoir un taux de compression très élevé (50% en moyenne) et un temps de compression assez rapide.
    - La bibliothèque doit être transmise avec le fichier.
    - Il est très sensible: la perte d'un bit entraîne une altération de toutes les données qui suivent lors de la décompression

- **ALGORITHME DEFLATE (GZIP)**

L’algorithme *deflate*, qui est une combinaison des algorithmes **LZ77 et Huffman**. 'Deflate' a été développé en réponse à des problèmes de brevet logiciel couvrant LZW et autres algorithmes de compression, limitant ainsi les utilisations possibles de *compress* et autres programmes d'archivage populaires.

- **ALGORITHME DEFLATE (GZIP)**

- L'algorithme DEFLATE produit un ensemble de blocs correspondant à des blocs de données successives. Chacun des blocs est compressé à l'aide de LZ77 et Huffman.
- Littéraux et les longueurs sont compressés par un arbre de Huffman, les distances sont compressées par un autre arbre de Huffman.
-
- Les arbres sont rangés au début de chaque bloc. Un bloc se termine si **deflate** détermine qu'il utile de démarrer un autre bloc avec un nouvel arbre.

- **ALGORITHME LZ77**

- LZ77 est proposé pour l'analyse des données et de déterminer comment réduire son espace en remplaçant les informations redondantes avec des méta données.
- Deux fenêtres : W_f pour se déplacer dans le texte qui reste à compresser, W_b qui contient ainsi la portion précédemment codée.

- Si une sous chaine de W_f existe dans W_b, alors elle est récrite par une métadonnée : (**pos**, **long**, **car_suiv**), où :

- **ALGORITHME LZ77**

- **pos** : indique où commence l'équivalence dans W_b,
- **long** : sa longueur
- **car_suiv** : est le caractère non compressé suivant.
- Ce caractère est inséré parce que les auteurs ont jugé que s'il n'a pas été codé dans l'expression lue, c'est qu'il y a de grandes chances qu'il ne soit pas compressé ensuite.
- Cet algorithme utilise un dictionnaire constitué de W_b.
- La distance **pos** est limitée à 32KO, et la longueur **long** à 258 Octets.

- **ALGORITHME LZ77**

- **Exemple de compression**
- Soit le texte à coder " how-much-wood-would-a-woodchuck " en entrée.
- Taille de W_b=16, taille de W_f=5.
-

ho	w-much-wood-woul	d-a-w	oodchuck

- ‘d-‘ sera codé : (6, 2, a).
- On déplace ensuite la fenêtre de 3 caractères. Ce qui donne alors :

How-mu	ch-wood-would-a	-wood	chuck

- ‘-wood‘ sera codé : (13, 5, c).

- **ALGORITHME LZ77**

- **Exemple de compression**

how-much-wo	od-would-a-woodc	huck	

- (0,0,h),(0,0,u), (0,0,c), (0,0,k).
- Si on ne trouve pas d'équivalence, un caractère prend plus de place qu'un octet!

- **ALGORITHME LZ77**

- **Les problèmes et inconvénients de la compression LZ77**
- Le problème est de trouver la plus grande équivalence au plus vite. La méthode brutale, qui consiste à tester une par une les équivalences, n'est pas acceptable. Il est alors recommandé d'utiliser un arbre binaire de recherche.
- L'autre inconvénient de LZ77 est qu'il renvoie toujours un triplet (position, longueur, caractère suivant) même si l'équivalence n'est que d'un octet (1 caractère) ou même aucun. Dans ce cas, nous utilisons plus que 8 bits pour coder 1 caractère.
- **Décompression LZ77**
- La décompression sera très rapide puisque la recherche d'équivalence se fait dans la fenêtre W_b.
-