Rapport de TP - SY26

TP02 - Codage de Huffman

Rémi Burtin

Cyril Fougeray

10 avril 2014

Université de Technologie de Compiègne

1 Introduction

L'objectif de ce TP est de comparer le codage de Huffman au codage arithmétique. Pour cela, nous travaillons sur une image ('lena.bmp') en niveaux de gris, ce qui permet de fixer une seule valeur (entre 0 et 255) pour chaque pixel de l'image. Nous avons donc réalisé deux scripts, chacun codant et décodant une image en utilisant un des deux algorithmes.

2 Codeur de Huffman

2.1 Mise en oeuvre

L'image que nous manipulons pour réaliser l'encodage/décodage selon Huffman ne contient qu'une seule composante correspondant à l'intensité du gris. L'image originale étant en couleurs, nous la convertissons en utilisant la fonction *rgb2gray* qui retourne une matrice à deux dimensions, de même taille que l'image.

Afin de réaliser l'encodage, il est nécessaire de connaître la loi de probabilités des niveaux de gris de l'image. Pour cela, nous commençons par réaliser l'histogramme comptant le nombre d'occurrences (l'effectif) de chaque valeur de gris. L'histogramme est en fait un vecteur de taille 256, ce qui correspond aux différentes valeurs de gris. Ainsi, à l'index $x \in [1, 256]$, nous obtenons le nombre d'occurrences de l'intensité de gris égale à $x - 1$ dans l'image (le gris est codé sur un octet, donc sa valeur est comprise entre 0 et 255). Afin d'obtenir une fonction densité de probabilités ($\sum P(X = x_i) = 1$), nous normalisons l'histogramme en divisant chacune de ses valeurs par le nombre total de pixels dans l'image.

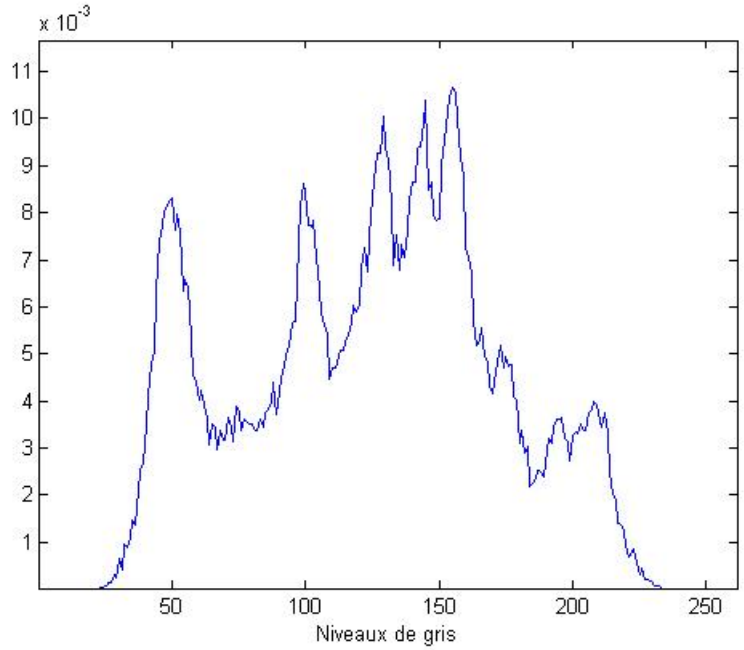

FIGURE 1 – Distribution de probabilités, niveaux de gris

L'algorithme d'Huffman fonctionne de sorte que les symboles ayant la plus grande probabilité utilisent le moins de ressources afin d'utiliser le moins de bande passante possible, et vice-versa, afin d'améliorer la compression. Dans le cas de notre image, beaucoup d'intensités de gris n'apparaissent pas dans l'image (leur probabilité d'apparition est alors nulle) : elles correspondent aux indexes de l'histogramme ayant pour valeur 0 (*cf. figure 1*). Afin de supprimer ces valeurs inutiles et qui utiliseront beaucoup de ressources, nous créons un nouvel histogramme, ayant des valeurs toutes strictement positives. Afin de reconstruire l'image originale après encodage/décodage, nous devons alors établir une table de correspondance. Ainsi, chaque index de l'histogramme sans zéro devra être lié à sa valeur de gris présent dans l'image.

Selon la longueur de l'histogramme sans zéro, nous créons un dictionnaire avec les nouveaux symboles allant de 0 à L, L étant la taille du nouvel histogramme. Dans le tableau 1 page 3, nous ne conservons que les niveaux de gris présents dans l'image, nous voyons que l'index dans le nouvel histogramme ne correspond plus au niveaux de gris. Une table de correspondance est alors créée, *cf. tableau 2*. Afin de construire le dictionnaire via le codeur de Huffman, nous devons créer les probabilités qui en découle (en normalisant l'histogramme). Chaque effectif est alors divisé par le nombre total des effectifs, soit ici le nombre de pixels.

