

GAMA DOS SANTOS Melissa  
GALLINA Thomas  
PAGÈS Clément

# LA COMPRESSION DES IMAGES NUMÉRIQUES

## INTRODUCTION

En informatique, les images font partie des données occupant le plus d'espace de stockage. Une photographie de  $10 \times 15$  cm de qualité moyenne dépasse rapidement les 25 méga-octets. Actuellement, les capacités de nos disques durs ne sont plus une impasse au stockage de données de tels volumes, mais leur échange se révèle plus délicat. En effet, les réseaux tels qu'Internet utilisent des images en grandes quantités. Or, faire transiter 25 méga-octets de données pour chaque image nécessaire est impossible.

Il a donc fallu trouver une solution efficace qui permette de profiter des images numériques sans pour autant subir le préjudice du volume de stockage qu'elles occupent. A cette fin, des algorithmes de compression ont été développés. L'objectif est de diminuer la quantité d'informations sans perdre de qualité visible.

De nombreux algorithmes de compression existent en informatique : les formats ".zip", ".rar" et ".gzip" sont devenus incontournables. Pour l'imagerie numérique, des algorithmes particuliers ont été mis en œuvre, répondant à différents critères tels que la difficulté de mise en application, la quantité de place économisée ou encore la perte de qualité engendrée par la compression.

Nous allons donc étudier les principaux algorithmes de compression utilisés en infographie, pour mieux cerner leurs points communs, leurs différences et leurs particularités et comprendre dans quelles mesures ils permettent de profiter au mieux du monde de l'imagerie numérique. A cette fin, nous présenterons les caractéristiques des images numériques et les différents moyens existants pour les coder ainsi que des algorithmes de compression, avant de nous attarder sur les algorithmes de compression conservateurs puis destructeurs appliqués aux images.

Les illustrations données tout au long de ce dossier sont copiées sur le CD-ROM ci-joint.

# PLAN DU DOSSIER

<b>1</b>	<b>Généralités</b>	<b>3</b>
1.1	Stocker les images . . . . .	3
1.1.1	Vocabulaire . . . . .	3
1.1.2	Les images matricielles . . . . .	3
1.1.3	Les images vectorielles . . . . .	3
1.2	La gestion des couleurs . . . . .	3
1.2.1	Les couleurs indexées . . . . .	3
1.2.2	Les images en couleurs vraies . . . . .	5
1.2.3	Le format BITMAP . . . . .	6
1.3	Généralités concernant la compression . . . . .	7
1.3.1	Evaluer la compression . . . . .	7
1.3.2	La compression RLE . . . . .	8
1.3.3	L'algorithme Huffman . . . . .	8
1.3.4	Les algorithmes à dictionnaire : la compression LZW . . . . .	10
<b>2</b>	<b>La compression conservatrice</b>	<b>11</b>
2.1	Implémentation des algorithmes RLE . . . . .	11
2.1.1	Fonctionnement de l'algorithme sur les images . . . . .	11
2.1.2	Application du RLE sur les images en couleurs vraies . . . . .	11
2.1.3	Application du RLE sur les images en couleurs indexées . . . . .	12
2.2	Implémentation des algorithmes LZW . . . . .	12
2.2.1	Compression LZW d'une image 256 couleurs . . . . .	12
2.2.2	Compression LZW d'une image en couleurs vraies . . . . .	13
2.3	Le codage prédictif . . . . .	13
2.3.1	Principe du codage prédictif . . . . .	13
2.3.2	Réalisation d'une prédiction sur un pixel . . . . .	14
<b>3</b>	<b>La compression non-conservatrice : la norme JPEG</b>	<b>16</b>
3.1	Principe et fonctionnement théorique du JPEG . . . . .	16
3.1.1	La Transformation en Cosinus Discrète . . . . .	16
3.1.2	La méthode JPEG . . . . .	17
3.2	Implémentation des algorithmes JPEG sur les images . . . . .	18
3.2.1	Vérification de l'efficacité de la compression JPEG . . . . .	18
3.2.2	Application du JPEG sur une matrice de pixels . . . . .	20
3.3	Conséquences sur la qualité des images . . . . .	21
3.3.1	Images en couleurs vraies . . . . .	22
3.3.2	Images en couleurs indexées . . . . .	22
	<b>Conclusion</b>	<b>24</b>

# 1 Généralités

## 1.1 Stocker les images

### 1.1.1 Vocabulaire

En infographie, les images sont représentées par une suite de points sur l'écran, qui sont la plus petite partie qui composent une image. On appelle ces points des pixels (contraction de PICTURE ELEMENT). Notons qu'un pixel n'a pas une dimension fixe, mais qu'elle peut varier selon la qualité des images.

Une image numérique est caractérisée, entre autre, par deux données :

- **Sa définition** : Elle indique le nombre de pixels qui composent l'image. On calcule ce nombre en multipliant le nombre de pixels en longueur par le nombre de pixels en largeur.
- **Sa résolution** : Elle indique le nombre de pixels contenus dans chaque unité de longueur. On utilise en général pour unité de longueur le pouce (Inch en anglais). La résolution se calcule en divisant la définition de l'image par le nombre de pouces qu'elle occupe. On obtient alors une résolution en pixel par pouce (Dots per Inch).

On différencie deux grandes façons de définir les images numériques : par des matrices et par des données géométriques.

### 1.1.2 Les images matricielles

Elles sont codées selon une suite de matrices (ensemble de nombres) qui définissent chaque pixel selon sa position dans l'image, sa couleur, etc.

### 1.1.3 Les images vectorielles

Elles sont réalisées selon un ensemble de tracés géométriques (courbes, droites, segments, etc.). Elles sont très utilisées en DAO (Dessin Assisté par Ordinateur) notamment, car elles permettent d'agrandir et de rétrécir l'image sans perte de qualité. Cependant, elles sont peu adaptées pour les images non géométriques.

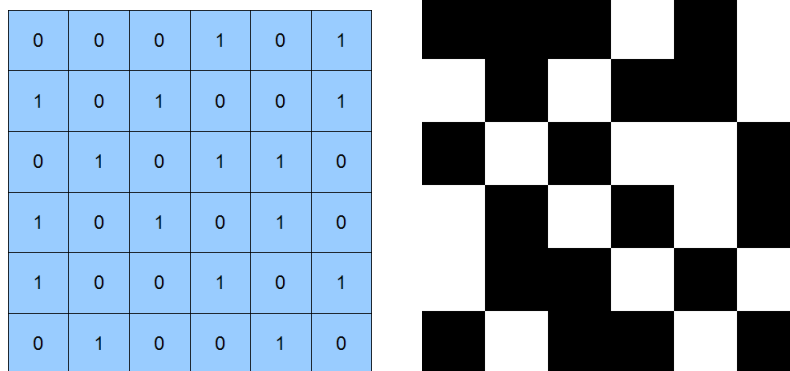
## 1.2 La gestion des couleurs

Les couleurs peuvent être définies par plusieurs méthodes, selon la qualité recherchée et la quantité de mémoire disponible. Ainsi, on peut limiter le nombre de couleurs dans une image (grâce aux couleurs indexées) ou choisir une représentation plus réaliste grâce à la coloration RGB, par exemple.

