DODANE Jean-Baptiste
ROUSSILLON Cyril

TS3

TPE 2001 - 2002

LA COMPRESSION GRAPHIQUE INFORMATIQUE

INTRODUCTION

Les images numériques font partie des données dont le stockage nécessite le plus de place. Ainsi une image de qualité photographique (600 points par pouce), de dimensions 10 x 15 cm, occupe environ 25 méga-octets, soit une suite de plus de 200 millions de 0 et de 1 !

Des problèmes se sont alors posés pour le stockage et surtout la transmission de ces images dans plusieurs domaines. En astronomie, pour la transmission des photographies prises par les sondes, dans les appareils photographiques numériques, qui disposent d'une quantité limitée de mémoire, et bien sûr pour Internet.

Une technique alors employée est la compression des informations, qui consiste à réduire la taille des données sans en perdre le contenu.

Plusieurs techniques existent, répondant à certains critères : **difficulté de mise en œuvre**, **efficacité de la compression**, **vitesse d'exécution**, **degré de conservation de l'information**. Selon l'utilisation, tel ou tel critère doit être privilégié, et l'algorithme de compression doit être choisi en conséquent.
Par exemple en astronomie on privilégiera la qualité, et pour les appareils photographiques numériques la difficulté de mise en œuvre, pour ne pas avoir à réaliser de circuits trop complexes.

Nous avons donc décidé d'étudier de plus près les méthodes de compression qui permettent cette opération magique qui consiste à réduire la taille d'une image sans voir de différences … Mais nous ne nous sommes pas arrêtés à une simple étude théorique puisque étant passionnés de programmation informatique, nous avons décidé de programmer nous-mêmes ces algorithmes. Vous pourrez trouver le programme que nous avons réalisé avec son code source complet pour Borland Delphi 4 sur le cd-rom joint, commenté tout au long de ce dossier, ainsi que le code source des principales parties du programme à la fin de ce dossier.

Dans ce dossier nous allons commencer par faire une présentation générale des images en informatique et des algorithmes de compressions généraux, pour ensuite nous intéresser aux méthodes spécifiques à la compression des images, conservatives puis non conservatives.

PLAN DU DOSSIER

1. GENERALITES

Avant tout, il est nécessaire de connaître les unités de stockage en informatique. Le **bit** (Binary Digit) est l'unité de la numération binaire. Il ne peut prendre que deux valeurs : 0 ou 1. Toutes les données informatiques sont enregistrées sous forme de bits, donc sous une suite de 0 et de 1.

On appelle **octet** un groupe de 8 bits. Ainsi, un octet peut prendre 2^8 = 256 valeurs différentes, de 0 à 255 (base 10), et plus couramment de 00 à FF (base 16).

Un caractère est un signe codé avec un octet, d'après le code ASCII (acronyme pour *American Standard Code for Information Interchange*), code standardisé de représentation des caractères alphanumériques. Les caractères sont toutes les lettres de l'alphabet (minuscules et majuscules), les chiffres (0 - 9), et quelques signes spéciaux. Ainsi, le "A" est codé avec 1 octet de valeur 65 (base 10). Pour avoir sa valeur binaire (en bits), on convertit 65 (base 10) en base 2, ce qui donne 01000001. Cette dernière valeur est écrite sur un support informatique (disque dur, disquette, cd-rom, …).

On se rend vite compte que le nombre de bits devient vite élevé quand on tape un texte. C'est pourquoi on utilise souvent les multiples de l'octet. Un **kilooctet**, est égal à $\mathbf{2^{10}}$ octets, soit 1024 octets (et donc 8192 bits). Le **mégaoctet** vaut $\mathbf{2^{20}}$ kilooctets, soit 2^{20} = 1048576 octets. Les disques durs actuels peuvent atteindre plusieurs **gigaoctets**, donc $\mathbf{2^{30}}$ octets soit plus d'un milliard d'octets.

Les valeurs qui seront le plus souvent utilisées au long de ce dossier sont le bit et l'octet (en base 16).

1.1. LE STOCKAGE DES IMAGES INFORMATIQUES

1.1.1. LE MODE DE STOCKAGE

- **LE BITMAP**

On échantillonne l'image en points auxquels on associe une couleur ce qui forme un pixel (contraction de *picture elements*). Ce mode est utilisé par la plupart des formats car il permet de représenter tous types d'images, mais en revanche les images deviennent vite très volumineuses.

- **LE VECTORIEL**

L'image est stockée sous la forme d'une définition mathématique. Par exemple un cercle est stocké sous la forme d'une instruction qui définit sa taille, sa position, sa couleur etc. L'avantage par rapport au bitmap est que les fichiers sont généralement plus petits, il est facile de lui appliquer des modifications et quand on zoome dessus l'image est recalculée en améliorant la précision. En revanche il n'est adapté qu'aux schémas et logos (figures géométriques).

- **LES META FICHIERS**

Les méta fichiers incluent plusieurs types d'informations, qui vont du mélange de bitmap et de vectoriel (pratiquement tous les formats vectoriels) jusqu'à du texte, des commentaires, de l'interactivité …

1.1.2. LA COULEUR

- **LES COULEURS INDEXEES (PALETTES)**

Dans ce mode, les couleurs sont préenregistrées dans une palette (que l'on écrit au début du fichier) et on leur attribue un nombre (elles sont indexées) que l'on retrouvera dans le fichier pour désigner les couleurs. Ainsi cela permet d'enregistrer des couleurs très diversifiées avec un nombre restreint de bits.
On peut distinguer les palettes chromatiques, qui regroupent les couleurs effectivement utilisées par l'image, des palettes systèmes qui regroupent toujours les mêmes couleurs garantissant un affichage fidèle sur n'importe quel écran. Pour ces dernières on utilise principalement les palettes 4 bits (16 couleurs), 8 bits (256 couleurs) ou 1 bit (2 couleurs, noir et blanc).

- **LES COULEURS VRAIES**

On répertorie dans cette catégorie tous les modèles qui permettent de représenter les couleurs fidèlement à la réalité. On peut encore scinder cette partie en plusieurs sous catégorie.

Echelles additives

- RGB (*Red Green Blue*, Rouge Vert Bleu)
Ce modèle est le plus couramment utilisé et sert de référence pour tous les autres modèles. Il consiste en une composition des couleurs primaires additives, en stockant séparément l'importance de chaque composante. Les couleurs peuvent être stockées sur 16 bits (5 bits pour le rouge et le bleu et 6 pour le vert, soit 65536 couleurs) ou plus couramment sur 24 bits (8 bits pour chaque composante, soit 256 nuances par composante et plus de 16 millions de couleurs au total).

- Luminance – chrominance

Ce modèle sépare la luminosité (luminance) d'une couleur de sa coloration (chrominance), en affectant une variable à la première et deux à la seconde. Si l'on ne garde que la luminance, on obtient une image en niveaux de gris. On obtient les paramètres du système luminance – chrominance par une simple combinaison linéaire des intensités de rouge, vert et bleu qui proviennent du modèle RGB, et la conversion est donc réversible. L'avantage de ce modèle est que nous sommes plus beaucoup plus sensibles à la luminance qu'à la chrominance et donc cela va servir pour la compression. Il existe plusieurs variantes de ce modèle : YIQ, YUV, YCbCr, mais nous allons seulement nous intéresser à la dernière car elle utilisée pour le JPEG.

Les paramètres Y (luminance), Cb et Cr (chrominance) sont de simples combinaisons linéaires des intensités de rouge (R), vert (G) et bleu (B) :

$$Y = 0.299\,R + 0.587\,G + 0.114\,B$$
$$Cb = -0.1687\,R - 0.3313\,G + 0.5\,B$$
$$Cr = 0.5\,R - 0.4187\,G - 0.0813\,B$$

On constate que pour la luminance (Y) la somme des coefficients est égale à 1, donc quand toutes les composantes sont au maximum la luminance est au maximum, et pour la chrominance (Cb et Cr) la somme des coefficients est nulle, donc quand toutes les composantes ont la même valeur la chrominance est nulle, ce qui concorde avec la définition du modèle.

La conversion inverse est donnée par :

$$R = Y + 1.402\,Cr$$
$$G = Y - 0.34414\,Cb - 0.71414\,Cr$$
$$B = Y + 1.772\,Cb$$

Echelles soustractives

Le système CMY (*Cyan Magenta Yellow*, Cyan Magenta Jaune) fonctionne de la même manière que le modèle RGB mais est une composition des couleurs primaires soustractives, utilisées en peinture, en photographie et pour les imprimantes. Elles s'obtiennent en soustrayant au blanc le rouge pour le cyan, le vert pour le magenta et le bleu pour le jaune. En pratique pour les imprimantes on utilise le système CMYK (*Cyan Magenta Yellow blacK*, Cyan Magenta Jaune Noir) pour ne pas utiliser trop d'encre.

Echelles par famille

Le système HSV (*Hue Saturation Value*, Teinte Saturation Lumière) est un classement proche de l'humain. La teinte (ou famille) détermine la famille de la couleur : rouge, vert ou bleu, la saturation détermine la « quantité de blanc » dans la couleur, plus la valeur est élevée plus la couleur est « pure » (par exemple un rouge saturé à 50% est un rose), et la lumière détermine la luminosité ou la brillance de la couleur.

La transparence

On distingue deux modes de transparence : la transparence simple est utilisée avec une palette, un des éléments n'est pas une couleur mais une absence de couleur, ce qui permet de « laisser passer » la couleur qu'il y a derrière l'image. La couche Alpha (ou canal Alpha) permet plusieurs niveaux de transparence : on rajoute aux trois octets RGB un octet qui définit sur 256 niveaux le niveau de transparence (par exemple les vitres d'une voiture).

1.1.3. LE FORMAT BMP

➢ PRESENTATION

Le format BMP est le format le plus simple et le plus classique, qui est utilisé comme référence. C'est un fichier *bitmap* qui gère la couleur sous forme de palette ou en couleur vraie. Nous l'avons donc étudié car il nous sert de référence pour nos formats de compression. Nous allons présenter ici les parties qui nous intéressent, car il propose de nombreuses options mais qui ne nous sont pas utiles.

La structure d'un fichier BMP est classique :

- Entête du fichier (*FileHeader*)
- Entête du bitmap (*BitMapHeader*)
- Palette (optionnelle)
- Corps de l'image

➢ ENTETE DU FICHIER

Il est composé de 4 informations :

- Signature : "BM", ou 42 4D en hexadécimal (*2 octets*)
- Taille du fichier en octets (*4 octets*)
- Espace réservé : octets égaux à 0 (*4 octets*)
- Offset de l'image : adresse par rapport au début du fichier du premier octet du bitmap (*4 octets*)

➢ ENTETE DU BITMAP

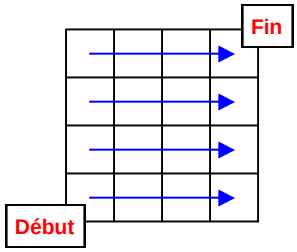
Il suit directement l'entête du fichier. Il donne des informations sur l'image que contient le fichier.

- Taille de l'entête (*4 octets*)
- Largeur de l'image (en pixels) (*4 octets*)
- Hauteur de l'image (en pixels) (*4 octets*)
- Nombre de plans utilisés : toujours égal à 1 (*2 octets*)
- Nombre de bits par pixels (*2 octets*)
- Méthode de compression : 0 pour une image non compressée (*4 octets*)
- Taille finale de l'image (*4 octets*)
- Résolution horizontale (pixels / mètre) (*4 octets*)
- Résolution verticale (pixels / mètre) (*4 octets*)
- Nombre de couleurs de la palette (*4 octets*)

➢ L'IMAGE

Pour les images en couleurs vraies (qui nous intéressent), après l'entête figure l'image : elle est codée selon le modèle RVB (RGB) : c'est à dire que chaque pixel (plus petite partie d'une image) est formé par addition de rouge, vert et bleu. Il est important également de noter que les trois octets sont dans l'ordre Bleu, Vert, Rouge, contrairement par exemple au code HTML.

Il faut également faire attention au sens de codage. En effet nous pensions que l'image était codée depuis le haut à gauche, puis de gauche à droite et de haut en bas. Mais nous nous sommes retrouvés avec des images retournées. En fait le codage se fait depuis le bas à gauche, puis de gauche à droite mais de bas en haut :



Une autre subtilité nous avais aussi échappée : lorsque l'on écrit une ligne de l'image, il faut que le nombre total d'octets soit multiple de 4, et s'il ne l'est pas on complète avec des 0. Nous n'en avions pas tenu compte au début, et donc on interprétait ces 0 supplémentaires comme des pixels ce qui provoquait un décalage pour tout le reste de l'image.

Avec ces informations, il est possible de décoder un fichier *.bmp. Nous avons donc écrit des fonctions pour lire et afficher des bitmaps, et pour les enregistrer. Nous nous servons ensuite de ces fonctions pour charger les images que nous allons compresser avec nos formats.

1.2. ALGORITHMES DE COMPRESSION GENERAUX

1.2.1. ÉVALUATION DE LA COMPRESSION

➢ GAIN DE PLACE

Le degré de réduction des données obtenu par une méthode de compression peut être évalué au moyen du quotient de compression défini par la formule :

$$Q_{comp} = TailleInitiale \,/\, TailleFinale$$

Le *taux de compressio*n, généralement exprimé en pourcentage, est l'inverse du quotient de compression

$$T_{comp} = 1 \,/\, Q_{comp}$$

Le *gain de compression* est également exprimé en pourcentage ; c'est le complément à 1 du taux de compression :

$$G_{comp} = 1 - T_{comp} = (TailleInitiale - TailleFinale) \,/\, TailleFinale$$

Exemple : Un fichier original de 2000 octets compressé en 800 octets présente un quotient de compression de 2.5, un taux de compression de 40 %, et un gain de compression de 60 %.

➢ PERTES

Pour les compressions non conservatives, on utilise une formule pour déterminer la qualité du résultat. On considérera que (n') est une donnée récupérée d'une donnée originale (n).

Si $n_1 = n'_1$, $n_2 = n'_2$ … on a une compression conservative.

Sinon, pour N suite de données, on mesure l'EQM (*Erreur Quadratique Moyenne)* grâce à la formule suivante :

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

➢ CONCLUSION

Un algorithme performant possède donc un quotient de compression maximal, et une EQM minimale.

Toutefois, cette valeur ne reflète pas les informations perçues par l'œil humain. Généralement, lors d'une compression avec pertes, les données perdues sont les moins importantes. L'EQM peut ainsi être élevée et l'image de bonne qualité.

1.2.2. LA COMPRESSION RLE

➢ PRESENTATION

La méthode de compression RLE (*Run Length Encoding*), appelée aussi RLC (*Run Length Coding*), est l'une des plus simples qui soit. Le principe est de remplacer une séquence de *n* éléments *v* identiques par un couple (n, v). C'est un peu comme remplacer des additions par une multiplication.

Exemple : ABBBBCAAACCC donne 1A 4B 1C 3A 3C
De cette manière 10 éléments peuvent en remplacer 12.

(NB : Des espaces ont été insérés pour faciliter la lisibilité de la chaîne. Ils ne figurent en aucun cas dans la suite des éléments compressés).

Cette méthode est efficace dans l'exemple ci-dessus. Le gain de compression est, d'après les formules vues précédemment, égal à ((12 – 10) / 12), soit 16,7 %. Mais lorsqu'il s'agit de coder une chaîne où les séquences répétitives sont rares, on augmente rapidement la taille de l'information source.

Exemple : ABBCBACABBAC donne 1A 2B 1C 1B 1A 1C 1A 2B 1A 1C
Il n'y a que 2 séquences répétitives. Les autres couples ont pour valeur (n) le nombre 1. Coder la lettre "A" par le couple "1A" est donc une perte très importante.

On code ainsi 12 éléments en 20, ce qui est naturellement contraire à la compression.
On obtient ici un gain de compression de ((12 – 20) / 12) = -66,7 %

C'est pourquoi on n'applique le couple *(NombreDeRépétitions, Valeur)* que lorsque le nombre de répétitions atteint un seuil. Dans le paragraphe suivant nous verrons la méthode conventionnelle.