Histogramme simple		Histogramme sans zéros		
Index	Effectif	Index	Effectif	Niveau de gris
1	0	1	2	3
2	0	2	6	5
3	2	3	7	6
4	0			
5	6			
6	7			
7	0			
8	0			

TABLE 1 – Exemple - Histogrammes

L'index de l'histogramme sans zéro correspond à un symbole lors de la création du dictionnaire par la fonction *huffmandict*.

Index (Histogramme sans zéro)	Niveau de gris associé
1	3
2	5
3	6

TABLE 2 – Exemple - Table de correspondance

Une fois le dictionnaire créé, nous devons adapter les niveaux de gris présents dans l'image pour qu'ils correspondent aux symboles passées lors de l'encodage. Pour cela, nous utilisons la table de correspondance. En suivant l'exemple ci-dessus, nous voyons que les pixels de niveau de gris 3 doivent être remplacés par 1.

Une fois le dictionnaire créé, nous pouvons encoder via la fonction *huffmanenco*. La fonction reçoit l'image dans un vecteur ligne ainsi que le dictionnaire comprenant la valeur d'encodage de chaque symbole. Elle nous retourne l'image encodé (soit une image compressée).

Nous décodons ensuite afin de retrouver l'image originale. La fonction *huffmandeco* permet de retrouver le vecteur ligne correspondant à l'image originale. Comme nous l'avons vu, et d'après l'exemple, une valeur décodée égale à 1 correspond à un niveau de gris de 3. Nous devons alors faire la conversion de chacune des valeurs du vecteur ligne vers sa valeur de gris associée. Ensuite, nous reconstruisons l'image à partir du vecteur colonne vers une matrice 2D, grâce à la fonction *reshape*. L'image

peut ensuite être affichée.

Le script du codeur de Huffman est présent en annexe A.1.

2.2 Résultats

Nous affichons un log grâce à la fonction *disp* :

```
>> huffman('../lena.bmp')
Lecture de l'image ../lena.bmp
Image couleur : conversion en gris
Creation vecteur des probalites des niveaux de gris
Creation du vecteurs des differents symboles
(nombre de symboles : 217)
Creation du dictionnaire (via huffmandict())
Longueur moyenne des mots encodes : 7.4675
Encodage de l'image
Decodage de l'image
Duree encodage/decodage : 156.7178s.
Taux de compression :
1 - taille finale / taille initiale = 0.066566
1 - longueur moyenne mot code / 8 = 0.066566
```

Nous voyons qu'il est possible de calculer le taux de compression uniquement via création du dictionnaire de symboles. L'algorithme retourne la longueur moyenne des mots. Cette moyenne étant pondérée par la probabilité certaine d'apparition des mots, nous pouvons utiliser cette valeur pour calculer la taille finale (après encodage) en la multipliant par le nombre de mots qui correspond ici au nombre de pixels dans l'image.

L'autre méthode consiste à comparer la taille de l'image encodée à la taille de l'image source. Dans notre cas, l'image encodée est un vecteur de bits. L'image source est elle un vecteur d'octets, sa taille en bits correspond donc à la taille de l'image multipliée par 8.

Plus le taux de compression est important, plus l'image est compressée.

3 Codeur arithmétique

3.1 Mise en oeuvre

L'algorithme du codeur arithmétique est simple à mettre en oeuvre dans la mesure où le travail effectué précédemment est largement utilisé.

La fonction d'encodage, *arithenco*, prend comme paramètres l'image ainsi que l'histogramme sans zéro. Ici, nous n'avons pas besoin de normaliser l'histogramme, mais la fonction n'accepte pas d'effectifs nuls pour un index de l'histogramme. Nous supprimons donc tous les zéros de l'histogramme et créons une table de correspondance, de la même manière que pour le codeur de Huffman, voir tableaux 1 et 2. L'image est modifiée : les niveaux de gris de l'image sont remplacés par les symboles correspondants. La fonction retourne l'image encodée.

La fonction de décodage, *arithdeco*, prend comme paramètres l'image encodée, l'histogramme des probabilités sans zéro ainsi que la taille de l'image. Elle retourne l'image décodée qui doit bien sûr être convertie via la table de correspondance afin de retrouver tous les niveaux de gris originaux.

3.2 Résultats

```
>> arith('../lena.bmp')
Lecture de l'image ../lena.bmp
Image couleur : conversion en gris
Encodage de l'image
Decodage de l'image
Duree encodage/decodage : 14.552s.
Taux de compression : 1 - taille finale / taille initiale = 0.069527
```

4 Conclusion

	lena.bmp (256*256)	peppers.tif (512*512)
Huffman	156.7178s	483.36s
Arithmétique	14.552s	30.95s

TABLE 3 – Temps d'exécution

	lena.bmp (256*256)	peppers.tif (512*512)
Huffman	6.66 %	4.77 %
Arithmétique	6.95 %	5.09 %

TABLE 4 – Taux de compression

On peut observer que le codeur arithmétique donne de meilleurs résultats que le codeur de Huffman. Les taux de compression sont quasiment égaux, alors que les temps d'encodage + décodage différent d'un facteur 16.