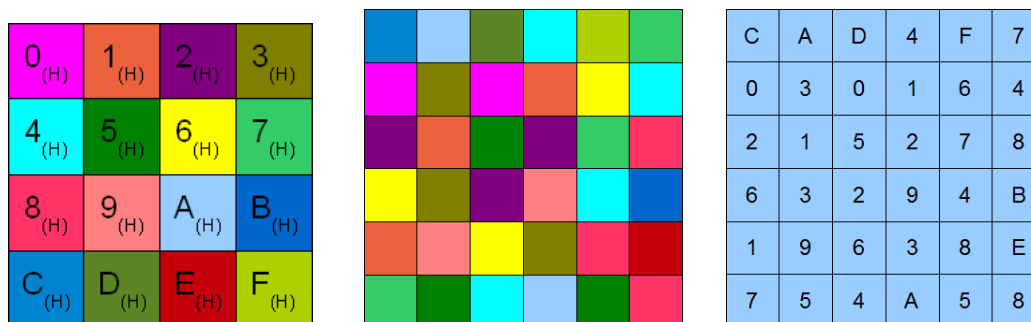
### 1.2.1 Les couleurs indexées

L'indexation consiste à dresser, au début de l'image, un " annuaire " des couleurs qui seront représentées dans l'image finale. Il suffit, ensuite, de coder pour chaque pixel la couleur qui lui correspond dans la LUT (Look-Up Table, ou palette de couleurs). Il existe plusieurs niveaux d'indexations, qui varient essentiellement selon le nombre de couleurs recherchées :

- **Les images en noir et blanc** : à chaque pixel est associé 1 bit, qui peut prendre deux valeurs possibles : 0 si le pixel est noir et 1 s'il est blanc, comme le montre l'exemple ci-dessous :



- **Les images 16 couleurs** : afin de restreindre le nombre de couleurs à 16, il faut coder, au début de l'image, une palette de couleurs (Look-Up Table), qui contient la valeur en binaire des couleurs qu'elle contient. Ensuite, il suffit, pour chaque pixel de l'image, de lui affecter la valeur de l'adresse de la couleur qui lui correspond dans la LUT. Ainsi, chaque pixel occupe 4 bits, dont la valeur est comprise entre  $0_{(H)}$  et  $F_{(H)}$ . De ce fait, deux pixels occupent un octet et le volume nécessaire reste restreint. Cependant, la qualité est elle aussi limitée par le nombre de couleurs disponibles.



Dans l'ordre : Palette de couleurs - Corps de l'image - Code du corps de l'image

- **Les images en nuances de gris** : pour coder les images en nuances de gris, chaque pixel est codé sur un octet, pouvant prendre des valeurs comprises entre  $00_{(H)}$  et  $FF_{(H)}$ . Plus la valeur s'approche de  $FF_{(H)}$ , plus la couleur s'approche du blanc :



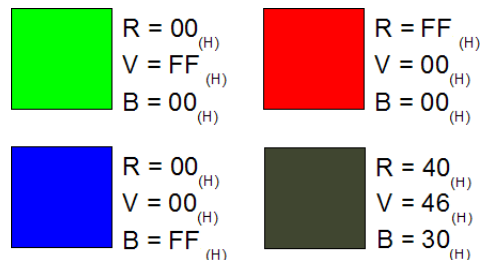
Valeurs des octets, dans l'ordre :  $00_{(H)}$  -  $40_{(H)}$  -  $80_{(H)}$  -  $C0_{(H)}$  -  $FF_{(H)}$

- **Les images 256 couleurs** : à la manière des images uniquement en nuances de gris, on a défini, parmi le grand nombre de couleurs existantes, 256 couleurs que l'on a assignées chacune à une valeur comprise entre  $00_{(H)}$  et  $FF_{(H)}$ . Ces valeurs sont enregistrées dans une palette de couleurs (Look-Up Table, ou LUT). Chacune des valeurs est alors assignée à l'adresse qui lui correspond dans la LUT. Ainsi, à chaque pixel est associé un octet contenant l'adresse de la couleur voulue. De ce fait, chaque pixel occupe un octet de mémoire. C'est pour cette raison que les images en 256 couleurs sont aussi appelées « images 8 bits ». Le volume occupé par l'image en octet est donc égal à la somme du nombre de pixel et du volume de la palette de couleurs en octets.

### 1.2.2 Les images en couleurs vraies

Avec les couleurs indexées, on ne représentait qu'un nombre restreint de couleurs pour chaque image. Dans le cas d'images simples, nécessitant peu de couleurs, ou pour les distribuer sur Internet ou un réseau, ces représentations sont satisfaisantes car elles occupent peu de volume sur disque dur. Cependant, ces représentations ne sont pas toujours très fidèles à la réalité, car le nombre de couleur est restreint. Pour rendre les images plus réalistes, il faut coder chaque pixel indépendamment des autres, en lui attribuant un grand nombre d'informations concernant sa couleur. Il existe trois principales échelles de codage des couleurs :

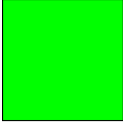
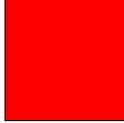
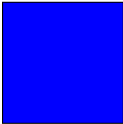

- Les échelles additives : Ces méthodes de codage consistent à combiner les effets de réflexion des différentes couleurs, afin de composer des couleurs plus variées. Il existe deux moyens principaux pour coder les couleurs vraies en échelles additives :
  - *Le codage RGB* : Les trois lettres RGB signifient en fait Red-Green-Blue en anglais, pour Rouge-Vert-Bleu. En effet, chaque pixel est codé sur 3 octets : le premier définit la quantité de rouge, le second la quantité de vert et le troisième la quantité de bleu, chaque fois pour le pixel concerné. Ainsi, chacune des trois composantes est affectée à une valeur comprise entre  $00_{(H)}$  et  $FF_{(H)}$  et est donc présente en plus ou moins grande quantité. En faisant varier chacune de ces trois quantités, il est possible de représenter  $256^3$ , soit 16 777 216 couleurs différentes. Cependant, chaque pixel nécessitant 3 octets de codage, l'espace occupé par une image devient vite très important. Le RGB étant une échelle additive, si les trois composantes R, G et B sont présentes en quantité maximale (et donc ont pour valeur  $FF_{(H)}$ ), la couleur obtenue au final est le blanc.







- *Le codage par Luminances-Chrominances* : On définit ici trois variables : Y, U et V. U et V sont les chrominances de chaque pixel et Y leur luminance. Ce codage découle en fait du codage RGB, et les valeurs Y ; U et V se calculent suivant le système d'équations suivant :

$$\begin{cases} Y = 0.299 \times R + 0.585 \times G + 0.114 \times B \\ U = -0.147 \times R - 0.288 \times G + 0.435 \times B \\ V = 0.615 \times R - 0.515 \times G - 0.100 \times B \end{cases}$$

- Les échelles soustractives : Contrairement aux échelles additives qui combinent les effets de réflexion des couleurs, les échelles soustractives combinent leurs effets d'absorption. L'échelle soustractive la plus utilisée est la CMJN.
  - *Le codage CMJN* : Les quatre lettres CMJN signifient Cyan-Magenta-Jaune-Noir. En effet, en quantités égales, les trois couleurs cyan, magenta et jaune créent une couleur noire. Cependant, la coloration CMJN étant essentiellement utilisée pour l'impression, la création du noir nécessiterait de grandes quantités d'encre. Afin d'accélérer le processus d'absorption, une composante noire a été ajoutée pour la composition des couleurs les plus foncées :

	$C = FF_{(H)}$ $M = 0_{(H)}$ $J = FF_{(H)}$ $N = 00_{(H)}$		$C = 00_{(H)}$ $M = FF_{(H)}$ $J = FF_{(H)}$ $N = 00_{(H)}$
	$C = FF_{(H)}$ $M = FF_{(H)}$ $J = 00_{(H)}$ $N = 00_{(H)}$		$C = 75_{(H)}$ $M = 00_{(H)}$ $J = 97_{(H)}$ $N = 89_{(H)}$

- Le codage par familles : Plus proche du fonctionnement de l'oeil humain, le codage par famille, aussi appelé HSV (pour Hue Saturation Value, Teinte-Saturation-Lumière) trie les couleurs selon leur famille : rouge, vert ou bleu : c'est leur teinte. La saturation désigne la quantité de blanc contenue dans la couleur. Bien sûr, plus la saturation est élevée, plus la couleur est claire. Ainsi, un rouge saturé à 50 % devient rose. Enfin, la Lumière d'une couleur définit sa luminosité et sa brillance.

	$H = 50_{(H)} = 80_{(10)}$ $S = F0_{(H)} = 240_{(10)}$ $V = 78_{(H)} = 120_{(10)}$		$H = 0_{(H)} = 0_{(10)}$ $S = F0_{(H)} = 240_{(10)}$ $V = 78_{(H)} = 120_{(10)}$
	$H = A0_{(H)} = 160_{(10)}$ $S = F0_{(H)} = 240_{(10)}$ $V = 78_{(H)} = 120_{(10)}$		$H = 47_{(H)} = 71_{(10)}$ $S = 65_{(H)} = 101_{(10)}$ $V = 4E_{(H)} = 78_{(10)}$

- Gestion de la transparence : pour certaines images, la transparence peut se révéler utile. Pour les images à palettes de couleurs, un pixel transparent est en fait un pixel auquel n'est affectée aucune couleur. Pour les images codées en RGB, il est possible de rajouter un quatrième octet, appelé canal alpha, pour chaque pixel. Cet octet indique, sur 256 niveaux, le degré de transparence du pixel en question. C'est très utile, par exemple, sur les sites Internet dont l'arrière plan est en couleur : la transparence de certains pixels permet de donner l'illusion que le contour de l'image est identique à l'arrière plan.