On peut définir les propriétés de cette méthode : Simple – Rapide – Performante. Mais performante seulement dans le cas où l'on retrouve des séquences de mêmes éléments consécutifs dans des données à compresser. Cette caractéristique apparaît très peu dans les fichiers textes (mise en forme : espaces, tabulations). Par contre, dans les images, on obtient souvent des suites de pixels identiques.

➢ CODAGE CONVENTIONNEL

Une compression RLE exige un langage particulier pour être reconnu par les logiciels de traitement graphique (on posera 1 octet = 1 pixel, et nous nous calculerons en base 16) :

Si un pixel est répété 3 fois ou plus, on écrit le nombre d'itérations suivi de sa valeur (loi 1)		
Exemple (1)	07 07 07 07 07	05 07
Exemple (2)	0B 0B 0B 0B 0B 0B 0B 0B 0B 0B	0A 0B
Sinon, la suite n'est pas codée. On écrit 00, puis le nombre de valeurs, puis ces valeurs (loi 2)		
Exemple	FB FB 89 23	00 04 FB FB 89 23
Si cette suite est impaire, on rajoute 00 à la fin (loi 3)		
Exemple	23 65 55 34 22	00 05 23 65 55 34 22 00

Explications *: Soit une image qui comporte 5 pixels consécutifs de la même couleur. Si l'on attribue le code 07 à la couleur du pixel, on obtient comme donnée à écrire 07 07 07 07 07. Étant donné que le nombre de pixels répétés est supérieur à 3, on applique la loi (1), c'est à dire le remplacement de la séquence de n éléments v identiques par le couple (n, v). Le couple équivaut dans l'exemple à (05, 07). L'exemple (2) est similaire. La valeur n est égale à 10. Puisque* $\mathbf{10^{(10)} = 0A^{(16)}}$*, on code (0A, 0B)*

Il existe aussi des codes spéciaux :
- 00 01 = fin de ligne
- 00 00 = fin de bitmap
- 00 02 XX YY = déplacer le pointeur dans l'image de XX colonnes et YY lignes (très peu utilisé)

Donc pour enregistrer un fichier bitmap compressé avec la compression RLE, il faut écrire l'entête classique de 54 octets (en ajustant les octets 30 à 34 en fonction de la compression), puis sauvegarder l'image bitmap en suivant la procédure ci-dessus.
Elle est la même pour les images 4, 8, 24 et 32 bits. Pour les images monochromes, on utilise le système suivant :

Une image monochrome comporte 2 couleurs maximum, le noir et le blanc. Coder un pixel sur 1 octet serait un gaspillage. En effet, chaque bit peut prendre 2 valeurs, tout comme un pixel d'une image monochrome. Non compressé, on a une suite de 0 (pixel noir) et de 1 (pixel blanc) :

000001111011000000001111

Pour compresser, il suffit d'écrire les itérations. En effet, si l'on fixe comme point de départ la couleur 0 (noir), et étant donné qu'il n'y a que 2 couleurs possibles, à chaque changement on connaît automatiquement la nouvelle couleur. On obtient donc :

5 4 1 2 7 4 (d)

Il peut se poser alors, tout comme dans d'autres méthodes de compression, le problème du 256ème pixel. Si la suite comporte plus de 255 pixels de la même couleur, on ne peut évidemment pas inscrire un nombre supérieur à 255 sur un octet. Il faut prendre soin de glisser un 00 tous les 255 pixels.

Exemple : 1111000 … 0011 est codé : 0 4 255 0 145 2 (d) soit 00 04 FF 00 91 02 (h)
(1111000 … : ⟷ x 400)

➢ **BILAN**

La compression RLE, malgré sa simplicité, est présente dans de nombreux formats : le BMP bien sûr, mais aussi le PCX, le TIFF, le IFF (Amiga), le TGA (Targa), le PSD (Photo Shop), le PSP (Paint Shop Pro), le JPEG, et l'incontournable métafichier PDF. On peut aussi citer les peu connus RLE, SGI, RGB, DIB ... formats dérivés des premiers cités.
Étant une compression sans perte, elle peut être utilisée dans de nombreux autres algorithmes, souvent en complément. Elle est simple et rapide, et n'utilise que très peu de ressources.
Ses performances sont tout à fait satisfaisantes (voir les résultats avec le logiciel), mais d'autres méthodes le sont plus. C'est pourquoi elle est surtout utilisée comme solution additionnelle.

1.2.3. LE CODAGE DE HUFFMAN

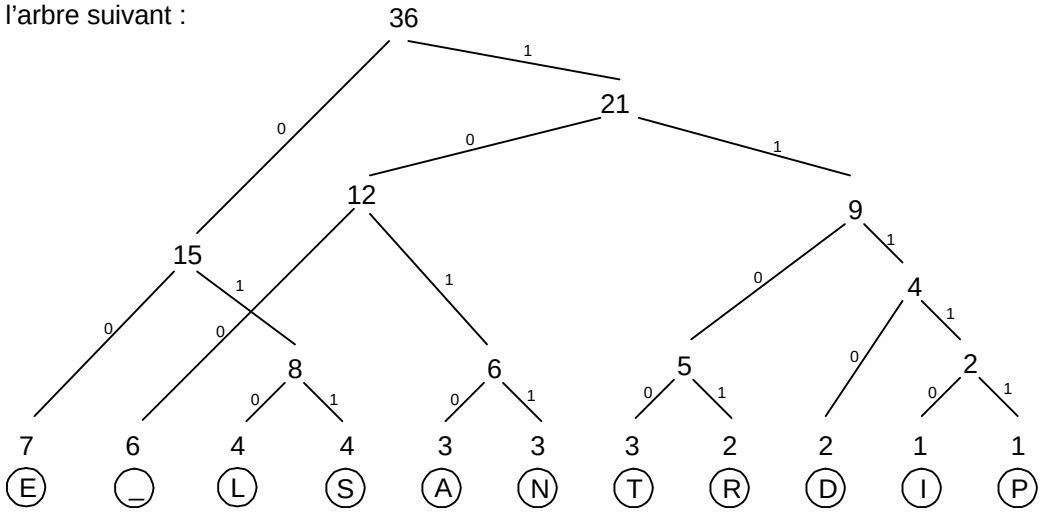
Le codage de Huffman est un codage statistique, c'est à dire qu'il se base sur la fréquence d'apparition d'un caractère pour le coder : plus le caractère apparaît souvent plus son code sera court et vice-versa. C'est pourquoi on appelle aussi ce codage un **VLC** (*Variable Length Code*, code à taille variable) **préfixé**. En effet chaque code n'est le préfixe d'aucun autre, d'où *préfixé*, ce qui permet un décodage unique.

Dans la pratique, pour déterminer le code de chaque caractère on construit l'arbre de Huffman :
- On calcule la fréquence d'apparition de chaque caractère (ou poids).
- On rassemble les deux caractères de plus faible poids pour former un noeud, dont le poids est égal à la somme des poids des deux caractères qui le composent.
- On affecte la valeur 0 au caractère le plus petit et 1 au caractère le plus grand des deux.
- On recommence les deux étapes précédentes en considérant chaque noeud formé comme un caractère, jusqu'à n'avoir plus qu'un noeud, la racine.

Par exemple pour la phrase « LE PRESIDENT EST ENTRE DANS LA SALLE » on a le tableau de fréquences d'apparition suivant :

E	Espace	L	S	A	N	T	R	D	I	P
7	6	4	4	3	3	3	2	2	1	1

Puis on construit l'arbre suivant :



Pour coder une lettre on part de celle-ci et on remonte jusqu'à la racine en relevant le « chemin », et pour décoder on part de la branche et on choisit son « chemin » en suivant les 0 ou 1 du code pour finalement arriver à la lettre décodée.
Ainsi la lettre E la plus fréquente est codée sur 2 bits : 00, le caractère P le moins fréquent est codé sur 5 bits : 11111. On constate qu'ainsi on n'a pas besoin de dire quand le code est terminé puisqu'il est impossible de descendre plus bas quand on est arrivé à une feuille (code préfixé).
La phrase compressée occupe alors (7*2 + 6*3 + 4*3 + 4*3 + 3*4 + 3*4 + 3*4 + 2*4 + 2*4 + 1*5 + 1*5) = 118 bits, au lieu de (36 * 8) = 288 bits, pour un gain de compression de ((288 – 118) / 288) = 59%.

Mais en informatique on ne peut pas modéliser l'arbre de Huffman sous la forme d'un véritable arbre en deux dimensions, car les données sont stockées de manière linéaire dans la mémoire de l'ordinateur. Nous avons décidé de le modéliser avec ses noeuds, un noeud étant une *structure* contenant les adresses de ses deux fils (des liens vers), de son père, son code et son poids (voir code source complet dans la disquette jointe au dossier ou principaux éléments à la fin de ce dossier). On va alors représenter l'arbre sous la forme d'un tableaux linéaire de noeuds, que l'on organisera ainsi (en reprenant les derniers caractères de l'arbre précédent) :

f0: -1 **f1: T** **p:** 7 **c:** 0 *3*	**f0:** -1 **f1: R** **p:** 7 **c:** 1 *2*	**f0:** -1 **f1: D** **p:** 6 **c:** 0 *2*	**f0:** -1 **f1: I** **p:** 5 **c:** 0 *1*	**f0:** -1 **f1: P** **p:** 5 **c:** 1 *1*	**f0:** 3 **f1:** 4 **p:** 6 **c:** 1 *2*	**f0:** 2 **f1:** 5 **p:** 8 **c:** 1 *4*	**f0:** 0 **f1:** 1 **p:** 8 **c:** 0 *5*	**f0:** 7 **f1:** 6 **p:** -1 *9*
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

(T) (R) (D) (I) (P)

A chaque fois que l'on forme un noeud on le rajoute au bout du tableau, on fixe la valeur de ses fils et la valeur père et le code des deux fils (en se référent à l'indice des éléments dans le tableau). Les cinq premiers éléments du tableau (de l'arbre) contiennent les caractères, ce sont les feuilles. Les éléments suivants sont les noeuds et le dernier élément est la racine.

Par exemple pour compresser un I, on va se placer dans l'élément du I (indice 3). On voit que cet élément a pour code 0 donc le premier bit est **0**, puis on va passer dans son père, c'est-à-dire l'élément (le noeud) d'indice 5. On constate que cet élément a pour code 1, donc le bit suivant est **1**, et on passe dans son père (élément 6), son code est **1**, son père est l'élément 8. En arrivant dans l'élément 8, on constate que c'est la racine de l'arbre (car son père a pour valeur -1 que l'on a mise exprès lors de la construction de l'arbre puisque la racine n'a pas de père), et on s'arrête. On a donc le code **011**, que l'on va *retourner* : **110**, puisque pour la décompression on va partir de la racine et faire le chemin inverse.

En décompressant on part donc de la racine, et le premier bit du code est 1, donc on va passer dans le fils 1 du noeud, c'est-à-dire l'élément d'indice 6. Le bit suivant du code est 1 donc on passe dans le fils 1, (l'élément 5), le bit suivant est 0 donc on passe dans l'élément 3. Arrivé dans cet élément on constate que c'est une feuille (car son fils 0 a la valeur -1 que l'on a mise exprès lors de la construction puisque les feuilles n'ont pas de fils), et on va lire dans fils 1 la valeur du caractère (son code ASCII plus précisément) donc le I.

Il existe trois variantes de cet algorithme :
- *Non adaptative :* l'arbre est élaboré à partir d'une table des fréquences fixe ce qui fait qu'il est adapté à un seul type de donnée.
- *Semi adaptative :* comme dans l'exemple, on lit une fois le fichier à compresser pour créer la table des fréquences, puis on s'en sert pour compresser le fichier. C'est la méthode la plus efficace mais elle présente l'inconvénient de nécessiter l'arbre que l'on devra introduire dans un en-tête.
- *Adaptative :* la table des fréquences est élaborée au fur et à mesure de la lecture du fichier et l'arbre est reconstruit à chaque fois. On réalise l'arbre avec les premiers caractères (que l'on ne compresse pas) et on s'en sert pour compresser les caractères suivants que l'on utilise aussi pour actualiser l'arbre etc. Elle est un peu moins efficace que la méthode semi adaptative mais le résultat final est souvent meilleur car elle ne nécessite pas le stockage de l'arbre, et de plus elle ne nécessite qu'une seule lecture du fichier.

1.2.4. LES ALGORITHMES A DICTIONNAIRE (LZ**)

Les algorithmes à dictionnaire, aussi appelés substitution de facteurs, consistent à remplacer des séquences (les facteurs) par un code plus court qui est l'indice de ce facteur dans un dictionnaire.

➢ **LZ77**

En 1977, Abraham Lempel et Jacob Ziv publient l'algorithme à dictionnaire LZ77 qui relance la recherche en matière de compression. La parution de l'ouvrage renfermant les spécificités du LZ77 a eu un effet dévastateur chez les informaticiens : le gain de compression s'est trouvé énormément amélioré.

L'algorithme utilise le principe d'une fenêtre coulissante de longueur ***N*** caractères, divisée en 2 parties, qui se déplace sur le texte de gauche à droite. La seconde partie, qui est la première à rencontrer le premier caractère du texte, contient ***F*** caractères : c'est le tampon de lecture. La première partie, alors égale à (***N - F***) sera appelée ***D***. C'est le dictionnaire.

Initialement, la fenêtre est située ***D*** caractères avant le début du texte, de façon à ce que le tampon de lecture soit entièrement positionné sur le texte, et que le dictionnaire n'y soit pas. Les ***D*** caractères sont alors des espacements.

A tout moment, l'algorithme va rechercher, dans les ***D*** premiers caractères de la fenêtre, le plus long facteur qui se répète au début du tampon de lecture. Il doit être de taille maximale ***F***. Cette répétition sera alors codée par le triplet (***i***, ***j***, ***c***) :

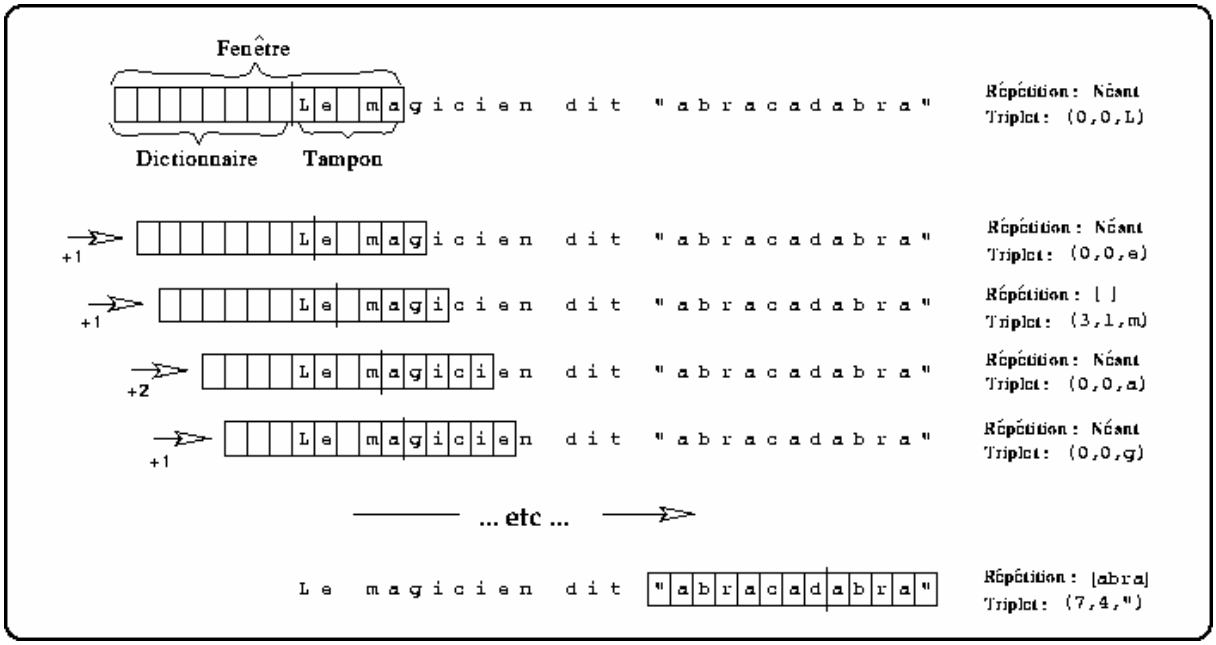
- ***i*** est la distance entre le début du tampon et la position de la répétition dans le dictionnaire.
- ***j*** est la longueur de la répétition.
- ***c*** est le premier caractère du tampon différent du caractère correspondant dans le dictionnaire.