A Codes source MATLAB

A.1 Algorithme - codage de Huffman

```
function compression = huffman(path)
global prob avglen dict histogramme histo_0 corres index;

% matrice de l'image
disp(['Lecture de l''image ' path]);
img = imread(path);

% initialisation vecteurs
histogramme = zeros(1,256); % occurences par niveaux de gris

composantes = size(img, 3); % Taille de la 3eme dimension

if (composantes == 3) % conversion en gris si image couleur
    disp('Image couleur : conversion en gris');
    img = rgb2gray(img);
end

dim = size(img,1)*size(img,2); % nombre de pixels dans l'image
list=reshape(double(img), 1, dim); % images en ligne (vecteur)

% Creation histogramme
for i=1:dim,
    histogramme(round(list(i))+1) = histogramme(round(list(i))+1) + 1;
end

% Suppression des zeros
[val, index] = find (histogramme ~= 0) ;
histo_0 = histogramme(index);

prob = zeros(1,length(histo_0)); % initialisation "prob" (histo normalise)

disp('Creation vecteur des probalites des niveaux de gris');
for i=1:length(histo_0),
    prob(i) = histo_0(i)/dim;
end

disp('Creation du vecteurs des differents symboles')
disp(['(nombre de symboles : ' num2str(length(histo_0)) ')']);
symbols = [1:length(histo_0)];

disp('Creation du dictionnaire (via huffmandict())');
[dict,avglen] = huffmandict(symbols,prob);
disp(['Longueur moyenne des mots encodes : ' num2str(avglen)])

% Creation des tables de correspondance
% pour pouvoir retrouver plus tard les bonnes
% valeurs des differents pixels
corres = zeros(1, 256);

for i=1:length(index),
    corres(index(i)) = i;
```

```
52 end
53
54 list_0 = zeros(1,dim);
55
56 for i=1:dim,
57     list_0(i) = corres(list(i)+1);
58 end
59
60 % encodage de l'image (passer un vecteur ligne)
61 disp('Encodage de l''image');
62 tic % calcul du temps d'encodage/decodage
63 enco = huffmanenco (list_0, dict);
64
65 % decodage (retourne un vecteur)
66 disp('Decodage de l''image');
67 deco = huffmandeco(enco, dict);
68 time = toc;
69 disp(['Duree encodage/decodage : ' num2str(time) 's.']);
70
71 % on remet les bonnes valeurs des pixels
72 deco_2 = zeros(1,dim);
73 for i=1:dim,
74     deco_2(i) = index(deco(i));
75 end
76
77 % reshape vecteur -¿ image 2D
78 imagedeco = reshape(deco_2, size(img,1), size(img,2));
79
80 imshow(imagedeco/256);
81
82 compression = 1-length(enco)/(dim*8);
83 disp('Taux de compression : ');
84 disp(['1 - taille finale / taille initiale = ' num2str(compression)]);
85 compression2 = 1-avglen/8;
86 disp(['1 - longueur moyenne mot code / 8 = ' num2str(compression2)]);
87
88 return;
89 end
```

A.2 Algorithme - codage arithmétique

```
function compression = arith(path)

% matrice de l'image
disp(['Lecture de l''image ' path]);
img = imread(path);

% initialisation vecteurs
histogramme = zeros(1,256); % occurences par niveaux de gris


composantes = size(img, 3); % Taille de la 3eme dimension

if (composantes == 3) % conversion en gris si image couleur
    disp('Image couleur : conversion en gris');
    img = rgb2gray(img);
end

dim = size(img,1)*size(img,2); % nombre de pixels dans l'image
list=reshape(double(img), 1, dim); % images en ligne (vecteur)

% creation histogramme
for i=1:dim,
    histogramme(round(list(i))+1) = histogramme(round(list(i))+1) + 1;
end

[val, index] = find (histogramme ~= 0) ;
histo_0 = histogramme(index);

corres = zeros(1, 256);

for i=1:length(index),
    corres(index(i)) = i;
end

list_0 = zeros(1,dim);

for i=1:dim,
    list_0(i) = corres(list(i)+1);
end

disp('Encodage de l''image');
tic % calcul du temps d'encodage/decodage
code = arithenco(list_0, histo_0);

disp('Decodage de l''image');
deco = arithdeco(code, histo_0, dim);
time = toc;
disp(['Duree encodage/decodage : ' num2str(time) 's.']);

deco_2 = zeros(1,dim);

for i=1:dim,
    deco_2(i) = index(deco(i));
```

```
54  end
55
56  imagedeco = reshape(deco_2, size(img,1), size(img,2));
57
58  imshow(imagedeco/256);
59
60  compression = 1-length(code)/(dim*8);
61  disp(['Taux de compression : ' num2str(compression)]);
62
63
64  return;
65
66  end
```