### 1.2.3 Le format BITMAP

Le format bitmap (d'extension BMP) a été développé conjointement par BMP et Microsoft. Il est composé de plusieurs sections :

- L'entête du fichier (file header) : Il est composé de 4 champs, qui contiennent :
  - *La signature de fichier*, codée sur 2 octets, indique qu'il s'agit d'un fichier BMP. Elle peut prendre plusieurs valeurs possibles, selon le bitmap dont il s'agit :
    - $424D_{(H)}$  s'il s'agit d'un bitmap Windows ;
    - $4241_{(H)}$  s'il s'agit d'un bitmap OS/2 ;
    - $4350_{(H)}$  s'il s'agit d'un pointeur de couleur OS/2 ;
    - $4943_{(H)}$  s'il s'agit d'une icône OS/2 ;
    - $5054_{(H)}$  s'il s'agit d'un pointeur OS/2 ;
  - *La taille du fichier* : Elle est codée sur 4 octets ;
  - *Un champ réservé*, aussi codé sur 4 octets ;
  - *Le décalage (offset) de l'image*, indiquant l'adresse du début du corps de l'image par rapport au début du fichier.
- L'entête du BITMAP (Information header) : Il contient les informations relatives au corps de l'image. Il est composé des champs suivants :
  - La taille de l'information header, codée sur 4 octets, qui peut prendre les valeurs suivantes :
    - $28_{(H)}$  pour Windows ;

- $0C_{(H)}$  pour OS/2 1.x ;
- $F0_{(H)}$  pour OS/2 2.x ;
- La largeur de l'image, codée sur 4 octets, qui correspond au nombre de pixels horizontaux ;
- La hauteur de l'image, codée sur 4 octets, qui correspond au nombre de pixels verticaux ;
- Le nombre de plans qui composent l'image, sur 2 octets, dont la valeur est toujours 1 ;
- La profondeur du codage de la couleur, cela correspond au nombre de bits composant la couleur d'un pixel. Elle peut valoir 1, 4, 16, 24 ou 32 bits ;
- La méthode de compression, codée sur 4 octets, qui peut valoir :
  - $0_{(H)}$  si l'image n'est pas compressée ;
  - $1_{(H)}$  s'il s'agit d'une compression RLE de 8bits/pixel ;
  - $2_{(H)}$  s'il s'agit d'une compression RLE de 4bits/pixel ;
  - $3_{(H)}$  s'il s'agit d'une compression bitfields.
- Les dimensions totales de l'image qui sont codées sur 4 octets ;
- La résolution horizontale de l'image, qui, codée sur 4 octets, indique le nombre de pixels par unité de longueur, souvent le pouce ;
- La résolution verticale de l'image, qui, codée sur 4 octets, indique le nombre de pixels par unité de longueur, souvent le pouce ;
- Le nombre de couleurs que contient la LUT ;
- Le nombre de couleurs importantes dans la LUT ; cette valeur vaut 0 si toutes les couleurs ont la même importance.
- **La LUT (Look Up Table ou palette de couleurs)** : Elle est définie sur 4 octets pour chaque couleur de la palette. Un octet définit la quantité de rouge, le second la quantité de bleu et le troisième indique la quantité de vert tandis que le dernier octet est réservé.
- **Corps de l'image** : Les pixels y sont écrits ligne par ligne, en partant du pixel en bas à gauche jusqu'au pixel en haut à droite. Ils sont codés selon le nombre de couleurs définies dans l'entête du fichier Bitmap et en référence aux couleurs définies dans la LUT. Chaque ligne est composée d'un nombre entier d'octets. Si ça n'est pas le cas, des bits de valeur 0 sont ajoutés pour compléter les octets incomplets.

## 1.3 Généralités concernant la compression

### 1.3.1 Evaluer la compression

Il existe un certain nombre d'outils qui permettent de mesurer l'efficacité des algorithmes selon plusieurs critères :

- **Le quotient de compression** : il se calcule en divisant l'espace d'origine occupé par l'image par sa taille après compression :

$$Q_{compression} = \frac{TailleInitiale}{TailleFinale}$$

- **Le taux de compression** s'exprime en général en pourcentages et est égal à l'inverse du quotient de compression :

$$T_{compression} = \frac{1}{Q_{compression}} = \frac{TailleFinale}{TailleInitiale}$$

- **Le gain de compression** : il s'exprime également en pourcentages et se calcule selon l'équation :

$$G_{compression} = 1 - T_{compression} = \frac{TailleInitiale - TailleFinale}{TailleInitiale}$$

- **L'erreur quadratique moyenne** : Elle permet de mesurer l'impact des algorithmes non-conservateurs sur les images. Elle se calcule avec la formule suivante :

$$EQM = \frac{1}{N} \sum_{i=1}^{i=N} (n'_i - n_i)^2$$

Avec  $N$  le nombre d'éléments compressés (dans notre cas, des pixels) ;  $n'_i$  un élément (donc un pixel) compressé, obtenu à partir du pixel d'origine  $n_i$ .

Un algorithme de compression efficace a donc un taux de compression élevé et une *EQM* faible. Cependant, nous verrons que souvent les algorithmes non-conservateurs détruisent les éléments auxquels nous sommes peu sensibles. De ce fait, les images peuvent avoir une *EQM* relativement élevée et être cependant de qualité tout à fait correcte.

### 1.3.2 La compression RLE

Le sigle RLE signifie Run Length Encoding. Cette méthode de compression peut être appliquée sur plusieurs types de données, dont les images. Nous étudierons ici uniquement le principe général avant de nous intéresser à la compression RLE sur les images dans la partie 2.1.

L'objectif est de repérer, dans une suite d'éléments à compresser, des valeurs égales pour les assembler, à la manière d'une factorisation algébrique.

Pour exemple, la chaîne de caractère "AAABBBCCRDDDEEE" compressée en RLE donnerait "3A 3B 1C 1R 3D 3E", soit 12 caractères nécessaires au lieu de 14, ce qui correspond à un gain de compression de 14 %.

Nota : Des espaces ont été ajoutés pour faciliter la lecture, mais ne figurent en aucun cas dans le code réel.

Cependant, la méthode RLE peut montrer ses limites pour les suites ayant peu de caractères redondants : "ABCDEPP-PDAZS" donnerait, après application de l'algorithme "1A 1B 1D 1E 3P 1D 1A 1Z 1S". On code ici 18 caractères au lieu de 13, soit un gain de compression de -27 % !

Cette méthode de compression peut donc se révéler très performante lorsque les éléments redondants sont nombreux, mais dans le cas contraire, elle peut entraîner une perte de place. Afin de ne pas subir cette perte causée par la présence de caractères uniques, des règles ont été définies pour l'application de l'algorithme RLE :

- Lorsque trois éléments consécutifs ou plus se suivent, la compression RLE est appliquée.
- Si moins de trois éléments identiques se suivent, la valeur  $00_{(H)}$  est insérée, suivie du nombre d'éléments différents consécutifs. Enfin, ces éléments sont codés comme si aucune compression n'avait eu lieu.
- Certains bits spéciaux sont codés :
  - $00\ 01_{(H)}$  pour indiquer une fin de ligne ;
  - $00\ 00_{(H)}$  pour indiquer la fin du fichier à compresser.