La répétition peut chevaucher le dictionnaire et le tampon de lecture.

Après avoir codé cette répétition, la fenêtre coulisse de ***j* + 1** caractères vers la droite. Le codage du caractère ***c*** ayant provoqué la différence est indispensable dans le cas où aucune répétition n'est trouvée dans le dictionnaire. On codera alors (0, 0, ***c***).

Exemple : Phrase à compresser : Le magicien dit "abracadabra".
Taille de la fenêtre : ***N*** = 13, ***D*** = 8, ***G*** = 5.
(Voir *schéma sur la feuille suivante*)

Explications : Au point de départ, aucune répétition n'est trouvée puisque le dictionnaire ne contient que des espacements. Le codage des deux premiers caractères du texte est alors (0,0,**L**) (0,0,**e**), la fenêtre se déplaçant à chaque fois d'un caractère vers la droite. Ensuite l'espacement est rencontré. Comme il est déjà présent dans le dictionnaire 3 caractères avant le début du tampon, et puisque le caractère **m** termine la répétition, le triplet est (3,1,**m**). On continue alors: (0,0,**a**) , (0,0,**g**) …



On continue ainsi jusqu'à la fin de la séquence à coder.

Le tampon de lecture arrive finalement alors **abra"**. Le plus long mot présent dans le dictionnaire et commençant le tampon contient 4 caractères, c'est **abra**. Dans le dictionnaire, il se situe 7 caractères avant le début du tampon. Le caractère du tampon marquant la différence avec le facteur correspondant du dictionnaire est "
Le triplet est donc (7,4,**"**).

A la fin, on obtient la suite de tous les triplets à coder : (0,0,**L**) (0,0,**e**) (3,1,**m**) (0,0,**a**) (0,0,**g**) (0,0,**i**) (0,0,**c**) (2,1,**e**) (0,0,**n**) (0,0,) (0,0,**d**) (5,1,**t**) (4,1,**"**) (0,0,**a**) (0,0,**b**) (0,0,**r**) (3,1,**c**) (2,1,**d**) (7,4,**"**).

On code ces triplets en binaire, en associant une valeur fixe pour chaque triplet

On remarque que le codage est d'autant plus intéressant que le mot répété est long. Pour avoir des chances d'obtenir de longues répétitions, le dictionnaire doit être de taille suffisante (de l'ordre de plusieurs milliers de caractères).

La décompression est très simple et rapide. A partir de la suite de triplets, le décodage s'effectue en faisant coulisser la fenêtre comme pour le codage. Le dictionnaire est donc reconstruit de gauche à droite en une seule fois.

L'algorithme comprime les informations au fur et à mesure du déplacement de la fenêtre. Contrairement à Huffman, il n'a besoin que d'un seul passage sur la chaîne à coder. Il peut donc être utilisé sur des données en transit. C'est pourquoi la norme V42bis des modems utilise la compression LZ77 pour réduire la quantité de données à télécharger. Il faut bien sûr que la mémoire du modem soit suffisante pour contenir la fenêtre de l'algorithme.
On peut ainsi dépasser la valeur maximale indiquée par le constructeur (33,6 Kbps, 56 Kbps…).

- **LZ78**

Il est tout simplement une amélioration du LZ77 par les mêmes auteurs.

La fenêtre coulissante a disparu. Le dictionnaire est toujours présent, mais séparé du pointeur. Le pointeur lit le fichier de gauche à droite.
Au point de départ, le dictionnaire ne contient aucun facteur. Les facteurs sont numérotés à partir de 1. A tout moment, l'algorithme recherche le plus long facteur du dictionnaire qui concorde avec la suite des caractères du texte. Il suffit alors d'encoder le numéro de ce facteur et le caractère suivant du texte : (***num***, ***c***). Le caractère ***c*** concaténé au facteur numéro ***num*** nous donne un nouveau facteur qui est ajouté au dictionnaire pour être utilisé comme référence dans la suite du texte.

Exemple : Codage de la chaîne **aaabbabaabaaabab**

Parcours du texte:	a	aa	b	ba	baa	baaa	bab
Numéro de facteur:	1	2	3	4	5	6	7
Codage:	(0,a)	(1,a)	(0,b)	(3,a)	(4,a)	(5,a)	(4,b)

Au point de départ, le dictionnaire ne contenant aucun facteur, aucune concordance ne peut être trouvée entre les caractères du début du texte et un facteur. On code donc (**0**,**a**) ce qui signifie le facteur vide suivi du caractère **a**. Ce nouveau facteur portera le numéro **1** dans le dictionnaire.

Ensuite, **aa** correspond au premir facteur du dictionnaire suivi de **a**. on code donc (**1**,**a**). Ce nouveau facteur est numéroté **2**.

Les caractères suivants (**bba** ...) sont inconnus dans le dictionnaire. On codera donc que l'on a le facteur vide suivi du caractère **b** : (**0**, **b**). Ce nouveau facteur porte le numéro **3**. Suivent alors les caractères **ba** considérés comme le facteur **3** suivie du caractère **a**, et ainsi de suite...

Comme pour LZ77, LZ78 comprime les informations au fur et à mesure de leur réception.

Le décodage est très simple également parce qu'il suffit au décodeur de reconstruire le dictionnaire au fur et à mesure du décodage. Les numéros des facteurs seront donc les mêmes que pour le codage et les facteurs pourront être interprétés sans problème.

- **LZW**

L'avantage de LZ78 par rapport à LZ77 était de passer de 3 informations à coder à 2. Terry Welch, en 1984, publie son travail, complément de l'algorithme LZ. Il est parvenu à n'avoir à coder plus qu'une seule information. Pour évincer la seconde information, Welch à eu l'idée d'un dictionnaire dynamique, qui se construit au fur et à mesure que l'on scanne le fichier à compresser. On ne transmet pas le dictionnaire ainsi formé, mais on le reconstruit de la même façon à la décompression.

C'est cette compression LZW qui est utilisée dans les images GIF.

Nous avons mis en œuvre cette méthode, c'est pourquoi nous en parlerons plus loin dans la partie **2.2**

2. LA COMPRESSION GRAPHIQUE CONSERVATIVE

2.1. MISE EN OEUVRE DE LA RLE ET DEMARCHE

Nous avons vu que la compression RLE était utilisée quasiment dans tous les algorithmes et tous les formats. Nous avons donc décidé de réaliser un programme avec Borland Delphi qui permettrait de voir en détail son fonctionnement informatique.

2.1.1. AU COMMENCEMENT

Nous avons décidé, pour réaliser une compression RLE d'une image, de ne pas utiliser les fonctions déjà définies *LoadFromFile()* et *SaveToFile()*. Dans une première phase, l'image est donc lue, compressée, puis enregistrée au format RTPE. Dans un second temps, l'image RTPE est décodée, affichée, puis enregistrable au format Bitmap Windows 24 bits.

Le nombre de couleurs nous limite dans notre exploitation des fichiers Bitmap. Les autres niveaux de couleurs utilisés (8 bits, 32 bits) ne peuvent pas être lus, car nous ne savons pas lire et enregistrer la palette présente au début du fichier, ni l'octet de la couche alpha, exprimant la transparence du pixel.

La taille (en octets comme en pixels), qui était un sérieux frein au début de l'exploitation, ne posent maintenant plus de problèmes. Nos programmes sont capables de lire tous les fichiers, et de tous les enregistrer (dans la limite de 2 octets pour la hauteur et la largeur (soit 65536 x 65536 pixels) et de 4 octets pour la taille (soit 4,29 Go)).

Pour faire une première approche avec la compression RLE, nous avons décidé d'utiliser l'algorithme avec du texte. Nous avons donc construit une application simple qui lisait le nombre de caractères identiques consécutifs, puis enregistrait le nombre d'itérations, suivi du caractère en question. Les résultats était médiocres : les taux avoisinaient 200%.

Ce premier contact avec la compression RLE nous fit vérifier que cette compression, certes conservative, est inefficace pour traiter les fichiers non graphiques.

2.1.2. LE TRAITEMENT DE L'IMAGE

Pour commencer à construire un programme traitant la compression RLE, nous avions différentes sources. Le traitement du Bitmap étant connu, le traitement de la compression restait plus délicat. D'après certains sites, le codage conventionnel se fait comme expliqué plus haut dans ce mémoire. Pour les autres (la majorité) qui avaient déjà essayé de traiter le sujet, il nous était proposé de compresser dès que la redondance atteignait 3 ou 4 pixels, et de ne pas compresser les pixels non redondants, en ajoutant un caractère spécial lorsque qu'on appliquait une compression.
Le problème du caractère spécial est son choix. Sur l'intervalle [0 .. 255], toutes les valeurs peuvent être prises par les couleurs du pixel. Les pixels, avec la compression RLE, s'enregistrent en 3 parties : un octet pour le rouge, un pour le vert, et un pour le bleu.

Nous avons donc décidé de mélanger les informations trouvées, en ajoutant ou en retirant des éléments. Finalement, après plusieurs essais, la compression qui en résulte est très simplifiée :

- Si des pixels consécutifs ne se répètent pas, on écrit 00, suivi du nombre de pixels, suivi de ces pixels. On n'ajoute pas 00 si la chaîne est impaire, comme le recommandait les pages Web.
- Si ils se répètent, on écrit le nombre de répétitions, puis le pixel à répéter.
- Les codes 00 00 et 00 01 pour la fin de ligne et fin de fichier (codage conventionnel) sont supprimés. Puisque l'on connaît la hauteur et la largeur du fichier, l'algorithme de décompression change de ligne automatiquement.
- Nous avons insérés les codes 01 et 02, qui étaient inutilisés (car le 00 signale les octets non redondants, et on ne code les autres qu'à partir de 3 répétitions (donc de 03 à 255)). Le 01 indique qu'il faut ajouter 256 à l'octet suivant pour répéter le pixel. On l'utilise si les pixels redondants sont compris entre 255 et 511. On ne rencontre déjà pas souvent plus de 500 pixels identiques consécutifs. Et pour le cas très rare ou plus de 511 pixels seraient alignés, on écrit 02, qui signale que le nombre de redondances est codé sur les 2 octets suivants. On trouve ensuite sur 3 octets la couleur à reproduire.

Il reste ensuite à définir comment va-t-on lire le fichier. Le plus naturellement possible on pense de gauche à droite, puis de haut en bas. Mais la compression RLE effectue plusieurs combinaisons, et garde la meilleure. Sur Internet on nous parle de zigzags. Les résultats par la première méthode sont satisfaisants, et donc on imagine des suivants. On pourrait également compresser verticalement, ou en boucles …

Le décodage de l'image se fait très simplement, car la RLE est une compression symétrique. On lit le premier octet de l'image, et on exécute l'action désignée (reproduire les octets suivants, simplement les copier, lire 2 octets …)

2.1.4. NOTRE FORMAT

Après la compression, il fallait enregistrer les nouvelles données dans un fichier. Pour lui donner plus de personnalité, nous avons choisi d'appeler les nouveaux formats créés. Celui-ci est le RTPE. Il est constitué comme la plupart des autres d'un entête et de l'image compressée :

- Signature : 'RTPE' qui permet de reconnaître le fichier (4 octets)
- Taille totale du fichier (4 octets)
- Largeur de l'image (2 octets)
- Hauteur de l'image (2 octets)
- Type de compression (1 octet)

2.1.5. RESULTATS

La compression RLE est rapide et efficace, mais seulement pour des fichiers qui possèdent des caractères redondants, ou des images avec des pixels identiques consécutifs.
Les tests avec cette méthode de compression sont très satisfaisants avec des images assez homogènes. Des taux de 1% ou 2% ne sont pas rares, des fichiers de moins de 1024 octets non plus. Il est des fois ou la taille du fichier final RTPE est meilleure que celle du fichier compressé avec des outils graphiques perfectionnés. Le désavantage réside surtout dans le fait que le fichier compressé n'est lisible que par le décompresseur inclus dans le programme, et non par celui des logiciels graphiques populaires.
Mais dès que l'on approche les images compliquées, colorées, le taux oscille entre 70% et 90%. C'est moins performant que les autres algorithmes.
Il arrive parfois que le fichier compressé soit supérieur au fichier original. Dans ce cas le taux est entre 100% et 101%

Cette compression RLE, bien que très simple, s'avère donc particulièrement efficace dans certains cas. Sa meilleure utilité est lorsqu'elle est couplée avec un autre algorithme de compression, comme dans la plupart des formats à compression.

2.2. Mise en oeuvre de la LZW

2.2.1. Fonctionnement du dictionnaire

La caractéristique principale de la compression LZW est son dictionnaire. Ce fut la première à en posséder un, et sans tenir comptes des variantes, la seule. Examinons l'exemple suivant :

Ici, la chaîne à coder est : **"XAVIER_MARMIER_PONTARLIER_"**

[1]	[2]	[3]	[4]	[5]	[6]	[7]
Étape (nb_bits)	**Lu**	**Émis (décimal)**	**Émis (binaire)**	**Tampon**	**Adresse**	**Séquence**
1 (8)	X			X	/	/
2 (8)	A	88	0101 1000	XA	260	XA
3 (8)	V	65	0100 0001	AV	261	AV
4 (8)	I	86	0101 0110	VI	262	VI
5 (8)	E	73	0100 1001	IE	263	IE
6 (8)	R	69	0100 0101	ER	264	ER
7 (8)	espace	82	0101 0010	R espace	265	R espace
8 (8)	M	32	0010 0000	espace M	266	espace M
9 (8)	A	77	0100 1101	MA	267	MA
10 (8)	R	65	0100 0001	AR	268	AR
11 (8)	M	82	0101 0010	RM	269	RM
12 (8)	I	77	0100 1101	MI	270	MI
13 (9)	E	256	1 0000 0000	IE	/	/
14 (9)	R	263	1 0000 0111	IER	271	IER
15 (9)	espace	/	/	R espace	/	/
16 (9)	P	265	1 0000 1001	R espace P	272	R espace P
17 (9)	O	80	0 0101 0000	PO	273	PO
18 (9)	N	79	0 0100 1111	ON	274	ON
19 (9)	T	78	0 0100 1110	NT	275	NT
20 (9)	A	84	0 0101 0100	TA	276	TA
21 (9)	R	/	/	AR	/	/
22 (9)	L	268	1 0000 1100	ARL	277	ARL
23 (9)	I	76	0 0100 1100	LI	278	LI
24 (9)	E	/	/	IE	/	/
25 (9)	R	/	/	IER	/	/
26 (9)	espace	271	1 0000 1111	IER espace	279	IER espace
27 (9)		32	0 0010 0000		/	/

Le résultat est 01011000 01000001 01010110 01001001 01000101 01010010 00100000 01001101 01000001 01010010 01001101 10000000 01000001 11100001 00100101 00000010 01111001 00111000 10101001 00001100 00100110 01000011 1100 0100 000 (binaire)

- Au préalable :
  - Charger le premier caractère dans le tampon *(étape 1).*
  - Mettre (nb_bits) à 8.
  - Mettre (Adresse) à 260.

- Principe général :
  - On passe à l'étape suivante.
  - On ajoute à Tampon le caractère de l'étape.
  - Si Tampon est dans le dictionnaire, on ne fait rien et on recommence. Sinon, on ajoute Tampon dans le dictionnaire en lui attribuant un code.
  - On émet Tampon sur (nb_bits) bits, sans le dernier caractère.
  - On retire de Tampon les caractères émis.