Si nous reprenons l'exemple précédent, la chaîne de caractères "ABCDEPPPPDAZS" serait codée ainsi : "00 5 ABCDE 3P 00 4 DAZS 00 00". Les valeurs "00 5" indiquent que la compression ne doit pas se faire sur les 5 caractères qui suivent. Les caractères en question sont donc codés comme s'ils n'étaient pas compressés. Ensuite, comme il y a 3 "P" consécutifs, la méthode RLE est appliquée. Enfin, les valeurs "00 4" empêchent la compression sur les 4 lettres suivantes, puis l'octet 00 00 termine la chaîne. Si on néglige le caractère  $00\ 00_{(H)}$  indiquant la fin de la chaîne à coder, il faut coder 17 caractères au lieu de 12. La perte de place est atténuée mais reste malgré tout importante. Cependant, la chaîne "AAABBBCCRDDDEEE" retournerait le code "3A 3B 00 2 CR 3D 3E", soit 13 caractères au lieu de 14. Le gain de compression n'est alors pas très important, mais il existe des suites d'éléments dans lesquelles plus de 50 valeurs consécutives sont égales : la compression RLE prend alors tout son intérêt.

### 1.3.3 L'algorithme Huffman

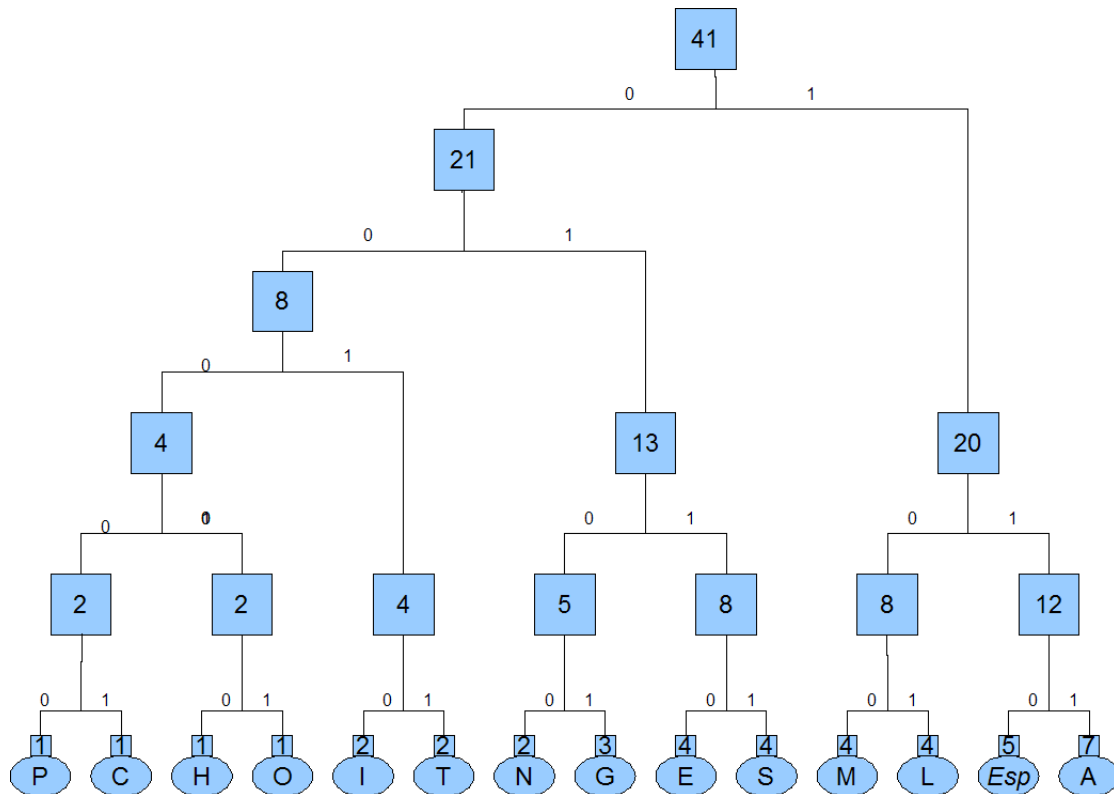
Le codage Huffman, du nom de celui qui l'a inventé, consiste à réaliser une statistique de la fréquence d'apparition de chaque valeur du fichier à compresser. Plus un élément apparaît fréquemment, moins il occupera de bits lors du codage. Il faut donc déterminer quel sera le code de chaque élément. Pour ce faire, on construit l'arbre de Huffman selon la méthode suivante :

- On calcule la fréquence d'apparition de chaque caractère (que l'on nomme aussi poids) et on la classe par ordre croissant.



- On assemble les caractères de plus faibles fréquences par paires. La somme des deux fréquences assemblées constitue un noeud, dont la valeur est égale à la somme des poids qui l'ont composé. Le lien entre deux poids est appelée branche.
- On répète le même processus jusqu'à ce qu'il ne reste qu'un seul poids. On affecte ensuite à chaque branche dirigée vers la gauche un bit de valeur 0 et à chaque branche dirigée vers la droite un bit de valeur 1 (ou inversement).
- Pour connaître le code correspondant à chaque caractère, il faut partir de la racine puis descendre branche par branche vers le caractère recherché, **en utilisant toujours le chemin le plus court**. Les bits de poids forts sont ceux qui sont les plus proches de la racine de l'arbre.

L'arbre de Huffman correspondant à la chaîne de caractères "PAGES CLEMENT GAMA MELISSA GALLINA THOMAS" est donc (les espaces sont notés *Esp*) :



Les caractères M, L, *espace* et A sont les plus fréquents, ils sont codés sur 3 bits. Les caractères I, T, N, G, E et S sont codés sur 4 bits et les caractères P, C, H et O sur 5 bits. La lettre A est codée avec les valeurs  $111_{(2)}$  et la lettre P avec les bits  $00000_{(2)}$ .

Notons que le codage Huffman peut être réalisé selon 3 méthodes différentes :

- **La méthode statique** : Un arbre est codé à l'avance, indépendamment du fichier. Il est ensuite enregistré dans l'en-tête du fichier en question. L'inconvénient est qu'il ne peut être utilisé que pour certains types de fichier (car par exemple la redondance des lettres "e" en français n'est pas du tout la même en anglais ou en allemand). L'arbre n'est jamais modifié d'un fichier à l'autre.
- **La méthode semi-adaptative** : C'est la méthode employée ici : le fichier à compresser est lu en entier et les fréquences de chacun des éléments sont relevées pour construire l'arbre. Il est donc nécessaire d'enregistrer dans l'en-tête du fichier en question les correspondances entre les valeurs réelles et les valeurs codées dans le fichier.
- **La méthode adaptative** : L'arbre est construit au fur et à mesure de la lecture du fichier, et est donc modifié à chaque élément relevé. Il est en conséquence plus long à mettre en place, mais permet de gagner davantage de place car il n'est pas nécessaire, dans ce cas, d'enregistrer l'arbre au début du fichier.

### 1.3.4 Les algorithmes à dictionnaire : la compression LZW

L'algorithme LZW est une méthode dite à dictionnaire, c'est-à-dire qu'elle remplace les séquences les plus fréquentes présentes dans une suite d'éléments par un code équivalent mais plus court. Il existe plusieurs versions de cet algorithme :

- **Les compressions LZ77 et LZ78** : Elles ont été développées respectivement en 1977 et 1978 par Abraham Lampel et Jacob Ziv qui l'ont nommée de leurs initiales. Elles consistent à diviser la chaîne à compresser en deux parties distinctes :
  - **Le dictionnaire** : il contiendra, après compression, des codes raccourcis pour les éléments les plus fréquents, à la manière de la compression Huffman, sauf que les fréquences ne sont pas prises en compte.
  - **Le corps de la chaîne à compresser**, aussi appelée **tampon de lecture** contient les éléments à compresser.

Un curseur virtuel se déplace dans la chaîne à compresser et associe à chaque valeur qu'il n'a pas encore lue deux variables :

- $i$  : le nombre de caractères qui séparent la valeur du dictionnaire de la valeur correspondante lue dans le tampon.
- $j$  : le nombre de caractères occupés par la valeur lue.

Une fois qu'une des valeurs est lue, le curseur se déplace de  $j + 1$  caractères afin de reproduire le même processus pour les caractères non lus. La particularité de la construction du dictionnaire est que les chaînes lues ne sont pas copiées dans le dictionnaire : la chaîne à compresser y est directement intégrée : les premiers caractères font donc partie à la fois du dictionnaire et de la chaîne à copier. Le curseur de lecture démarque en fait la fin du dictionnaire, et comme ce dernier se déplace, les premiers caractères appartiennent au dictionnaire.