- Exemple sans répétition (étape 1) :
  - On passe à l'étape 2.
  - On ajoute à Tampon le caractère *(étape 2)*. (***"X" devient alors "XA"***).
  - On regarde si Tampon est présent dans le dictionnaire *(Colonne [7])*. S'il n'est pas, on l'y ajoute et on lui attribue un code ***(260 : "XA")***.
  - On émet *(c.a.d. on écrit dans le fichier de sortie)* Tampon sans le dernier caractère (***émission de "X"***) sur 8 bits (colonne [4]), c'est à dire son code. Ici, le code ASCII (***"X" = 88***).
  - On ôte le ou les caractère(s) émis de Tampon. (***Tampon devient "A"***).

- Exemple avec répétition (étape 20) :
  - On passe à l'étape 21.
  - On ajoute à Tampon le caractère (***"A" devient "AR"***).
  - On regarde si Tampon est présent dans le dictionnaire. S'il est présent, on émet rien et on continue.

- Suite de l'exemple avec répétition (étape 22) :
  - On ajoute au Tampon le caractère suivant (***"AR" devient "ARL"***).
  - On regarde si Tampon est dans le dictionnaire. Comme il n'y est pas, on l'ajoute et on lui attribue un code *(**277 : "ARL"**).*
  - On émet Tampon sans le dernier caractère (***"AR"***) sur 9 bits. Ce n'est pas un caractère. Il n'y a pas de code ASCII, mais il a obligatoirement son code dans le dictionnaire (***"AR" = 268***).
  - On ôte les caractères émis. (***Tampon devient "L"***)

- Notes :
  - nb_bits est égal à 8 au départ, car le premier caractère lu à son code ASCII compris entre 0 et 255. Ce code nécessite au moins 8 bits.
  - Adresse est égal à 260 au départ, car de 0 à 255 on confondrait un code du dictionnaire avec un code ASCII, et on réserve les codes 256, 257, 258, 259 pour un usage spécial.
  - Lors de la rencontre avec la première répétition, au lieu de ne rien faire, on incrémente (nb_bits) de 1, puis on émet le code 256, synonyme d'augmentation de (nb_bits). Cela n'est utile que pour le décompacteur. En règle générale, on émet ce 256 lorsque l'on remarque que le code de Tampon est supérieur à $(2^{nb\_bits} - 1)$**.** Ainsi, après une recherche dans le dictionnaire, si l'on s'aperçoit que Tampon à pour code 512, au lieu de ne rien émettre et de continuer, on émet 256, puis on incrémente (nb_bits) de 1.
  - Si (Adresse) = (4096 + 260), donc si on vient de rentrer une séquence dans la dernière ligne du dictionnaire, on émet le code spécial 257, on vide le dictionnaire, et on initialise (nb_bits) à 8 ainsi que (Adresse) à 260.
  - Le nombre d'octets au départ était de 26. Le fichier de sortie contient 187 bits, soit 24 octets (donc un gain de 7%). Et plus le nombre d'étapes devient grand, plus le gain devient grand.

On remarque que la compression, pour être performante, nécessite un dictionnaire bien rempli. Grâce à l longueur variable de (nb_bits), on gaspille moins d'espace. Mais plus le dictionnaire est grand, plus la recherche est longue.

Examinons à présent la décompression :

Comme à la compression, on initialise (nb_bits) à 8, (Adresse) à 260 et (Pos) à 8. On lit les 8 premiers bits et on émet le caractère correspondant à la valeur lue.

Principe :
- On lit (nb_bits) bits à partir de (Pos). Si la valeur décimale est inférieure à 256, on émet le caractère qui correspond au code ASCII de la valeur. Si elle est supérieure à 259, on émet la séquence du dictionnaire qui correspond à la valeur lue.
- Si cette valeur est un code spécial, on exécute l'action correspondante et on recommence.
- On ajoute dans le dictionnaire la séquence lue concaténée avec le caractère suivant.

La décompression est quasi-instantanée. Elle n'a qu'a lire des valeurs et à suivre les instructions. Elle n'effectue aucun travail de recherche.

2.2.2. CARACTERISTIQUES DE NOTRE FORMAT WTPE ET DEMARCHE

➢ De façon à bien cerner le mécanisme de l'algorithme, nous avons commencé par l'appliquer sur une petite chaîne :

Par exemple, "AVOGADRO" ne contient aucune répétition. Le dictionnaire contient {AV, VO, OG, GA, AD, DR, RO}. La taille en sortie sera identique.
Mais "EINSTEIN" contient 2 fois "EIN". Lorsque l'on applique les méthodes, on obtient :

"EINST" + Code(256) + Code(260) + "N", ce qui fait (5 * 8) + 9 + 9 + 9 = 67 bits. La taille passe de 64 à 67 bits, mais un code supplémentaire a été inséré (256). C'est pourquoi les 5 premiers caractères sont codés sur 8 bits, et le dernier "N" sur 9 bits.
On remarque que l'algorithme LZW n'a pas remarqué la suite "EIN". Il est obligé de procéder octet par octet, sinon le dynamisme ne serait plus supportable. Des autres algorithmes variants parviennent à repérer les longues suites du premier coup. Mais en général, ils sont moins efficaces.

Après de nombreux essais, on remarque que le saut 8 bits - 9 bits se réalise assez rapidement. C'est pourquoi le format WTPE commence directement à 9 bits. S'il devait coder "EINSTEIN", cela donnerait :
(5 * 9) + 9 + 9 = 63 bits. La plupart des fichiers possèdent un en-tête. Celui-ci est généralement beaucoup plus grand que l'utilisation qu'on en fait. Par exemple, dans un fichier bitmap, les 54 premiers octets de l'en-tête sont pour beaucoup des 00. C'est pourquoi on se permet de démarrer directement sur 9 bits.

➢ Problème : Dans le stockage des séquences du dictionnaire, l'emploi du type *string* est fortement prohibé. En effet, au commencement, nous avons utilisé un tableau de *string*. Les fichiers graphiques comportent beaucoup le caractère "00" (Dans le 8 bit, 00 = Absence de couleur = blanc, et dans le 24 bits, 00 = absence d'une composante, rouge, verte ou bleue).
Or, un *string* est défini comme un tableau de *byte*, qui se termine par un caractère nul. Donc dès qu'une de nos séquences contenait un caractère 00, elle était mal enregistrée dans le dictionnaire. Et donc toute la suite du fichier compressé était erronée.

➢ Autre problème, que nous avons appelé "bug du 000". Les premières fois, la compression n'était pas conservative, à cause de lui. Pour voir en quoi il consiste, essayons de compresser la chaîne "00 00 00" (peu importe le contenu, du moment que les 3 octets sont identiques).

Pour une meilleure compréhension, nous allons compresser "$0_1 0_2 0_3$", les chiffres en indice servant juste à nous repérer dans l'algorithme :

- On charge 0_1 dans le Tampon
- On y ajoute 0_2. La séquence $0_1 0_2$ n'est pas présente dans le dictionnaire. On l'y ajoute, en lui attribuant le code 260. On émet ensuite "0"
- Le Tampon de l'étape suivante contient $0_2 0_3$. Or, "00" est déjà dans le dictionnaire. On émet donc le code de la séquence, c'est à dire 260.
- Le résultat est donc {00, 260}.

Voyons maintenant le travail du décompresseur :

- Lecture de {0}. Émission de "0".
- Lecture de {256}. Le décompresseur va donc rechercher cette chaîne dans le dictionnaire. Mais elle n'a pas encore été crée. Le plantage survient alors. Donc dans l'algorithme de compression, il faut inclure un test supplémentaire, qui vérifie que l'on a pas d'octets qui fait référence à une séquence qui se code avec lui-même.

➢ Nous nous sommes ensuite penchés sur la LZW dans le bitmap. Pour les autres formats, les fonctions de compression reçoivent en argument un tableau de TColor et un nom de fichier.

Cela n'a pas été possible pour la LZW. Le dictionnaire, dont l'exemple plus haut explique le fonctionnement avec une chaîne simple, recueille les toutes les séquences de pixels, dans une image. Prenons l'exemple d'une image 24 bits. Chaque pixel est décomposé en 3 composantes Rouge, Vert et Bleu. Le dictionnaire doit donc être un tableau de *TBitmap*, soit un tableau de tableaux de *TColor*, soit encore un tableau de tableaux de tableaux de *byte*. Le problème est ici : Chaque nouvelle séquence lue doit être comparée à toutes celles présentes dans le dictionnaire. Il faut donc rassembler tous les *bytes* du dictionnaire, ce qui représente un énorme effort du processeur. Le temps pris est énorme. On

comprend ainsi pourquoi la compression GIF, qui n'utilise que la LZW, ne fonctionne que pour une image de 256 couleurs. Car 256 couleurs équivaut à 1 pixel = 1 octet, ce qui facilite grandement la tâche.

A ce point, nous nous sommes dit que la compression LZW des images GIF était très similaire à la compression LZW de WinZip. De plus, nous ne pouvons pas lire les palettes des images Bitmap, ce qui ne nous permet pas de visualiser un fichier. Donc nous attaquons directement le fichier, du premier octet au dernier.

➢ Après de longues heures de déboguage, le programme donne enfin signe de conservatisme. Cependant, le temps de compression est énorme. Mais nous avons pu le diviser par 10 entre le premier programme et le programme actuel :

Nous savons que le point faible de la LZW est la recherche de séquences identiques dans le dictionnaire. Plus ce dernier est gros, plus le temps dépensé est grand. Mais plus il est gros, meilleure est le gain de compression. Il y a donc un compromis entre temps et gain. La taille du dictionnaire oscille entre 4096 ou 8192 emplacements.
Mais la taille des séquences est connue. Donc nous avons décidé de rallonger les séquences dans le dictionnaire de 1, ce qui offre une place libre pour un octet. Cet octet libre contient la taille de la chaîne, qui est inscrit en même temps que cette dernière. Par exemple, si un dictionnaire contient 2 chaînes, "AB" et "ABC", il donne : { {2, 65, 66}, {3, 65, 66, 67} }.
Ainsi, lors d'une comparaison d'une chaîne inconnue à 3 caractères, disons XXX, on compare d'abord la taille de XXX avec celle de la séquence du dictionnaire : 3 est différent de 2, donc on ne regarde pas si XXX = "AB", ce qui serait inutile. Lorsqu'on arrive à la seconde séquence, le résultat est vrai. Donc on compare si X_1 = "A" et X_2 = "B" et X_3 = "C".

Le programme utilise beaucoup de tableaux, statiques et dynamiques. Les derniers ont besoin d'êtres souvent redimensionnés, car le Delphi ne permet pas l'inscription d'une donnée dans une case du tableau inexistante, et déclenche une erreur. On utilise la fonction *SetLength()*. Or, cette fonction permet de gérer les blocs mémoire alloués par Windows. Elle utilise ainsi un temps considérable (quelques millisecondes de trop …). Le fait de placer des tests qui font sauter l'exécution à une autre portion de code juste avant des fonctions telles que *SetLength()* permet aussi de réduire le temps de compression.

En général, dans une boucle répétée très souvent, il faut essayer de placer le maximum de tests avant, pour éliminer les cas dont on peut prédire à l'avance le résultat.

Un peu plus tard, sur Internet, nous avons trouvé une page parlant de l'amélioration de l'algorithme LZW. Il était conseillé de faire un second tableau contenant les tailles des séquences, qui permettrait la division du temps par 12. Et nous avions déjà eu cette idée. Malheureusement, elle ne nous a divisé le temps "que" par 5. Un autre conseil donné par le site était de programmer en assembleur. Il est bien connu que le code ASM n'a pas son pareil en vitesse d'exécution. Mais de là à passer de Delphi à Notepad …

➢ Le format WTPE utilise uniquement l'algorithme LZW, modifié comme indiqué ci-dessus. L'image crée est similaire au GIF. Bien que le temps de compression soit beaucoup plus élevé que celui de PSP ou Photo Editor, le gain de compression est parfois plus élevé.
Étant donné que la méthode LZW était en dehors de nos objectifs initiaux, les compresseurs et décompresseurs LZW ne sont pas tout a fait achevés. Toutefois, nous travaillons encore activement sur le sujet, et les dernières versions de nos programmes sont disponibles à http://crteknologies.multimania.com

➢ Son en-tête :

- Signature : 'WTPE'qui permet de reconnaître le fichier (4 octets)
- Taille totale du fichier (4 octets)
- Largeur de l'image (4 octets)
- Hauteur de l'image (4 octets)

2.3. Le codage predictif et sa mise en oeuvre

2.3.1. Principe du codage predictif

Comme son nom l'indique, le codage prédictif effectue une prédiction. Étant donné que des blocs d'une image en couleurs vraies (24 bits) ont tendance à varier d'une manière progressive, c'est à dire que deux blocs voisins ont de sérieuses chances d'être similaires, on peut essayer, à partir de quelques pixels, d'imaginer ce que pourraient être les pixels voisins. Par exemple, sur une image 24 bits, un pixel (j, i) dont la valeur Rouge est 200, et un pixel (j, i+2) dont la valeur Rouge est aussi 200, on peut supposer que le pixel (j, i+1) a lui aussi une valeur Rouge égale à 200, ou du moins proche.
Le but est alors de stocker la différence entre la couleur du pixel et sa prédiction, qui sera généralement faible. Cette différence est donc une valeur non plus absolue, mais relative.

- **Prediction**

Considérons une matrice de pixels :

(1)	(2)	.	.	.	.	.	(8)
(9)	.	A	B	C	.	.	.
.	.	D	X				

Nous codons de haut en bas et de gauche à droite. Nous avons donc les valeurs réelles de A, B, C et D. Plusieurs choix s'offrent alors à nous pour réaliser la prédiction de la valeur de X, en prenant en compte un nombre plus ou moins grand de pixels environnants et en leur donnant plus ou moins de poids selon leur proximité. Le principe général est donc de réaliser une moyenne pondérée. Par exemple :

> (A + B + C + D) / 2
> (2B + 2D + A + C) / 6
> (B + D) / 2

- **Codage complementaire**

Si on examine bien le principe de coder la différence entre la valeur prédite et la valeur réelle, on se rend compte qu'elle ne peut pas être utilisée seule. En effet, en considérant par exemple que chaque pixel (ou composante) est codé sur un octet (256 niveaux), si on prédit qu'une valeur sera égale à 0, et qu'en réalité elle est égale à 255, il faudra enregistrer 255, et si on prédit qu'elle sera égale à 255 et qu'en fait elle est nulle, on devra enregistrer -255. Cela fait donc une amplitude de 511 niveaux, qu'il faudrait enregistrer au minimum avec 9 bits puisqu'il faut prévoir toutes les possibilités, ce qui augmenterait la taille du fichier.

Il faut donc ensuite coder les données avec un codage statistique de type Huffman qui sera efficace puisque les valeurs à coder sont en général faibles, et donc plus fréquentes ce qui leur vaudra un code plus court. On peut également traiter les valeurs nulles avec une RLE (avant Huffman), puisque pour des étendues de même couleur la différence entre la valeur prédite et la valeur réelle sera nulle.

2.3.2. Caracteristiques de notre format PTPE et demarche

Nous avons donc créé un format, que nous avons nommé PTPE et qui utilise cette méthode. Nous nous sommes inspirés de l'en-tête des fichiers .BMP pour réaliser l'entête (entête classique d'ailleurs) :

- Signature : 'PTPE' qui permet de reconnaître le fichier (4 octets)
- Taille totale du fichier (4 octets)
- Largeur de l'image (4 octets)
- Hauteur de l'image (4 octets)

Après l'entête on trouve naturellement les données correspondant à l'image.

Pour effectuer la prédiction, nous avons, après plusieurs essais, privilégié la formule (B + D) / 2 pour un pixel au centre de la matrice. Si X est sur la première colonne, on prend seulement B en compte. Si X est sur la première ligne, on prend seulement D en compte. Cette solution donne le meilleur taux de compression, et reste relativement simple.

Nous avons choisi d'utiliser la méthode adaptative de l'algorithme de Huffman (vous pouvez vous référer au code source à la fin de ce dossier et dans le cd-rom inclus avec le dossier), en réactualisant l'arbre tous les 20 pixels, et nous avons commencé par appliquer le codage de Huffman en construisant un arbre de 511 éléments afin de pouvoir coder toutes les valeurs. Cette méthode était efficace mais se révélait très lente. En effet l'arbre étant très gros et rafraîchis fréquemment prenait beaucoup de temps. De plus les valeurs assez grandes pour justifier la taille de cet arbre étant rares, il pouvait être intéressant de réduire la taille de l'arbre et de rajouter un caractère spécial qui informe que les 9 bits suivants contiennent la valeur, quand celle-ci est supérieure à la taille de l'arbre. Après plusieurs essais nous avons choisi une taille de 255 caractères pour l'arbre qui offre le meilleur compromis entre vitesse d'exécution et efficacité.

Nous avons ensuite inclus une RLE avant Huffman, en ajoutant deux caractères spéciaux à l'arbre pour informer que les 6 ou 12 bits qui suivent (suivant le caractère spécial) contiennent le nombre de valeurs nulles consécutives. Ces valeurs 6 et 12 ont été également choisies après plusieurs essais sur l'efficacité, puisque plus elles sont grandes moins on est obligé de répéter l'opération en cas de très grandes étendues de même couleur, mais plus on perd de place à chaque utilisation où un nombre de bits inférieur serait suffisant. C'est d'ailleurs pour la même raison que nous avons choisi de mettre deux caractères spéciaux RLE avec des tailles différentes.

2.3.3. RESULTATS

Nous avons ensuite, comme à chaque essai, comparé la taille des fichiers PTPE avec la taille des fichiers compressés aux formats TIFF et PNG (les plus utilisés pour les images en couleur vraie sans pertes, utilisant la méthode LZW) produits par des logiciels spécialisés (PSP 7.0, Photo Editor).
A notre grande surprise, le taux de compression de notre format pour des images photographiques dépassait nettement ces autres formats, et même le compresseur WinZip. Nous avons cependant découvert dans le compresseur WinRar une option intitulée *« Utiliser la compression multimédia »* qui d'après l'aide est efficace sur les images en 24 bits, et qui donne des résultats très similaires à notre format. Nous en avons donc déduit qu'il utilisait cette même méthode.

Cette compression prédictive se révèle donc être la meilleure méthode de compression conservative des images en couleur vraie, ce qui montre que les formats courants ne font que « supporter » les images 24 bits et ne sont pas optimisés pour. On pourrait s'étonner de cela, mais même avec le codage prédictif notre format PTPE ne parvient pas à diviser la taille du fichier par deux, alors que la méthode JPEG, un algorithme destructif, permet de diviser par 20 à 40 la taille d'une image, tout en gardant une bonne qualité de l'image.

Nous avons donc étudié cette méthode que nous allons maintenant présenter.

3. LA COMPRESSION NON CONSERVATIVE : LA NORME JPEG

3.1. THEORIE ET PRINCIPES DE LA NORME JPEG

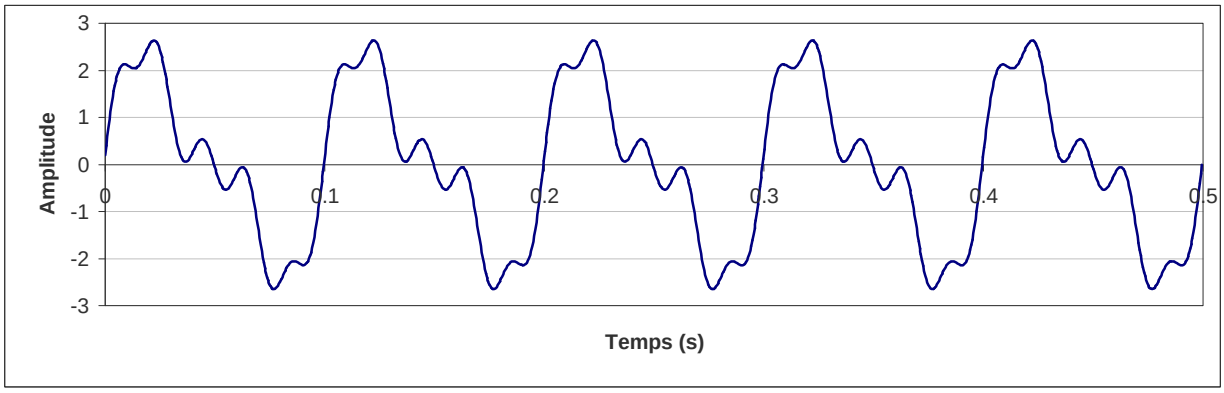
L'association de deux groupes de normalisation, le CCITT (*Commission Consultative Internationale de la Télégraphie et de la Téléphonie)* et l'ISO (*International Standards Organisation*) a donné naissance au JPEG (*Joint Photographic Experts Group*) qui a défini la norme JPEG, aboutissement de la recherche d'une méthode pour la compression des données graphiques. Cette norme comprend des spécifications pour le codage conservatif et non conservatif, mais nous allons seulement nous intéresser à la partie non conservative, la plus intéressante.
Le JPEG est juste une méthode de compression mais n'est pas un format : le format .jpg le plus utilisé est en fait le format JFIF (*Jpeg Interchange Format*), mais la méthode JPEG est aussi utilisée dans d'autres formats comme le TIFF ou le TGA, et même pour stocker les images dans les appareils photographiques numériques.
C'est une méthode destructive (elle perd une partie des données) qui est adaptée aux images naturelles pour lesquelles elle donne de très forts taux de compression pour une perte de qualité indécelable, car elle exploite les défaillances de notre système de vision, mais en revanche elle convient mal aux images géométriques.

3.1.1. LA TRANSFORMEE EN COSINUS DISCRETE (DCT)

La clé du processus de compression est la DCT (*Discrete Cosine Transform*) bidimensionnelle ou transformée en cosinus discrète, une variante de la transformée de Fourier discrète. Comme cette dernière elle transforme un signal *discret* (signal composé d'éléments distincts, les pixels dans notre cas) bidimensionnel d'amplitude en une information bidimensionnelle de fréquence.

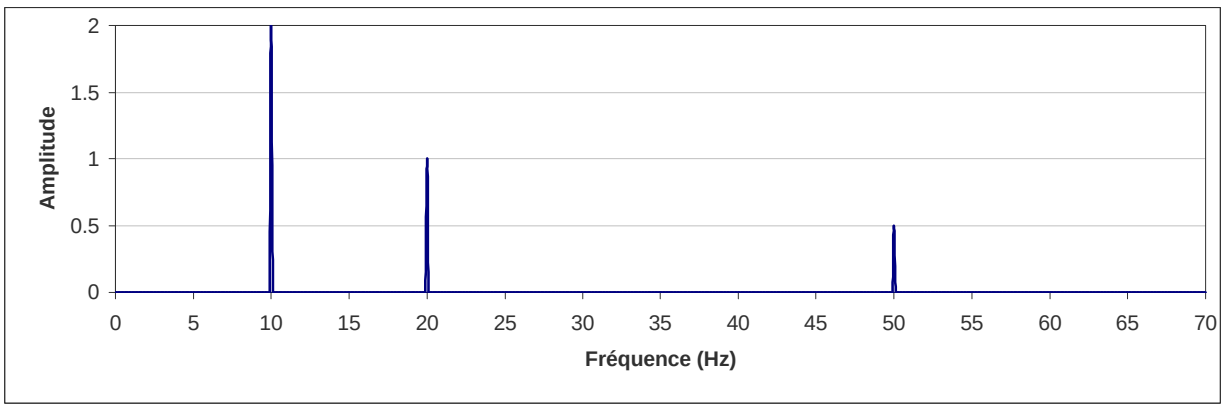
➢ LA TRANSFORMEE DE FOURIER

Etudions la transformée de Fourier unidimensionnelle pour se rendre compte de son fonctionnement. Elle va transformer un signal en une somme de sinusoïdes (fonctions sinus et cosinus) de différentes fréquences, amplitudes et phases. Par exemple imaginons le signal suivant, qui peut être un signal sonore ou électrique :



:

En lui appliquant la transformée de Fourier :



I

Elle nous révèle ainsi que le signal est composé de la superposition de trois sinusoïdes :

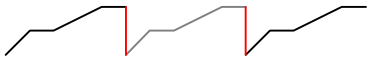
- une première de fréquence 10 Hz et d'amplitude 2
- une seconde de fréquence 20 Hz et d'amplitude 1
- une dernière de fréquence 50 Hz et d'amplitude 0.5

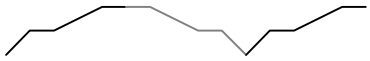
On constate l'utilité que peut avoir une telle représentation, puisqu'il est facile de filtrer le signal (par exemple pour les sons il est facile de supprimer toutes les fréquences inférieures à 20 Hz ou supérieures à 20 KHz qui sont inaudibles pour l'homme), et une transformée inverse permet de restituer le signal d'origine.

➢ LA DCT

La différence entre la DCT et la transformée de Fourier se situe au niveau de la périodisation du signal. En effet pour pouvoir transformer un signal en sinusoïdes, il faut que celui-ci soit périodique. Mais quand il ne l'est pas, il est transformé afin de le devenir.

Avec la transformée de Fourier, il est périodisé en reproduisant l'intervalle défini du signal « bout à bout » alors que la transformée en cosinus le fait en « dépliant » l'intervalle de sorte qu'il n'y ait plus de discontinuité aux bords :

TF : 

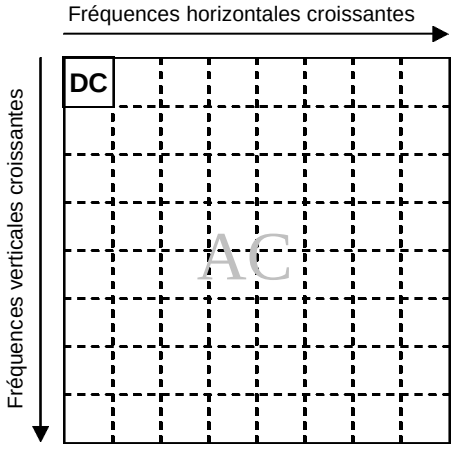
DCT : 

Ainsi la DCT est plus adaptée aux images, car en principe le signal n'est pas du tout périodique, et les discontinuités causées par la FFT provoquent l'apparition de fréquences élevées parasites, qui n'apparaissent pas avec la FFT.

La fonction représentant le signal est alors paire, et sa série de Fourier ne comporte plus que des termes en cosinus, d'où le nom de transformée en cosinus. De plus une transformée de Fourier donne des coefficients complexes (partie réelle et imaginaire), qui donnent ainsi une information sur la phase des sinusoïdes (le décalage dans le temps). Mais comme la DCT ne comporte plus de termes en sinus, elle donne des coefficients réels. Ceci est avantageux car les coefficients réels sont plus faciles à utiliser, et la phase n'est pas utile dans le cas des images.

Pour la DCT bidimensionnelle, le principe est exactement le même, mais les fréquences sont représentées sur deux axes.
L'élément (0,0) est appelé composante DC (pour *Direct Component*) ou coefficient continu. Il représente la valeur moyenne des éléments avant transformation à un coefficient près et est le plus grand coefficient de la matrice. Les autres éléments sont les composantes AC (*Alternative Component*) qui représentent l'amplitude des fréquences spatiales (horizontales et verticales), et plus on s'éloigne de la composante DC plus cela concerne des fréquences élevées.

On peut se représenter la DCT d'une matrice de 8 x 8 éléments ainsi :



La transformation inverse existe, que l'on appellera IDCT, et permet de restituer exactement les données de départ (en l'absence d'erreurs d'arrondis).

Les formules de la DCT bidimensionnelle et de son inverse sont les suivantes :

$$DCT(i,j) = \frac{1}{\sqrt{2N}} c(i)\,c(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} Img(x,y).\cos\left(\frac{(2x+1)ip}{2N}\right).\cos\left(\frac{(2y+1)jp}{2N}\right)$$

$$IDCT(x,y) = \frac{1}{\sqrt{2N}} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} c(i)\,c(j)\,DCT(i,j).\cos\left(\frac{(2x+1)ip}{2N}\right).\cos\left(\frac{(2y+1)jp}{2N}\right)$$

$$\text{avec } c(a) = \begin{cases} \frac{1}{\sqrt{2}} \text{ si } a = 0 \\ 1 \text{ si } a > 0 \end{cases}$$

L'implémentation de ces formules est assez simple : on remplace les deux sigma par deux boucles imbriquées (`For i := 0 To 7 Do For j := 0 To 7 Do`) et on remplit les matrices (tableaux bidimensionnels). Pour plus de détail voir le code source (annexe de ce dossier).

3.1.2. LA METHODE JPEG

- **ETAPES**

PREPARATION

On va donc commencer par traduire les informations du modèle RGB au modèle Luminance - Chrominance, et étant donné que notre oeil est plus sensible à la luminance qu'à la chrominance on va pouvoir stocker avec moins de précision la chrominance. Par exemple, l'oeil ne peut discerner de différences de chrominance au sein d'un carré de 2 x 2 points, et on va donc sous échantillonner la chrominance (regrouper les pixels par carrés de quatre et ne prendre que la moyenne de leur chrominance).

DCT

On applique ensuite la transformée en cosinus. Mais le calcul ne peut se faire sur l'image entière d'une part parce que cela génèrerait trop de calculs, et d'autre part le signal de l'image doit absolument être représenté par une matrice carrée. On va donc décomposer l'image en blocs de 8 pixels de côté et on appliquera la DCT indépendamment sur chacun de ces blocs. Les plus petits blocs en bordure devront être traités par une autre méthode.

QUANTIFICATION

C'est la seule étape non conservative (sauf les erreurs d'arrondis dans les précédentes étapes). La transformée en cosinus nous donne des informations sur la fréquence ce qui va permettre de filtrer de l'information. En effet, non seulement l'information effective de la plupart des images naturelles est concentrée dans les basses fréquences, mais de plus notre oeil est beaucoup moins sensible aux fréquences élevées (changements de couleur brusques) qu'aux fréquences basses (changements lents). La quantification consiste donc à diminuer la précision des fréquences élevées, en divisant chaque élément DCT par l'élément correspondant dans la table de quantification, et en arrondissant à l'entier le plus proche (voir exemple plus loin). Ainsi beaucoup d'éléments deviendront nuls ou très faibles et occuperont donc moins de place.

LINEARISATION

Pour écrire le bloc dans un fichier il nous faut une suite d'octets et nous avons une matrice carrée. Il faut donc la linéariser. On va pour cela employer une méthode particulière, dite de *zigzag*, de façon à regrouper ensemble les éléments les moins importants (voir exemple plus loin).

CODAGE

Les composantes DC sont généralement de grands nombres mais en revanche elles varient lentement d'un bloc à l'autre, puisque ce sont les moyennes de leur bloc. On va donc coder non pas la composante DC elle-même mais sa différence avec celle du bloc précédent, ce qui donnera de plus petits nombres prenant moins de place. Cette méthode est appelée DPCM (*Differential Pulse Code Modulation*). On va ensuite appliquer un codage de Huffman et une RLE pour les zéros qui sont nombreux.