Si nous compressons la chaîne "ESSAI ESSAI " (avec un *espace* après le deuxième "ESSAI") avec l'algorithme LZ78, nous avons la chaîne ESSAI {6 6} : le premier E est affecté à l'ensemble {0 ; 0}, le premier S à l'ensemble {0 ; 0}. A première vue, il nous a semblé que le caractère S devait être associé aux valeurs {1 ; 0} car nous croyions que le curseur ne bougeait pas. Or, ce dernier se déplace de  $j + 1$  caractères, donc ici de 1 caractère. Le second S est temporairement associé aux valeurs {1 ; 1} car il y eu un premier S codé : le premier 1 l'indique, le second indique la longueur de la répétition. Le curseur va donc se déplacer de  $j + 1$  caractères, donc ici de 2 cases. On arrive donc à la lettre A, qui est affectée aux nombres {0 ; 0} ; puis à la lettre I associée aux valeurs 0 ; 0. Enfin, l'espace est aussi affecté aux nombres {0 ; 0}.

Cependant, le deuxième "ESSAI" a déjà été entièrement lu précédemment. Il est donc inutile de le coder entièrement. Il suffit d'ajouter les valeurs {6 ; 6} à la suite du premier mot ESSAI. Ainsi, le curseur se déplacera de 7 cases vers l'arrière et copiera les 6 caractères suivants automatiquement. Il est intéressant de constater que le codage du deuxième S dans le premier "ESSAI" a été ignoré dans le second mot : la priorité est donnée à la chaîne la plus longue.

- **La compression LZW** : Inventé en 1984 par Terry Welch, le principe de l'algorithme LZW est le même que celui des algorithmes LZ77 et LZ78. La seule différence est que le dictionnaire est généré dynamiquement au fur et à mesure de la lecture du fichier à compresser. Cela permet d'avoir non plus deux valeurs à enregistrer pour chaque répétition, mais une seule. En effet, la longueur de la répétition se révèle alors inutile.

L'algorithme LZW se révèle donc très performant pour les fichiers très volumineux, car les répétitions ont alors plus de chances d'être nombreuses, ce qui donne tout son intérêt au dictionnaire.

Il peut être utilisé pour de nombreux types de fichiers, et pas uniquement les images. On s'en sert notamment dans les systèmes de compression ZIP et RAR. Son application aux images numériques se fait par les formats GIF et PNG.

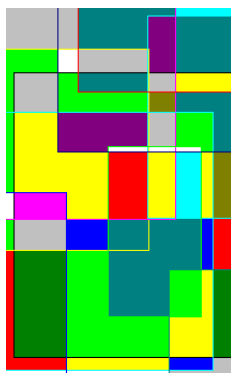


A l'origine, elle occupait 39.1 Ko. Après application de la compression RLE, elle nécessite 28.5 Ko, soit un gain de compression de 27 %, ce qui se révèle être assez peu. L'algorithme RLE n'étant pas destructeur, l'image est conservée à l'identique, mais le nombre de pixels consécutifs identiques est peu élevé, ce qui limite l'efficacité de l'algorithme. Dans notre exemple, il y a malgré tout un certain nombre de pixels qui peuvent être compressés, mais il arrive parfois que les images compressées en RLE occupent davantage d'espace que leur homologue non-compressé, essentiellement pour les images codées sur 24 bits. C'est le cas, par exemple, pour l'image "dégradé.bmp" (lisible sur le CD-ROM joint à ce dossier dans le répertoire *TPE : La compression des images numériques \ Partie 2*, qui occupait à l'origine 5.04 Mo et nécessite après compression 6.53 Mo.

La compression RLE n'est donc pas adaptée aux images en couleurs vraies car les pixels redondants peuvent y être assez rares.

### 2.1.3 Application du RLE sur les images en couleurs indexées

Après avoir appliqué la méthode RLE sur une image 24 bits, nous avons décidé de réaliser les mêmes essais sur une image 16 couleurs :



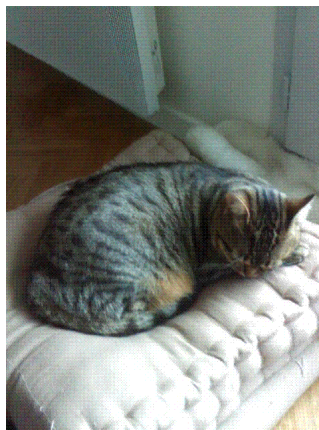
Cette image occupait à l'origine 36 Ko et occupe après compression 8.85 Ko, soit un gain de compression de 77.5 %. Une telle économie de place est facile à expliquer : les pixels identiques sont très fréquents, et même si lorsqu'on change de couleur, on est obligés d'ajouter des valeurs non compressées, la place économisée par les couleurs identiques est de loin supérieure à celle utilisée par les caractères spéciaux tels que  $00_{(H)}$ .

## 2.2 Implémentation des algorithmes LZW

La compression LZW s'applique sur les images, essentiellement au format GIF, dont la société créatrice détient les droits d'exploitation de l'algorithme. Le GIF étant limité aux images 256 couleurs, l'algorithme LZW est aussi utilisé pour les formats PNG 24 bits. Nous allons donc étudier le gain de place réalisé par la compression LZW avec une image 8 bits et une image 24 bits.

### 2.2.1 Compression LZW d'une image 256 couleurs

Nous avons décidé de tester l'efficacité de la compression LZW sur les images monochromes (256 couleurs). Nous avons donc pris une photographie que nous avons enregistrée au format bmp 256 couleurs, puis nous l'avons ensuite compressée en LZW. Voici l'image que nous avons compressée :

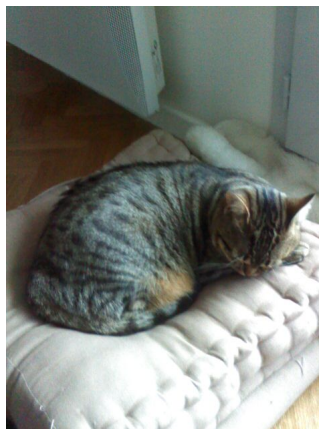


Nota : L'image est de qualité relativement médiocre car elle était à l'origine encodée en 24 bits, mais nous l'avons enregistrée en 8 bits pour pouvoir comparer avec fiabilité la taille compressée et la taille non compressée.

L'image non compressée occupe 301 Ko et l'image compressée 77.3 Ko, soit un gain de compression de 74 % ! La méthode de compression LZW est donc de toute évidence particulièrement efficace pour les images monochromes. Cependant, son utilisation est limitée par les droits d'auteurs qui empêchent les formats de l'utiliser également. Le format PNG a cependant réussi à développer un algorithme basé sur le même principe, qui peut par conséquent être utilisé sur des images en couleurs vraies.

### 2.2.2 Compression LZW d'une image en couleurs vraies

Nous avons compressé la même image en LZW, mais cette fois codée avec 24 bits par pixel. Au format bitmap non compressée, elle occupe 900 Ko. Une fois compressée en LZW grâce au format PNG, on obtient une image qui occupe 625 Ko. Le gain de compression est alors de 30.5 % L'image compressée est la même que précédemment, mais elle est d'une qualité sensiblement supérieure car elle est codée sur 24 bits par pixel :

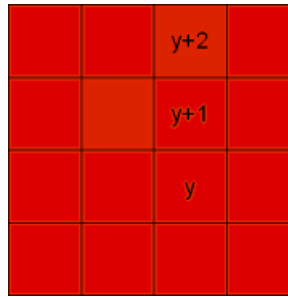


## 2.3 Le codage prédictif

### 2.3.1 Principe du codage prédictif

Comme indiqué dans son nom, le codage prédictif effectue une prédiction. On sait que les intensités des pixels en couleurs vraies varient progressivement, c'est-à-dire que deux pixels voisins ont de grandes chances de partager une ou plusieurs valeurs égales lors du codage, et ce quelle que soit la manière de coder les couleurs des pixels, RGB ou CMJN.

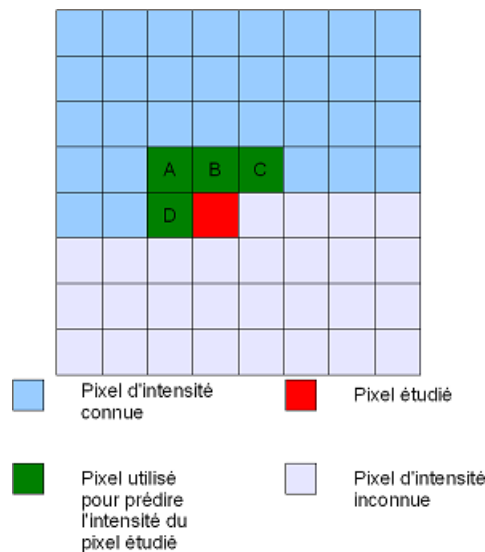
Si nous prenons l'exemple de deux pixels de coordonnées respectives  $(x; y)$  et  $(x; y + 2)$  ayant tous les deux une composante rouge égale à  $C8_{(H)}$  dans une image codée en RVB, on peut déduire que le pixel de coordonnées  $(x; y + 1)$  aura une valeur très proche, sinon égale à celle de ses voisins :



L'objectif est donc de stocker la différence entre la prédiction de cette valeur et la valeur réelle, qui est généralement très faible.

### 2.3.2 Réalisation d'une prédiction sur un pixel

Le choix du voisinage est très important dans le codage prédictif puisque cette étape influe sur l'efficacité de l'algorithme. On sait cependant que le pixel doit être reconstruit en utilisant les différentes valeurs ainsi que les erreurs des pixels déjà reconstruits, voici un exemple de voisinage qui, très commun, offre néanmoins de bonnes performances :



Dans ce dernier exemple, les pixels de l'image se lisent de haut en bas et de gauche à droite.

Maintenant que nous connaissons les valeurs réelles de A, B, C et D, plusieurs chemins sont possibles pour réaliser la prédiction du pixel étudié : en effet, nous allons accorder plus ou moins d'importance aux pixels du voisinage selon leur position par rapport au pixel de base ; on pourrait même en délaisser certains.

Dans tous les cas, le principe est le même : on calcule une moyenne pondérée dont les coefficients sont choisis selon la proximité des pixels utilisés par rapport au pixel recherché. Plusieurs calculs sont alors possibles :

$$\frac{A+B+C+D}{4}$$

Ici la même importance est accordée à chacun des pixels adjascents au pixel rouge.

On peut aussi décider arbitrairement d'affecter un coefficient supérieur aux pixels A et B. On calculerait alors :

$$\frac{2A+2B+C+D}{6}$$

Une autre possibilité est de ne pas prendre en compte les pixels A et C. Auquel cas, le calcul serait le suivant :

$$\frac{B+D}{2}$$

Notons que pour les pixels en bordure de l'image, les pixels d'abscisses  $x + 1$  sont considérés comme ayant une valeur nulle.

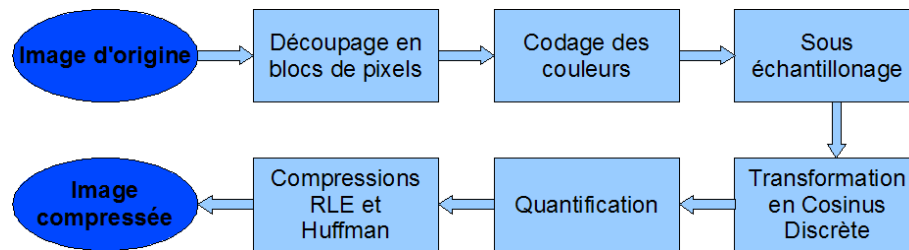
Si on examine bien le principe de coder la différence entre la valeur prédite et la valeur réelle (c'est-à-dire l'erreur commise), on remarque que cette valeur ne peut être utilisée seule : si un pixel est codé sur un octet, soit 256 niveaux, et que l'algorithme a prédit une valeur de  $00_{(H)}$  au lieu de  $FF_{(H)}$ , on devra enregistrer  $FF_{(H)}$ , et à l'inverse si l'algorithme a prédit  $FF_{(H)}$  au lieu de  $00_{(H)}$ , on devra enregistrer  $-FF_{(H)}$ . L'amplitude est donc de 511 niveaux, qu'il faudrait enregistrer avec 9 bits au minimum pour prévoir tous les cas possibles, ce qui entraînerait une augmentation de la taille du fichier.

Comme on code la différence entre la prédiction et la couleur réelle du pixel, les données obtenues peuvent être compressées en RLE et avoir un gain de compression assez intéressant, plus particulièrement pour les valeurs nulles. En effet, les erreurs sont souvent égales d'un pixel à l'autre. Par ailleurs, le nombre d'erreurs différentes est généralement assez bas, ce qui permet un codage statistique de type Huffman.

### 3 La compression non-conservatrice : la norme JPEG

#### 3.1 Principe et fonctionnement théorique du JPEG

L'algorithme de compression JPEG, qui est en fait l'acronyme de Join Photographic Expert Group, est né de l'association de la CCITT (*Commission Consultative Internationale de la Télégraphie et de la Téléphonie*) et de l'ISO (*International Standards Organisation*). Son vrai nom est *ISO/IEC IS 10918-1 | ITU-T Recommendation T.81*, mais il reste peu usité. La compression JPEG met en œuvre à la fois des méthodes de compressions conservatrices, telles que les compressions RLE et Huffman, et des méthodes destructrices. Elles se déroulent selon ces six étapes principales, que nous allons détailler :



Notons que les images compressées avec l'algorithme JPEG peuvent être décompressées en reproduisant les étapes inverses dans l'ordre inverse. Cependant, les données perdues ne pouvant être récupérées, elles seront retrouvées par des approximations, ce qui empêche de retrouver une image exactement identique à son originale, même après décompression.

##### 3.1.1 La Transformation en Cosinus Discrète

La Transformation en Cosinus Discrète (abrégiée DCT pour Discret Cosinus Transforming, en anglais) est le processus clé de la compression JPEG. Cette dernière permet de changer le domaine d'étude d'une fonction tout en la conservant à l'identique. Dans le cas d'une image, il s'agit d'une fonction à trois dimensions :  $x$  et  $y$ , qui indiquent la position du pixel et  $z$  la valeur de la couleur du pixel en question. Ces trois variables sont transformées en deux chaînes de fréquences : une chaîne en abscisses et une chaîne en ordonnées. Les hautes fréquences représentent les changements rapides de couleur d'un pixel à l'autre. De tels changements sont assez rares, ce qui permet de représenter une image complète avec peu de valeurs élevées, donc nécessitant peu d'espace sur un disque dur. La DCT se calcule selon l'équation suivante :

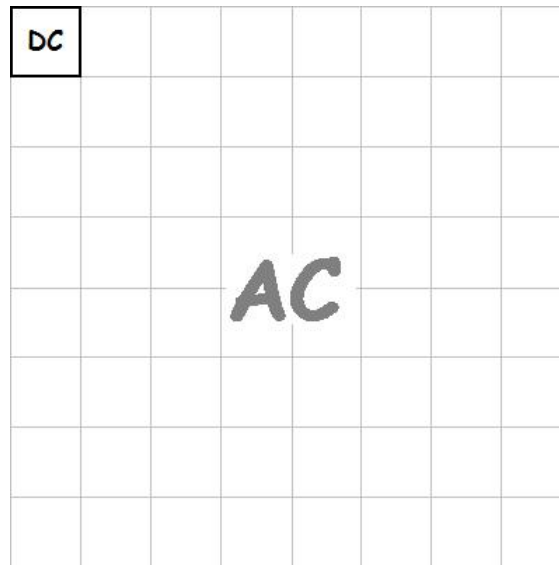
$$DCT(i, j) = \frac{2}{N} C(i) C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \text{pixel}(x, y) \cos \left[ \frac{(2x+1)i\pi}{2N} \right] \cos \left[ \frac{(2y+1)j\pi}{2N} \right]$$

Avec

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

Pour une matrice carrée de 8 fois 8 pixels (nous verrons en effet que la DCT s'applique uniquement pour de telles matrices), le pixel de coordonnées (0 ; 0) est appelé Composant Direct (Direct Component, abrégé DC), et a pour valeur la moyenne de chacun des éléments de la matrice avant l'application de la DCT. Le DC a donc la valeur la plus élevée de la matrice à laquelle il appartient. Les autres composants sont appelés Composants Alternatifs (Alternative Component, AC). Une matrice carrée de 8 fois 8 éléments peut donc se représenter ainsi :





En conséquence de cette transformation, plus on s'éloigne du Composant Direct, plus les fréquences prennent des valeurs élevées.