➢ **EXEMPLE PRATIQUE :**

170	154	138	145	151	137	151	133
157	126	138	158	135	130	135	114
136	153	156	138	146	151	148	118
235	220	210	211	224	208	209	156
237	224	228	246	233	211	188	149
159	143	160	168	149	162	159	117
89	121	128	121	127	130	105	76
134	131	123	140	134	125	134	106

Matrice de pixels

1244	70	-58	42	-16	32	-17	7
47	4	29	2	14	4	-1	6
-218	-33	26	-14	9	-5	-4	-4
32	10	-6	2	9	9	5	-5
155	28	8	16	19	-1	-15	5
-28	-12	3	5	-23	-22	-3	8
-19	-18	20	-2	-9	2	-1	2
-17	7	-19	-10	-9	-20	2	-2

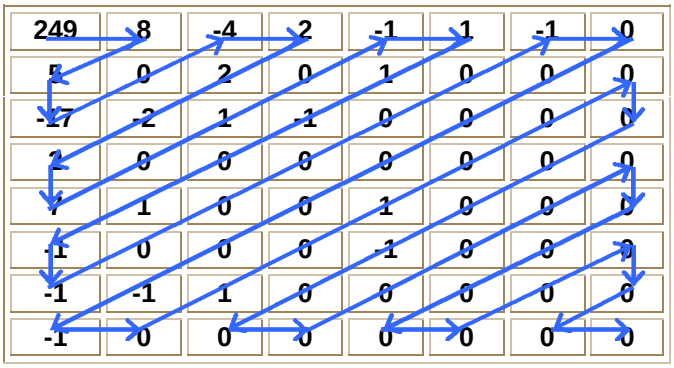
Matrice DCT

5	9	13	17	21	25	29	33
9	13	17	21	25	29	33	37
13	17	21	25	29	33	37	41
17	21	25	29	33	37	41	45
21	25	29	33	37	41	45	49
25	29	33	37	41	45	49	53
29	33	37	41	45	49	53	57
33	37	41	45	49	53	57	61

Matrice de quantification

249	8	-4	2	-1	1	-1	0
5	0	2	0	1	0	0	0
-17	-2	1	-1	0	0	0	0
2	0	0	0	0	0	0	0
7	1	0	0	1	0	0	0
-1	0	0	0	-1	0	0	0
-1	-1	1	0	0	0	0	0
-1	0	0	0	0	0	0	0

Matrice DCT quantifiée



Lecture zigzag

161	151	139	147	138	131	150	144
152	143	139	158	149	131	132	116
147	149	136	134	134	146	155	121
232	231	222	228	223	212	197	153
239	219	212	241	236	202	185	159
151	162	156	155	155	166	168	124
91	134	140	123	115	132	123	55
127	127	128	144	134	120	126	109

Matrice de pixels décodée

On obtient après la lecture zigzag la suite :
‘249,8,5,-17,0,-4,2,2,-2,2,7,0,1,0,-1,1,1,-1,0,1,-1,-1,0,0,0,0,0,-1,0,0,0,0,0,0,0,-1,-1,0,1,0,1,0,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0’
qui comporte de grandes suites de 0 que l’on va coder avec une RLE et de nombreux chiffres très petits qui auront un code très court avec le codage de Huffman puisque plus ils sont petits plus ils reviennent fréquemment.

Pour décompresser on replace cette suite dans la matrice, on multiplie les coefficients par la matrice de quantification et on applique la DCT inverse. On constate que les coefficients décodés sont assez proches de ceux d’origine, alors que la matrice DCT quantifiée ne semblait plus contenir beaucoup d’information.

3.2. MISE EN OEUVRE DU JPEG

Nous avons donc réalisé un programme en Turbo Pascal avec Borland Delphi qui applique la méthode JPEG (joint à ce dossier avec son code source). Deux parties concernent le JPEG, auxquelles vous pouvez accéder au moyen du menu *'Aller à'* :

- La rubrique *'Essais JPEG 1'* permet de vérifier le bon fonctionnement des fonctions de transformées en cosinus, de conversion RGB / Luminance – Chrominance et des tables de quantification, et de se rendre compte de leur fonctionnement séparé.
- La rubrique *'Essais JPEG 2'* permet de vérifier expérimentalement les effets de modifications comme les tables de quantification, le sous échantillonnage de la chrominance, et ainsi confirmer les fondements de la méthode JPEG et mesurer les pertes acceptables.

Enfin la rubrique *'Fichiers'* concerne également le JPEG puisqu'elle permet d'enregistrer et d'ouvrir des images en JPEG et au format BMP. Nous avons pour cela créé notre propre format utilisant la méthode JPEG, que nous avons nommé JTPE et qui est décrit dans la deuxième partie. Il suffit de sélectionner dans la boîte de dialogue *'Ouvrir'* ou *'Enregistrer'* le type *'Fichiers JTPE (*.jtpe)'*.

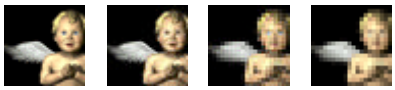
3.2.1. VERIFICATION EXPERIMENTALE DES FONDEMENTS DE LA METHODE JPEG

➢ SENSIBILITE A LA CHROMINANCE

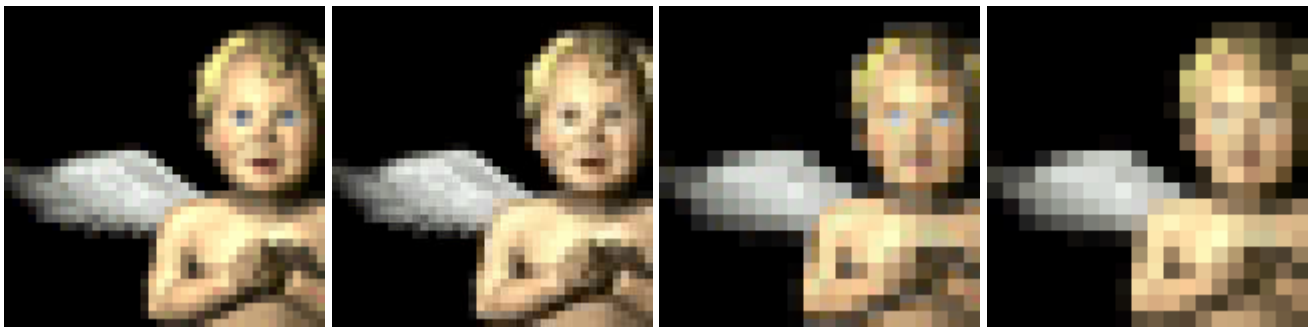
Les données sont converties du modèles RGB au modèle luminance – chrominance car nous sommes incapables de distinguer des variations de la chrominance de deux pixels côte à côte. Pour vérifier cela nous avons donc écrit des procédures pour sous échantillonner à différents niveaux la luminance ou la chrominance.

Pour effectuer vous-même les tests, choisissez 'Essais JPEG 2' dans le menu 'Aller à', puis cliquez sur le bouton 'Ouvrir' pour ouvrir une image .BMP ('Messiah 40x40.bmp' ou autre de votre choix mais de dimensions multiples de 8), puis choisissez l'échelle de sous échantillonnage et cliquez sur 'Lum', 'Chrom' ou 'Lum + Chrom'. Vous pouvez aussi vous référer au code source joint à l'exécutable.

En sous échantillonnant la chrominance deux fois, on ne voit aucune différence en taille réelle avec l'original, mais lorsque l'on zoome, on constate un palissement des couleurs, surtout au niveau des yeux du sujet qui comportaient quelques pixels bleus qui ont été perdus. En revanche la taille de l'image est passée de 40^2 * 3 = 4800 octets à 20^2 * 2 + 40 ^2 = 2400 octets, et a donc été divisée par 2 pour une perte de qualité indécelable sans zoom.



Dans l'ordre : Original, Chrominance / 2, Luminance / 2, Résolution / 2



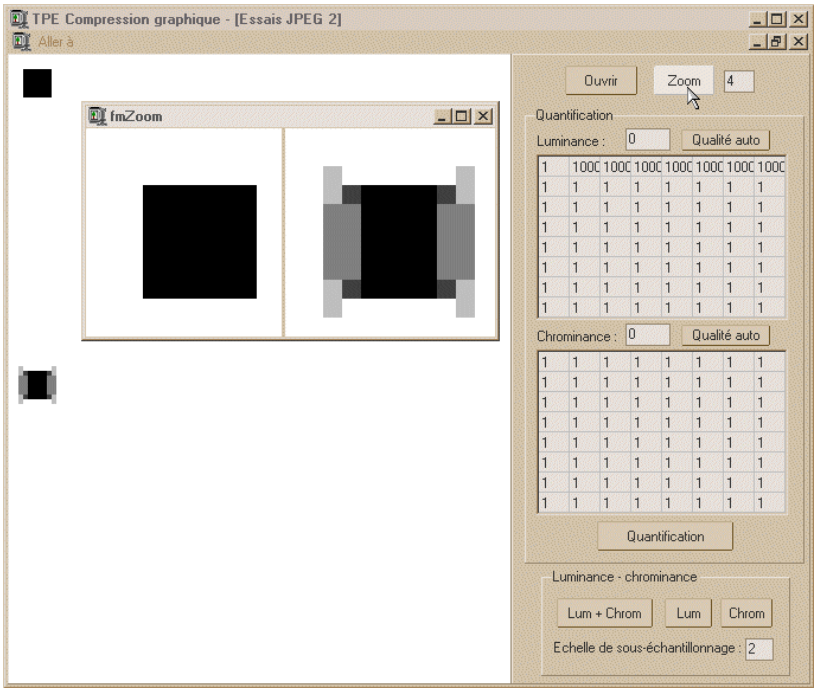
Si l'on fait la même chose avec la luminance, on voit de gros blocs de 4*4 pixels qui sont pratiquement de la même couleur, et on a l'impression que la résolution a été divisée par deux (quand on sous échantillonne à la fois la luminance et la chrominance). De plus la taille de l'image n'est passée que de 4800 à 3600 octets.
Quand on sous échantillonne encore plus la chrominance, les couleurs palissent de plus en plus jusqu'à devenir niveaux de gris. C'est logique puisque seule la chrominance contient les informations sur la couleur.

On vérifie donc que notre oeil ne peut discerner de différences de chrominance entre deux pixels accolés, ce qui permet dores et déjà de réduire de moitié la taille des données sans aucune perte visible, et que d'une façon générale nous sommes beaucoup moins sensible à la chrominance qu'à la luminance, d'où l'utilité de la conversion du modèle RGB au modèle luminance – chrominance.

➢ RESULTAT DE LA DCT

La transformée en cosinus donne des informations sur la fréquence, selon les deux dimensions de l'image. Nous avons donc écrit des procédures pour vérifier le résultat de la DCT.

Pour effectuer vous-même les tests choisissez 'Essais JPEG 2' dans le menu 'Aller à', puis cliquez sur le bouton 'Ouvrir' pour ouvrir le fichier 'Carre 48x48.bmp', remplissez les tables de quantification avec les valeurs de votre choix (ou en cliquant sur 'Qualité auto') puis cliquez sur 'Quantification'. Vous pouvez aussi vous référer au code source joint à l'exécutable.



Quand on met 10000 dans la ligne du haut de la luminance (sauf le coefficient continu, élément (0,0)) (voir copie d'écran précédente), c'est-à-dire que l'on supprime ces composantes, on constate que l'image devient floue, mais uniquement dans le sens horizontal, et quand on met 10000 dans la première colonne l'image devient floue dans le sens vertical, ce qui confirme la définition de la DCT :

Quand on met 10000 dans les 5 cases en haut à gauche (sauf le coefficient continu (0,0)), on constate que les zones qui étaient uniformes sont déformées alors que l'on voit toujours très nettement la rupture qui forme le contour du carré noir, ce qui montre que l'on a supprimé les fréquences basses ce qui confirme aussi la définition de la DCT. On revérifie cela en mettant 10000 dans toutes les autres cases (sauf le coefficient continu (0,0)) et en remettant 1 dans les 5 en haut à gauche. En effet on constate cette fois ce sont bien les hautes fréquences qui ont été supprimées puisque les bords du carré apparaissent flous tandis que les parties uniformes sont parfaitement conservées.

➢ SENSIBILITE AUX HAUTES FREQUENCES

L'exemple précédent permet de vérifier que nous sommes beaucoup plus sensibles aux basses fréquences qu'aux hautes fréquences. En effet dans le premier cas, le carré est méconnaissable alors que moins de 8% des coefficients ont été supprimés, et dans le second cas le carré est relativement bien conservé alors que plus de 90% des coefficients ont été supprimés.

3.2.2. CARACTERISTIQUES DE NOTRE FORMAT JTPE ET DEMARCHE

➢ ENTETE DU FICHIER

L'entête du format JTPE est identique à celle du format PTPE à la différence qu'elle contient un octet de plus qui contient la qualité choisie par l'utilisateur lors de la compression :

- Signature : 'JTPE' qui permet de reconnaître le fichier (4 octets)
- Taille totale du fichier (4 octets)
- Largeur de l'image (4 octets)
- Hauteur de l'image (4 octets)
- Qualité (0 - 100) (1 octet)

➢ TABLES DE QUANTIFICATION

Pour pouvoir choisir plusieurs qualités pour enregistrer les images, il nous fallait plusieurs tables de quantification. Mais nous avons trouvé sur plusieurs sites Internet une formule donnant le pas de quantification en fonction des coordonnées de l'élément :

$$QuantL(i, j) = 1 + Qualité \,.\, (1 + i + j)$$

Cette formule est valable pour la luminance, mais comme nous sommes moins sensibles à la chrominance, nous avons essayé de trouver une autre formule qui conviendrait mieux pour la chrominance. Après plusieurs essais nous avons choisi la formule suivante :

$$QuantC(i, j) = 1 + Qualité \,.\, (1 + i^2 + j^2)$$

Ces formules donnent de bons résultats mais nous avons constaté que la qualité diminuait très rapidement et que le choix dans les qualités basses était trop important par rapport à celui des hautes qualités. Mais il n'était pas possible d'augmenter ce choix en gardant des coefficients de quantifications entiers. Nous avons donc décidé de corriger le terme de *Qualité* en lui appliquant une fonction, qui donnerait des pas de quantification réels. Nous avons fait plusieurs essais, en traçant les fonctions sur une calculatrice graphique pour avoir une idée de ce qu'elles donneront, et nous avons fini par arriver à la formule :

$$Qualité = 2000 \,/\, (Qualité + 16.21) - 17.21$$

Le coefficient 2000 définit la « courbure » de la courbe et donc l'étendue du choix dans les hautes qualités par rapport aux basses qualités, et les coefficients 16.21 et 17.21 ont été calculés afin de renvoyer 0 lorsque qualité est à 100 et renvoyer 99 lorsque qualité est à 1.

Le groupe JPEG propose par ailleurs deux tables de quantification par défaut pour la luminance et la chrominance, que nous avons inclues dans le programme.

➢ ORGANISATION DES DONNEES, RLE, HUFFMAN

Si l'on veut sous échantillonner la chrominance, cela nous oblige à découper l'image en macro blocs de 16x16 pixels, puisque pour appliquer la DCT il faut une matrice de 8x8, et le sous échantillonnage de la chrominance du macro bloc 16x16 le ramène à un bloc de 8x8. Pour chaque macro bloc on doit donc écrire quatre blocs de luminance et deux de chrominance.

Notre première idée était donc de mettre les six blocs 8x8 bout à bout, dans l'ordre dans lequel ils sont disponibles, c'est-à-dire les quatre blocs de luminance puis les deux de chrominance puisqu'il faut lire le macro bloc 16x16 en entier pour sous échantillonner la chrominance. Mais cela posait un problème pour la décompression puisque pour reconstituer l'image il faut avoir à la fois la luminance et la chrominance, et en lisant on avait déjà les quatre blocs de luminance mais on ne pouvais pas dessiner les pixels correspondant puisque l'on n'avait pas la chrominance.

Nous avons alors mis un octet avant chaque macro bloc pour donner la taille totale des quatre blocs de luminance compressés, et pouvoir à la décompression aller lire directement la chrominance puis revenir à la luminance, mais cela a posé un autre problème au niveau de la compression Huffman. En effet nous avons choisi de compresser avec un codage de Huffman *adaptatif*, c'est-à-dire que l'on construit l'arbre au fur et à mesure de la compression de l'image, en commençant avec un arbre qui donne un code de même taille à chaque caractère (puisque même probabilité de chaque caractère : 0) puis en actualisant l'arbre à chaque bloc 8x8 écrit avec le contenu du bloc.