### 3.1.2 La méthode JPEG

Comme nous l'avons vu, la compression JPEG dépend de six processus indépendants les uns des autres :

- **Le découpage en blocs de pixels** : Nous l'avons montré, la Transformation en Cosinus Discrète implique un grand nombre de calculs très gourmands en ressources et elle ne peut se faire que sur des matrices carrées. Ainsi, réaliser une telle transformation sur une image complète est impossible. Pour contourner le problème, l'image est découpée en groupes carrés de 8 fois 8 (donc de 64) pixels, sur lesquels une Transformation en Cosinus Discrète est appliquée, indépendamment des blocs à proximité.
- **La transformation des couleurs** : La compression JPEG peut être effectuée avec tous les moyens de codages des couleurs, mais est plus efficace avec certains d'entre eux, et notamment avec l'encodage YUV. En effet, ce dernier définit une variable de Luminance et deux variables de Chrominances. Or, l'œil humain est peu sensible à la Chrominance d'une couleur. De ce fait, on peut diminuer la quantité d'informations la concernant sans perte visible. C'est à cette fin que les images sont encodées en coloration YUV avant que d'autres transformations y soient appliquées.
- **Le sous échantillonnage** : Une fois les couleurs codées en YUV, il est possible d'influer sur les Chrominances U et V sans pertes visibles. On calcule ainsi, sur chaque carré de 2 fois 2 pixels, les chrominances moyennes. On a donc, pour chaque carré de 4 pixels, une valeur moyenne de U et une valeur moyenne de V. Cela permet, pour un carré de 4 pixels, de n'avoir à coder que 6 valeurs (une luminance Y par pixel, soit 4 luminances au total, et deux chrominances U et V) au lieu de 12.
- **La Transformation en Cosinus Discrète** : On applique ensuite, pour chaque bloc de 64 pixels, la DCT (Discret Cosinus Transforming) indépendamment des autres blocs. Il faut utiliser une autre méthode pour les pixels en bordures d'images, avec lesquels il est impossible de créer des matrices de 64 pixels. La DCT renvoie une matrice de 64 nombres pour chaque bloc.
- **La quantification** : Elle s'effectue à l'aide d'une matrice supplémentaire, appelée matrice de quantification, composée de coefficients choisis judicieusement par les programmeurs. La quantification consiste à diviser chaque valeur de la matrice à quantifier par la valeur qui lui correspond dans la matrice de quantification. L'objectif est d'atténuer les hautes fréquences obtenues par la DCT, qui nécessitent un grand volume de données lors du codage. Suite à la division de la matrice à quantifier par la matrice de quantification, de nombreuses valeurs tendent vers 0. Les valeurs obtenues étant ensuite arrondies à l'entier de plus proche, elles sont affectées à une valeur nulle. Les hautes fréquences étant essentiellement en bas à droite de chaque bloc de pixels, les seules valeurs importantes sont celles situées en haut à gauche, car elles contiennent les basses fréquences. Cette étape est donc la seule à déteriorer



aussi importantes que la diminution de la résolution. Notons que la division par deux de la résolution consiste en fait à sous échantillonner à la fois la luminance et la chrominance. De plus, comme le montre l'image ci-dessous, si on accentue l'échantillonnage de la chrominance, l'image finit par être composée uniquement de nuances de gris, ce qui prouve que seule la chrominance donne des informations concernant la couleur.



Sous échantillonnage accentué de la chrominance

Par ailleurs, le gain de place n'est pas négligeable lorsque l'on sous échantillonne la chrominance: l'image, qui a pour définition  $36 \times 36$ , était à la base codée sur trois octets par pixel (deux pour les Chrominances U et V et un pour la Luminance Y), soit 3 888 octets nécessaires au total. Une fois les Chrominances sous échantillonnées, elle n'occupe plus qu'un octet par pixel pour la Luminance et deux octets pour chaque ensemble de  $2 \times 2$  pixels pour les Chrominances, soit une moyenne de 12 bits par pixel, ce qui correspond au final à 1 944 octets nécessaires. Le taux de compression est alors de 50%, pour une perte indécélable sans grossissement ! Si on échantillonne la luminance, on passe des 3 888 octets de départ à 2 916 octets au final pour une perte de qualité extrêmement visible.

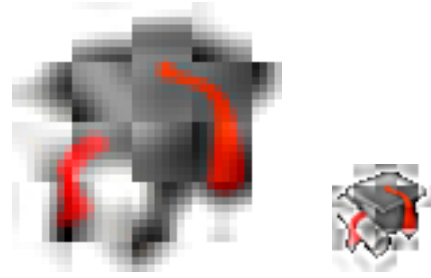
- **Essais de Transformation en Cosinus Discrète :** Afin de mieux appréhender l'effet qu'avait la Transformation en Cosinus Discrète sur les pixels d'une image, nous avons décidé de l'appliquer aussi sur une image, en définissant des valeurs variables pour la Luminance sur laquelle s'applique la DCT. Voici les résultats obtenus, selon les valeurs de chrominances affectés à chacun des blocs de 64 pixels :

1	10000	10000	1	1	1	1	1
10000	10000	1	1	1	1	1	1
10000	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1



Dans l'ordre : Valeurs affectées à la Luminance - Image après transformation, agrandie - Image après transformation, taille réelle

1	1	1	10000	10000	10000	10000	10000
1	1	10000	10000	10000	10000	10000	10000
1	10000	10000	10000	10000	10000	10000	10000
10000	10000	10000	10000	10000	10000	10000	10000
10000	10000	10000	10000	10000	10000	10000	10000
10000	10000	10000	10000	10000	10000	10000	10000
10000	10000	10000	10000	10000	10000	10000	10000
10000	10000	10000	10000	10000	10000	10000	10000
10000	10000	10000	10000	10000	10000	10000	10000



*Dans l'ordre : Valeurs affectées à la Luminance - Image après transformation, agrandie - Image après tranformation, taille réelle*

On constate donc très clairement que nous sommes nettement moins sensibles aux hautes fréquences, qui sont les valeurs les plus éloignées du Composant Direct, qu'aux basses fréquences. En effet, lorsque l'on ne détruit que les cinq valeurs les plus proches du Composant Direct (ce qui correspond à supprimer environ 8 % valeurs) -donc les basses fréquences-, l'image est presque aussi détériorée que lorsque l'on supprime toutes les valeurs sauf les plus proches du Composant Direct (soit environ 90 % des valeurs) -c'est-à-dire toutes les fréquences les plus hautes-. Nous avons aussi essayé de détruire uniquement le Composant Direct, afin de prouver son rôle primordial, et voici ce que nous avons obtenu :

10000	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1



*Dans l'ordre : Valeurs affectées à la Luminance - Image après transformation, agrandie - Image après tranformation, taille réelle*

On voit ici tout à fait clairement que le Composant Direct contient, à lui seul, toutes les informations relatives à la matrice de pixels à laquelle il appartient.

### 3.2.2 Application du JPEG sur une matrice de pixels

Afin de mieux visualiser les transformations successives que subissent les pixels au cours de l'application de l'algorithme JPEG, nous avons décidé de l'appliquer à une matrice de pixels, voici les résultats obtenus :

139	144	149	153	155	155	155	155
144	144	151	151	153	156	156	156
150	155	160	163	158	156	156	156
159	161	162	160	160	159	159	159
159	160	161	162	162	155	155	155
161	161	161	161	160	157	157	157
162	162	161	163	162	157	157	157
162	162	161	161	163	158	158	158

Matrice d'origine

1260	-1	-12	-5	2	-2	-3	1
-23	-17	-6	-3	-3	0	0	-1
-11	-9	-2	2	0	-1	-1	0
-7	-2	0	1	1	0	0	0
-1	-1	1	2	0	-1	1	1
2	0	2	0	-1	1	1	-1
-1	0	0	-1	0	2	1	-1
-3	2	-4	-2	2	1	-1	0

Matrice après DCT

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Matrice de quantification

79	0	-1	0	0	0	0	0
-2	-1	0	0	0	0	0	0
-1	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Matrice quantifiée

La matrice obtenue au final n'a donc rien à voir avec la matrice de départ, et pourtant, la perte de qualité reste peu visible.