Mais si pendant la compression on écrit d'abord la luminance, puis la chrominance, en les utilisant pour actualiser l'arbre, et qu'à la décompression on va directement lire la chrominance, on n'a pas pu actualiser l'arbre avec la luminance et donc l'arbre n'est pas le même qu'à la compression d'où un décodage complètement faux. Nous nous sommes donc résigné pendant la compression à lire complètement le macro bloc pour écrire la chrominance, puis à le relire une seconde fois pour écrire la luminance.

Nous avons ensuite rencontré un autre problème car il arrive que les coefficients DCT quantifiés soit supérieurs à 255, surtout quand une bonne qualité est choisie. Mais avec Huffman on ne peut compresser qu'un octet à la fois (entre 0 et 255). La solution était donc de coder les coefficients sur deux octets si ils étaient supérieurs à 255, mais comment savoir au décodage si il s'agissait d'un coefficient sur deux octets ou de deux coefficients sur un octet ? Nous avons cherché un moment puis nous nous sommes rendu compte qu'il était facile avec l'arbre de Huffman d'ajouter des caractères spéciaux. Nous avons donc rajouté un caractère 256, que l'on place avant d'écrire un coefficient sur deux octets, de sorte qu'à la décompression quand on rencontre ce caractère on sait que les deux octets suivants codent un même coefficient.

Ce principe de caractère spécial nous a aussi ouvert la porte à une autre amélioration. Jusque-là on codait tous les 0 avec une RLE : quand on rencontrait un 0, il y avait obligatoirement le nombre de 0 de suite juste après. C'était le plus simple mais cela présentait un inconvénient puisque quand un 0 est seul on le code avec deux octets au lieux d'un. Nous avons donc rajouté un autre caractère spécial, 257, qui est l'équivalent du 0 dans le cas précédent car on place juste après le nombre de 0 de suite, mais en revanche lorsqu'un 0 est isolé on écrit simplement 0 et on ne perd pas de place.

Dans notre traque de la place perdue nous avons trouvé une dernière amélioration, qui consiste à coder la taille des RLE et des grands coefficients non pas sur un nombre d'octets rond mais juste sur le nombre de bits qui est nécessaire. En effet pour la RLE, le nombre de 0 se suivant ne peut dépasser 64 puisque les blocs ne comprennent eux-mêmes que 64 éléments. Il est donc inutile de stocker ce nombre sur un octet (8 bits, 256 possibilités) alors que 6 bits suffisent (64 possibilités).

De la même manière les coefficients DCT ne peuvent pas dépasser en valeur absolue le coefficient continu (par définition), qui lui-même ne peut dépasser 2040 (255 * 8, moyenne maxi des pixels multipliée par 8, par définition). Il est donc inutile d'utiliser deux octets (16 bits, 65536 possibilités) alors que les coefficients ne peuvent osciller qu'entre -2040 et 2040 et donc 12 bits suffisent (4096 possibilités).

Cette optimisation faite, nous avons eu l'idée de ne construire l'arbre de Huffman qu'avec 128 caractères, ce qui réduisait la taille de chaque code de un bit, et lorsque le coefficient dépassait 127 l'écrire avec la méthode précédente sur 12 bits. En pratique le gain de place est assez faible, car les coefficients supérieurs à 127 devant être écrits sur 12 bits compensaient en partie le gain sur les autres coefficients, mais nous avons choisi de garder cette solution car l'arbre de Huffman étant deux fois plus petits occupe moins de mémoire et est beaucoup plus rapide à construire.

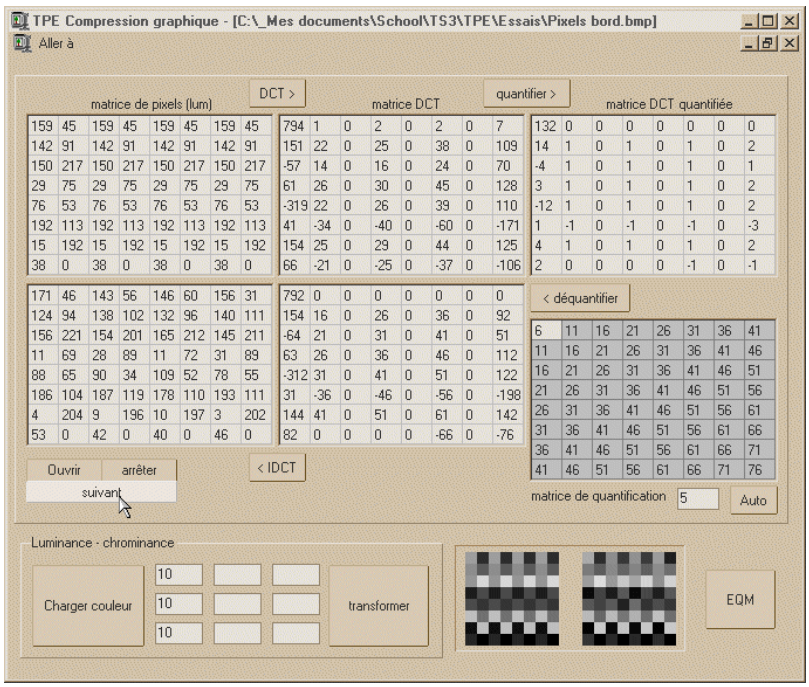
➢ PIXELS DES BORDS DE L'IMAGE

Quand les dimensions de l'image ne sont pas des multiples de 16, il reste des pixels sur les bords de l'image que l'on ne peut traiter de la même façon. Nous n'avons trouvé nulle part la méthode utilisée : sur tous les sites Internet que nous avons vus les auteurs disent qu'une autre méthode est utilisée mais il n'y a jamais plus de précision. Nous voulions donc au début simplement sous échantillonner la chrominance, puisque cela divisait déjà la taille par deux, mais cela ne permet pas de choisir la qualité.

En examinant des images compressées en JPEG par des logiciels de dessin, nous avons constaté que les bords de l'image avaient exactement les mêmes propriétés que le reste de l'image, à savoir que des blocs de 8x8 se détachaient (même s'ils étaient tronqués), ce qui montre que la méthode utilisée est très semblable.
Nous nous sommes alors rendu compte que même si l'on n'a pas une matrice 8x8 complète, on peut quand même opérer une DCT dessus par exemple en complétant tous les éléments vides par des 0. On peut alors coder le bloc de la même façon, et ne retenir à la décompression que les pixels utiles (possible puisque l'on connaît les dimensions de l'image). Mais cela oblige à stocker une matrice 8x8 complète même si il n'y a qu'une colonne de pixels en trop, ce qui gaspille de la place.

Cependant, la DCT faisant ressortir les fréquences, nous nous sommes dit qu'il était peut-être possible de simplifier la matrice DCT résultante en périodisant d'une certaine façon la matrice d'entrée, et nous avons donc fait plusieurs essais.

Pour effectuer vous-même les tests, choisissez 'Essais JPEG 1' dans le menu 'Aller à', puis cliquez sur le bouton 'Ouvrir' pour ouvrir le fichier 'pixels bord.bmp', puis appuyez sur le bouton 'DCT >' pour voir la DCT de la matrice de pixels. Cliquez sur le bouton 'Suivant' pour accéder aux différents exemples.



Quand on a une colonne de pixels et que l'on complète les autres éléments avec des 0, la DCT résultante est complexe et on doit l'enregistrer en entier pour retrouver les pixels de départ. Par contre, quand on reporte la colonne de pixels dans les autres colonnes de la matrice, la DCT résultante comporte des nombres dans la première colonne et tous les autres éléments sont à 0. On peut donc quantifier de la même façon cette matrice puis ne stocker que la première colonne.

Pour deux colonnes de pixels, la DCT est toujours complexe si on complète les autres éléments avec des 0, mais si on reporte les deux colonnes dans la matrice, la DCT comporte trois colonnes à 0 (voir copie d'écran précédente). Cela oblige encore à stocker 5 colonnes de la DCT ce que nous avons trouvé décevant. Nous avons alors eu l'idée de reporter les deux colonnes d'une autre manière, en les « dépliant » c'est-à-dire en inversant les deux colonnes à chaque fois, de sorte qu'il n'y ait pas de discontinuité (un peu comme la définition de transformée en cosinus par rapport à la transformée de Fourier). Le résultat est alors bien meilleur puisque la DCT comporte 6 colonnes à 0 et il n'y en a plus que deux à stocker.

Nous avons alors voulu appliquer la même chose pour trois et quatre colonnes. Pour quatre il n'y a pas de problème, 4 colonnes DCT sont à 0, mais pour trois colonnes, étant donné que 8 n'est pas un multiple de 3 on ne peut pas reporter les 3 colonnes un nombre entier de fois. Mais en revanche on peut les stocker en deux fois, en stockant une fois 2 colonnes et une fois 1 colonne, et de cette manière il n'y a que 2 + 1 = 3 colonnes DCT à stocker.

On peut faire de même pour 5, 6 ou 7 colonnes, et on a ainsi toujours la même quantité de données DCT à stocker que d'origine, ce qui permet de compresser avec la même efficacité les pixels du bord que le reste de l'image.

On peut résumer ainsi la périodisation des matrices en fonction du nombre de colonnes de pixels, les lettres représentant des colonnes :

```
- 1 → AAAAAAAA → x0000000 ........................ = 1 colonne à stocker
- 2 → ABBAABBA → x000x000 ........................ = 2 colonnes à stocker
- 3 → ABBAABBA + CCCCCCCC → x000x000 + x0000000 .. = 3 colonnes à stocker
- 4 → ABCDDCBA → x0x0x0x0 ........................ = 4 colonnes à stocker
- 5 → ABCDDCBA + EEEEEEEE → x0x0x0x0 + x0000000 .. = 5 colonnes à stocker
- 6 → ABCDDCBA + EFFEEFFE → x0x0x0x0 + x000x000 .. = 6 colonnes à stocker
- 7 → ABCDDCBA + EFFEEFFE + GGGGGGGG  ............. = 7 colonnes à stocker
```

Les résultats sont les mêmes pour les lignes que pour les colonnes, et peuvent même se coupler (par exemple pour le coin inférieur droit).

Nous avons finalement comparé les résultats de notre format à celui du format JFIF (.jpg), et nous avons été satisfaits puisqu'il donne des résultats comparables. Il est difficile d'être plus précis car étant une méthode à pertes, il faut juger quelle image a la meilleure qualité avec une taille identique.

C'est pourquoi nous avons décidé de mettre en œuvre le calcul de l'EQM vu dans la première partie, même si nous avons trouvé cette technique discutable, car le principe de la compression JPEG est justement de perdre des données où elles seront le moins visibles, en se servant des défauts de notre œil, et cet EQM ne reflète pas la qualité 'visible'. Cependant pour comparer notre format au format JFIF, étant donnés qu'ils utilisent la même méthode, le résultat devrait être fiable.

Nous avons donc enregistré une image en JTPE, puis en JFIF avec le logiciel PaintShopPro en choisissant la qualité de manière à ce que la taille du fichier soit la même que celle du fichier JTPE. Nous avons ensuite réenregistré les fichiers en BMP, puis calculé l'EQM dans les deux cas.

Pour effectuer vous-même les tests, choisissez 'Essais JPEG 1' dans le menu 'Aller à', puis cliquez sur le bouton 'EQM', puis choisissez successivement l'image de référence et l'image à examiner. Vous pouvez créer vous-même vos fichiers ou utiliser les images fournies dans le dossier 'Comparaison JPEG' du cd-rom.

Nous avons alors été déçus car il se révèle que notre format est moins performant que le format JFIF, par exemple une EQM de 23 contre 18, ou 43 contre 37. La différence est flagrante, mais en fait étant donné qu'il s'agit d'une moyenne quadratique, si l'on prend les racines carrées la différence n'est plus que de 0.5, ce qui est négligeable sur 256. L'écart peut aller jusqu'à 1.5 pour des taux plus élevés, mais cela reste très peu visible.

En revanche au niveau de la vitesse, notre algorithme est beaucoup plus lent. En effet notre code n'est pas du tout optimisé, contrairement à celui des logiciels spécialisés.

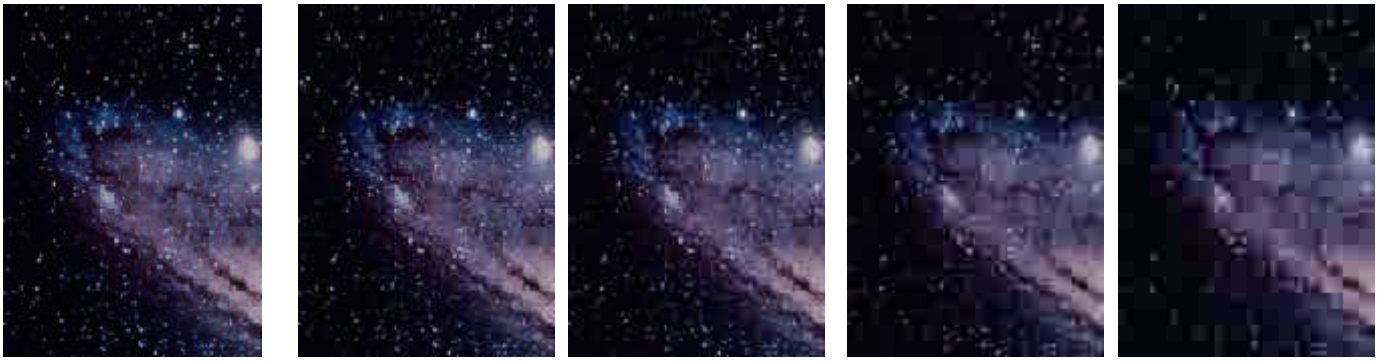
3.3. ETUDE DES PERTES DUES A LA COMPRESSION

3.3.1. CARACTERISTIQUES DE NOTRE FORMAT JTPE ET DEMARCHE

- **IMAGES NATURELLES**

Examinons une image d'Andromède compressée à plusieurs niveaux en JTPE (vous pouvez retrouver ces images sur le cd-rom dans le dossier 'Pertes JPEG') :

Image originale (66 Ko) JTPE 80 (4.94 Ko) JTPE 60 (2.47 Ko) JTPE 40 (1.41 Ko) JTPE 20 (924 o)



On se rend compte qu'en qualité 80 aucune différence n'est visible alors que la taille de l'image a été divisée par 13. Quand la qualité diminue, on voit deux symptômes apparaître :

- certaines étoiles sont atténuées ou disparaissent
- des blocs uniformes commencent à apparaître

On peut expliquer cela avec le fonctionnement de la compression JPEG.

En effet la base du JPEG et des pertes engendrées est le filtrage des hautes fréquences, car nous y sommes moins sensibles. Mais dans certaines limites, et les hautes fréquences, ce sont les détails. Cela explique donc la disparition des étoiles quand on augmente le taux.

De plus les arrondis effectués lors de la quantification, si le pas est grand peuvent amener à réduire significativement voire supprimer les amplitudes de certaines fréquences. Ceci a pour effet d'atténuer les variations de couleur, et de les rapprocher de la valeur du coefficient continu (qui rappelons-le est une

moyenne du bloc). Cela a donc pour effet d'atténuer le contraste au sein d'un bloc, et d'uniformiser les blocs, ce qui explique leur apparition nette lorsque l'on pousse la compresssion.

Mais nous avons découvert que dans certains cas peut apparaître un phénomène beaucoup plus gênant. Celui-ci est illustré avec les deux images suivantes. La première est l'original et la seconde est fortement compressée (JTPE 20) :

On constate à l'endroit indiqué par la flèche que des parasites apparaissent. Ce phénomène apparaît pour de forts taux de compression, et lorsque une petite moitié d'un bloc est d'une couleur, et le reste d'une autre couleur, les deux couleurs ayant un fort contraste entre elles.

On peut expliquer ce phénomène par la présence de hautes fréquences dans l'image originale, qui avaient pour rôle de compenser les fréquences plus basses qui tendaient à faire réapparaître la première couleur. Mais lors de la compression les coefficients des hautes fréquences ont été supprimés, annulant l'effet de compensation, et permettant à la première couleur de réapparaître.