Cette matrice est lue selon la méthode "zigzag" puis une compression RLE est appliquée à toutes les valeurs égales à 0. Un caractère spécial, nommé *EOB*, indique la fin d'une matrice de pixels. On obtient donc pour cette matrice, en appliquant les règles de la compression RLE, le code suivant : 00; 8; 79; 0; -2; -1; -1; -1; 0; -1; 57; 0; *EOB*. Le caractère 00 empêche l'application du RLE sur un nombre de valeurs données. Le nombre 8 dit que la compression RLE ne s'applique pas sur les 8 valeurs qui suivent le caractère 00. Les valeurs 57; 0 ne disent pas qu'un élément a pour valeur 57 et un autre pour valeur 0, mais que 57 éléments consécutifs ont une valeur nulle. Enfin, le caractère *EOB* délimite la fin de la chaîne de compression.

Nota : Des espaces et des points-virgules ont été ajoutés pour faciliter la lecture, mais ils ne figurent en aucun cas dans le code réel.

Le même processus est appliqué à chacune des matrices, et le codage complet de l'image est encore compressé grâce à l'algorithme Huffman.

### 3.3 Conséquences sur la qualité des images

Nous avons dit au début que l'algorithme de compression JPEG était destructeur pour les images. Nous pouvons maintenant expliquer pourquoi : les valeurs obtenues lors de la DCT sont arrondies créent les premières inexactitudes, même si elles ne sont généralement pas significatives. Il en est de même pour le sous échantillonnage, qui diminue le nombre de bits contenant la chrominance. Cependant, la quantification consistant en une division " arbitraire " des valeurs

des pixels, elle entraîne une détérioration très importante des pixels en question. C'est pourquoi les taux de compressions doivent être choisis judicieusement selon la qualité voulue, et selon les types d'images. Nous allons étudier les impacts de la compression JPEG sur les images en couleurs vraies (24bits) et sur les images à couleurs indexées.

### 3.3.1 Images en couleurs vraies



*Image originale - Compressée JPEG 90 - Compressée JPEG 70 - Compressée JPEG 50 - Compressée JPEG 10*  
 118Ko                      13.1Ko                      7.48Ko                      5.70Ko                      2.44Ko

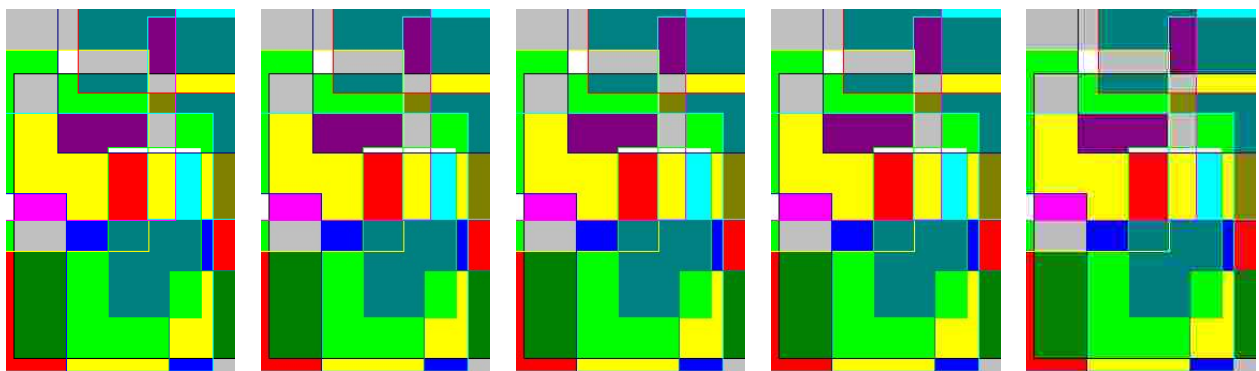
Nota : L'impression du dossier a rendu les détériorations de l'image difficilement perceptibles. Vous pourrez cependant les visualiser en consultant le répertoire *La compression des images numériques\Partie 3\Sous-partie 3.3* enregistré dans le CD-ROM ci-joint.

On voit qu'entre l'image originale et l'image compressée en JPEG 90, l'espace occupé a été divisé par 9, alors que la perte de qualité est invisible. Quelques parasites apparaissent pour la compression en JPEG 70, mais restent peu préjudiciables compte tenu de la taille économisée (l'espace occupé est divisé par 15). Cependant, passée la compression JPEG 70, les pertes deviennent trop visibles. Notamment, la compression en JPEG 10 provoque l'apparition de blocs de pixels qui rendent l'image peu agréable à regarder.

Une compression en JPEG 80 semble donc être un bon compromis entre perte de qualité et économie de place.

### 3.3.2 Images en couleurs indexées

Pour réaliser les tests de compressions à différents taux, nous avons choisi une image en 16 couleurs composée de rectangles et de changements brusques de couleurs. Voici ce que nous avons obtenu :



*Image d'origine - Compressée JPEG 90 - Compressée JPEG 70 - Compressée JPEG 50 - Compressée JPEG 10*  
 23.5Ko                      14.4Ko                      10.4Ko                      8.70Ko                      4.48Ko

Nota : L'impression du dossier a rendu les détériorations de l'image imperceptibles. Vous pourrez cependant les visualiser en consultant le répertoire *TPE : La compression des images numériques\Partie 3\Sous-partie 3.3* enregistré dans le CD-ROM ci-joint.

On voit ici dès la compression en JPEG 90 l'apparition de défauts dans les couleurs, notamment lorsque l'on passe d'un rectangle à l'autre. Le gain de place n'est pas négligeable, mais la perte de qualité devient trop préjudiciable pour rendre la compression JPEG intéressante pour les images de ce genre. De plus, la compression RLE réalisée sur la même image

dans la partie 2.1 a permis d'obtenir un volume de 8.85 Ko sans perte. La perte de qualité s'explique par le fonctionnement du JPEG. En effet, la DCT transforme les différences de couleurs importantes entre deux pixels en hautes fréquences. Or, ces fréquences sont diminuées voire détruites lors de la quantification, ce qui entraîne des dégradations. Lorsque les hautes fréquences sont rares, ces pertes sont peu visibles. Cependant, dans l'image étudiée ici, les hautes fréquences sont nombreuses car les changements de couleurs entre un rectangle et un autre sont rapides.

D'une manière générale, la compression JPEG n'est pas utilisée pour les images composées de peu de couleurs et de formes géométriques car les méthodes de compression conservatrices telles que la RLE ou la LZW donnent des résultats tout aussi satisfaisants, sans détériorer l'image en question.

## CONCLUSION

Les images étant des fichiers qui nécessitent une grande quantité d'espace sur un disque dur, différents algorithmes de compression ont été développés. Chacun d'entre eux exploite différents aspects des images numériques. Ils ont tous leurs particularités et leurs limites. Certains, comme le RLE et le codage Huffman conservent l'image telle qu'elle était à l'origine, mais la codent d'une façon astucieuse, ce qui permet d'économiser de la place.

Cependant, ces algorithmes montrent des limites, surtout lorsque le nombre de couleurs est élevé, comme c'est le cas pour les images 24 bits. Dans ce cas, d'autres moyens sont mis en œuvre, exploitant les limites de notre système oculaire : la compression JPEG détériore les informations des images auxquelles nous sommes peu sensibles, telles que les chrominances et les hautes fréquences obtenues grâce à la DCT. Les taux de compressions sont alors bien plus élevés, mais la qualité de l'image peut s'en ressentir.

Par ailleurs, la compression est en perpétuelle évolution, et de nouvelles méthodes sont en train de voir le jour. La compression JPEG par ondelettes, par exemple, permet d'obtenir des taux de compression comparables à ceux du JPEG avec une détérioration de l'image nettement moins visible.