Ces défauts interdisent donc le JPEG pour certaines utilisations. Il serait en effet très gênant de voir sur une photographie astronomique des étoiles qui en fait n'existent pas. Cependant il peut tout de même convenir si l'on utilise des taux très modérés qui n'altèrent pas la qualité de l'image. Mais bien entendu on perd une part de l'intérêt du JPEG, qui est d'offrir des taux très élevés.

- **IMAGES GEOMETRIQUES**

Nous avons dit au début que le JPEG n'était pas adapté aux images géométriques, mais sans le justifier. Maintenant que nous connaissons le fonctionnement du JPEG, cela est facile à faire.

Les images naturelles ont la plus grosse partie de l'information concentrée dans les basses fréquences. Cela se voit en effectuant la DCT, les coefficients dans la partie inférieure droite sont naturellement plus faibles que dans la partie supérieure gauche. En effet les hautes fréquences correspondent aux changements brusques de couleurs, et dans les images naturelles les changements sont progressifs.

De plus, nous sommes moins sensibles à ces hautes fréquences, ce dont se sert JPEG en les supprimant (ou du moins en perdant de la précision dessus). Mais dans le cas des images géométriques, il y a au contraire beaucoup de changements brusques de couleur correspondant à des limites de figures, qui vont être dégradés par la compression.

On le constate avec l'image suivante : 

Mais ces pertes plus importantes sont aussi accentuées par un autre phénomène : une image géométrique est généralement beaucoup plus simple qu'une image naturelle, et on sait quelle étendue doit être exactement de la même couleur, où doit-il y avoir une ligne nette, ce qui fait que le moindre écart de couleurs où il ne devrait pas y en avoir ne nous échappe pas, et nous choque beaucoup plus que sur une image naturelle.

Le JPEG n'est donc pas utilisé pour stocker des images géométriques car il n'y est pas adapté sur le point de vue de la qualité, mais surtout ce type d'image est très bien géré par les méthodes conservatives, que ce soit la RLE, le prédictif, la LZW, qui offrent d'aussi bon sinon meilleurs taux de compression que le JPEG.

3.3.2. IMAGES SATELLITES

La compression est très utilisée dans les images satellites, par exemple pour les cartes topographiques. Depuis 1986, les fonctions de la télédétection (observer la Terre à distance et couvrir d'une seule image une zone assez vaste), faites par les satellites français Spot, ont recueillies plus de 4 millions d'images !

Par exemple, une carte nationale numérique au 1:25000 est fournie à une résolution de 500dpi et couvre une zone de 17 sur 12 km.

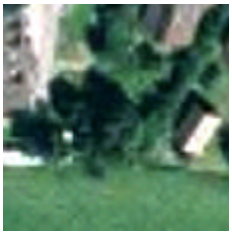
Pour couvrir la Suisse, il faut 260 images de 14000 x 9600 pixels. Actuellement ces images sont livrées en 8 couleurs, mais pour inclure les reliefs, on utilise 24 bits. Ce qui donne (24 * 14000 * 9600 * 260 / 8) octets, soit plus de 97 Gigaoctets en Bitmap. Mais les images sont distribuées en format TIFF, autre format très connu, qui permet de compresser des images avec plusieurs algorithmes. Et dans le format TIFF-LZW, les images font 4 gigaoctets pour toute la Suisse (gain de 95% !).

Les différentes méthodes utilisées sont la compression JPEG par ondelettes, et certains autres algorithmes variants développés par des sociétés.

Similairement aux clichés compressés avec le JTPE, les images aériennes se présentent ainsi :

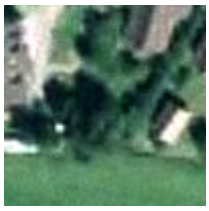
Image originale (Copyright Swissimage).

Compression JPEG Standard (37KB). Les défauts de la compression JPEG standard sont nettement visibles par l'effet de damier correspondant au fenêtrage utilisé par l'algorithme de compression.



La compression par l'algorithme commercial de LizardTech, MrSID, présente également un relativement bon rendu visuel mais on observe un déplacement latéral (de l'ordre de trois pixels vers la droite) de l'information ce qui parait non négligeable lorsque l'on désire conserver la géoréférence des images aériennes ou satellitaires.

Compression JPEG2000. La compression par JPEG2000 présente un résultat qui allie un bon rendu visuel de l'image et la conservation de la géométrie des objets que l'on distingue dans l'image.



Human Vision S (strong CSF filtering & synthetic texture). La compression par les algorithmes prenant en compte la perception humaine de la couleur (Human Vision S) présente les meilleurs résultats de conservation de l'aspect visuel de l'image et de la géométrie des objets présents.

On remarque que les groupes privés sont à la pointe de la technologie en matière de compression.

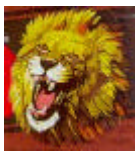
BILAN

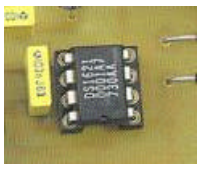
Pour bilan, voici un comparatif des formats les plus répandus et utilisés (gérés avec MS PhotoEditor) et de nos propres formats, que nous allons commenter.

Ce tableau donne les gains de compression (en %, plus il est élevé plus le gain est grand) en fonction des formats et des images de différents types :

Format	***GIF*** *(LZW)*	***PNG*** *(LZW)*	***TIFF*** *(LZW)*	***ZIP*** *(LZW & stat)*	***WTPE*** *(LZW)*	***PTPE*** *(préd. & stat)*	***JTPE*** *(JPEG)*
'Equation' (4 bits)	**89**	**90**	**83**	**94**		X	X
'Terre' (4 bits)	**50**	**51**	**46**	**58**		X	X
'Billets' (8 bits)	**34**	**43**	**34**	**43**		X	X
'Lion' (8 bits)	**42**	**48**	**43**	**49**		X	X
'Eclair' (24 bits)	X	**20**	**14**	**21**		**34**	**95**
'Papillon' (24 bits)	X	**3**	**18**	**8**		**31**	**94**
'Puce' (24 bits)	X	**20**	**-9**	**8**		**20**	**94**



Equation **Terre** **Billets** **Lion** **Eclair** **Papillon** **Puce**

➢ COMPARAISON DES METHODES

On confirme donc que pour des images de faibles profondeurs (faible nombre de couleurs) et très géométriques (comme 'Equation'), les taux sont très bons avec la méthode LZW. Mais dès que l'image devient un peu plus complexe, ou que la profondeur augmente, les taux redescendent vite.

Ils sont encore de 50% en 256 couleurs (8 bits), soit une taille divisée par 2, mais dès que l'on arrive en 24 bits cette méthode montre ces limites. Le codage prédictif se révèle alors nettement meilleur, mais les taux restent encore très modestes, alors que le JPEG divise la taille des fichiers par 20 pour une perte de qualité indécelable.

➢ COMPARAISON DES FORMATS

On constate également des différences aux niveaux des formats, qui pourtant utilisent la même méthode. Cela montre que les méthodes sont paramétrables, et chaque format « l'accommode à sa sauce ».

Ces différences s'accentuent nettement pour les images 24 bits, et on est même surpris de voir que le TIFF a un taux négatif pour l'image 'Puce', ce qui veut dire que l'image compressée est plus grosse que l'image non compressée. Plutôt surprenant pour un format aussi répandu.

On constate de plus que le format ZIP offre toujours les meilleurs taux, alors que ce n'est pas un format spécialisé dans la compression des images.

CONCLUSION

La compression graphique peut donc être considérée comme une branche à part de la compression. Si elle peut utiliser seulement des algorithmes généraux (LZW), ceux-ci montrent rapidement leurs limites, surtout pour les images en couleurs vraies. Il faut alors utiliser les propriétés des images afin de trouver de nouvelles méthodes qui y sont adaptées (codage prédictif), quitte à perdre un peu de l'information pour arriver à des taux beaucoup plus élevés (JPEG).

En effet, il serait illusoire de croire que la compression devient obsolète à cause de la capacité des mémoires qui augmente sans cesse, car la quantité de données à conserver croît de la même manière, et les réseaux évoluent pour leur part beaucoup moins vite. La compression peut même apporter un gain de vitesse car par exemple il est plus rapide d'enregistrer une image sur son disque dur en JPEG qu'en BMP car on perd moins de temps à compresser l'image qu'à écrire plus de données sur le disque dur.

Le travail sur les méthodes est toujours en amélioration. En 1998, R. K. Young, le créateur d'ARJ, a lancé un nouveau compresseur, JAR, encore plus performant ! La compression JPEG par ondelettes est elle aussi assez récente, et on étudie de près en ce moment la compression fractale.

L'exploration de notre univers nous révèle de nombreux secrets qu'il faut conserver. L'espionnage, la cartographie, la médecine, … sont quelques domaines parmi tous ceux qui nécessitent un stockage important, où le compression, conservative ou non, est indispensable.

La compression est gagnante sur tous les plans, et c'est pourquoi elle reste encore un front d'innovations.

ANNEXE A : BIBLIOGRAPHIE

- **PAGES INTERNET**

  http://perso.libertysurf.fr/IPhilGood/Codage/Compressionjpeg.htm
  http://saturn.umh.ac.be/~olivier/CompressionInformatique/node41.html
  http://compressions.multimania.com
  http://www.chez.com/algorithmejpeg
  http://discon.multimania.com/comp.htm
  http://www.edunet.tn/ressources/formation/edutic1/modules/frontpage/formats.htm
  http://ruses.com/Pages/0001000C.htm
  http://www.data-compression.com/lossless.html

- **LIVRES**

  Delphi3, Secrets d'experts, Editions Simon & Schuster Macmillan (France), 1997

ANNEXE B : CODE SOURCE EN PASCAL POUR DELPHI DES ELEMENTS CLES DU PROGRAMME

Nous vous rappelons que les sources complets sont disponibles sur le cd-rom inclus dans le dossier.

Vous pourrez trouver dans le fichier '_Bibliotheque.pas' les fonctions utilisées par plusieurs formats, c'est-à-dire les fonctions d'ouverture et d'enregistrement de fichiers BMP, le codage de Huffman, l'écriture et lecture d'entiers sur un nombre variable de bits, de calcul de l'EQM.

Le code correspondant à chaque format se trouve dans son fichier homonyme, la RLE dans le fichier '_RTPE', le codage prédictif dans le fichier '_PTPE', le JPEG dans le fichier '_JTPE', la LZW dans le fichier '_WTPE'.

Implémentation de la DCT :

```
Type
  Tab8x8: Array[0..7, 0..7] Of Integer;



//--- stricte application de la formule
Procedure DCT(Var Source, Resultat: Tab8x8);
Var                                          // déclaration des variables
  i, j, x, y: Integer;                       // compteurs de boucles
  c, Temp: Double;                           // 'c' = coefficient c(a)
Begin
  For i := 0 To 7 Do For j := 0 To 7 Do      // boucles pour traiter chaque élément de la DCT
  Begin
   c := 1; Temp := 0;                        // initialisations
   If i = 0 Then c := c / Sqrt(2);           // calcul de c(a), Sqrt(2) = racine carrée de 2
   If j = 0 Then c := c / Sqrt(2);           // ...
   For x := 0 To 7 Do For y := 0 To 7 Do     // calcul des deux sigmas
     Temp := Temp + Source[x,y] * cos((x + 0.5)*i*Pi/8) * cos((y + 0.5)*j*Pi/8);
   Resultat[i, j] := Round(c * 0.25 * Temp); // normalisation du résultat (1 / V(2N) * c(i) * c(j) )
  End
End;



//--- stricte application de la formule
Procedure IDCT(Var Source, Resultat: Tab8x8);
Var
  i, j, x, y: Integer;
  c, Temp: Double;
Begin
  For x := 0 To 7 Do For y := 0 To 7 Do      // boucles pour traiter chaque element de l'IDCT
  Begin
    Temp := 0;
    For i := 0 To 7 Do For j := 0 To 7 Do    // calcul des deux sigmas
    Begin
      c := 1;                                // on initialise ...
      If i = 0 Then c := c * UnSurRac2;      // et calcule le coefficient c(a) à chaque fois ...
      If j = 0 Then c := c * UnSurRac2;      // comme dans la formule
      Temp := Temp + c * Source[i, j] * cos((x + 0.5)*i*Pi/8) * cos((y + 0.5)*j*Pi/8);
    End;
    Resultat[x, y] := Round(Temp * 0.25);    // normalisation du résultat (1 / V(2N) )
  End
End;
```

Implémentation de la modélisation et création de l'arbre de Huffman :

```
Type
  sNoeud = Record                                  // déclaration du type sNoeud
    ndFils: Array[0..1] Of Word;                   // qui contient l'adresse des deux fils
    ndPere: Word;                                  // celle du père
    Code: Byte;                                    // le code du nœud (fils 0 ou 1 du père ?)
    Poids: Cardinal;                               // le poids (nb d'occurences)
  End;

Const
  nbCar = 130;                                     // constante qui définit la taille de l'arbre
Var
  Arbre: Array[0..(nbCar - 1) * 2] Of sNoeud;      // arbre lui-même, tableau de noeuds


//---  renvoie dans 'Result' les indices des deux noeuds de plus faible poids non déjà traités
Procedure Donner2PlusPetits(Max: Integer; Var Result: Array Of Word);
Var
  i: Integer;
  Grand: ^Word;  // pointeur sur le nœud de plus grand poids des deux stockés dans 'Result'
Begin
  For i := 0 To Max Do If Arbre[i].Code = 3 Then
    Begin Result[0] := i; Break End;               // on recherche le premier noeud non déjà traité
  For i := i + 1 To Max Do If Arbre[i].Code = 3 Then
    Begin Result[1] := i; Break End;               // on recherche le second noeud non déjà traité
  If Arbre[Result[0]].Poids > Arbre[Result[1]].Poids Then
    Grand := @Result[0] Else Grand := @Result[1]; // faire pointer 'Grand' sur celui de + grand poids
  // on parcourt tous les noeuds
  For i := i + 1 To Max Do
    If (Arbre[i].Code = 3) And (Arbre[i].Poids < Arbre[Grand^].Poids) Then
    Begin           // et si le noeud a un poids inférieur au plus grand et n'a pas encore été traité
      Grand^ := i;  // on le met à la place du plus grand
      // et on revérifie que 'Grand' pointe sur le noeud de plus grand poids
      If Arbre[Result[0]].Poids > Arbre[Result[1]].Poids Then
        Grand := @Result[0] Else Grand := @Result[1];
    End;
  // on fait bien attention que le plus petit élément est dans Result[0]
  If Result[0] > Result[1] Then
    Begin i := Result[0]; Result[0] := Result[1]; Result[1] := i End
End;

//---  construit l'arbre
Procedure ConstruireArbre;
Var
  i: Integer;
  Petits: Array[0..1] Of Word;
Begin
  For i := 0 To nbCar - 1 Do                                  // initialiser l'arbre en créant les feuilles
    Begin Arbre[i].ndFils[0] := i; Arbre[i].ndFils[1] := nbCar * 2 - 1 End;
  For i := 0 To (nbCar - 1) * 2 Do Arbre[i].Code := 3;    // initialiser l'arbre
  For i := nbCar To (nbCar - 1) * 2 Do
  Begin  // pour chaque nouveau nœud (après les feuilles)
    Donner2PlusPetits(i - 1, Petits);  // on recherche les deux nœuds de plus faible poids, les fils
    Arbre[i].ndFils[0] := Petits[0];   // on écrit les adresses des fils dans le père (nœud en cours)
    Arbre[i].ndFils[1] := Petits[1];   //
    Arbre[Petits[0]].ndPere := i;      // on écrit l'adresse du père dans les deux fils
    Arbre[Petits[1]].ndPere := i;      //
    Arbre[Petits[0]].Code := 0;        // on écrit les codes des deux fils
    Arbre[Petits[1]].Code := 1;        // v  le père a comme poids la somme de ceux de ses fils
    Arbre[i].Poids := Arbre[Petits[0]].Poids + Arbre[Petits[1]].Poids;
  End;
  Arbre[(nbCar - 1) * 2].ndPere := nbCar * 2 - 1;
End
```