Licence de Sciences et Technologies
S4 – mention Informatique

Université de Perpignan Via Domitia
V. Zucca

TP3 Algorithmique des images, textes et données

Compression JPEG.

Le but de ce TP est de créer un programme permettant de compresser une image à l'aide d'une partie de l'algorithme JPEG.

Attention : tout autre include que `stdlib.h`, `stdio.h` et `math.h` est interdit ! De plus, vos programmes et fonctions ne causeront pas de fuites mémoires.

1 Conversion depuis le format `ppm` vers le format `pgm`

Le format de fichier *Portable PixMap* (`ppm`) permet d'enregistrer des images en couleur ou chaque pixel est codé en RGB. Le système RGB code un pixel par 3 octets, chaque octet prend donc une valeur entre 0 et 255, donnant respectivement le ton de rouge, vert et bleu. Ainsi en ouvrant une image de format (`ppm`) préalablement enregistrée en ASCII avec un éditeur de texte type `gedit` vous pourrez observer le code RGB de chaque pixel. Par exemple en ouvrant le fichier `lena.ppm` vous observerez le format suivant :

```
P3 <-- Code pour ppm
# Une ou plusieurs lignes
# de commentaires
512 512 <-- largeur et hauteur
255 <-- valeur maximale possible
224 <-- code R du pixel 0
136 <-- code G du pixel 0
129 <-- code B du pixel 0
224 <-- code R du pixel 1
136 <-- code G du pixel 1
129 <-- code B du pixel 1
...
...
```

où les pixels sont numérotés ligne par ligne. Le pixel du coin supérieur gauche est à la position 0, son voisin de droite est à la position 1 etc... :

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

De la même façon, le format *Portable GrayMap* (`pgm`) permet d'enregistrer des images en niveaux de gris. Dans ce cas, chaque pixel est représenté par un seul octet (une valeur entre 0 et 255). Le contenu d'un fichier `pgm` est similaire à celui d'un `ppm` et ressemble à :

```
P2 <-- Code pour pgm
# Une ou plusieurs lignes
# de commentaires
512 512 <-- largeur et hauteur
255 <-- valeur maximale possible
162 <-- code du pixel 0
162 <-- code du pixel 1
162 <-- code du pixel 2
162 <-- code du pixel 3
162 <-- code du pixel 4
159 <-- code du pixel 5
...
...
```

Cette partie a pour but de créer un programme permettant de convertir une image couleur enregistrée au format `ppm`, en une image en noir et blanc enregistrée au format `pgm`.

On rappelle que les tons de gris sont donnés par la composante Y du modèle YUV et que la conversion entre le RGB et le YUV se fait grâce à la transformation linéaire suivante :

$$\begin{cases} Y &= 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \\ U &= -0.147 \cdot R + 0.289 \cdot G + 0.436 \cdot B \\ V &= 0.615 \cdot R - 0.515 \cdot G - 0.100 \cdot B \end{cases}$$

Par la suite nous représenterons un pixel codé en RGB à l'aide de la structure `RGB` suivante :

```
struct RGB_t
{
  unsigned char R, G, B;
};
typedef struct RGB_t RGB;
```

Une image est formée de `largeur` × `hauteur` pixels. Nous représenterons donc une image avec un tableau 2D de pixels. L'image pouvant être en cours ou en nuance de gris notre structure `picture` doit comporter soit un tableau de `unsigned char` soit un tableau de `RGB`. Afin d'éviter de stocker les deux tableaux et de perdre de la mémoire inutilement nous allons utiliser la structure d'`union` :

```
struct picture_t
{
  unsigned char type; /* Type de l'image (0 ou 1) */

  unsigned int hauteur, largeur;

  unsigned char value_max; /* Valeur max utilisee pour representer les pixels */

  union
  {
    unsigned char **pixels; // un pointeur vers un tableau 2D d'unsigned char (pour les pgm)
    RGB **pixels_rgb; // un pointeur vers un tableau 2D de pixels_rgb (pour les ppm)
    };
  };
typedef struct picture_t picture;
```

1. Créez une fonction `picture *get_picture(char *file_name)` qui stocke les information de l'image enregistrée au format `pgm` ou `ppm` dans le fichier `file_name`. Votre fonction gèrera les cas d'erreurs éventuels en affichant un message d'erreur adapté.
2. Créer une fonction `void free_picture(picture *image)` qui libère la mémoire allouée pour la structure de type `picture` pointée par `image`.
3. Créer une fonction `double get_Y_component_from_RGB(RGB pixel)` qui renvoit la valeur de la composante Y du modèle YUV d'un pixel `pixel` représenté en RGB.
4. Créer une fonction `picture *ppm_to_pgm(picture *image)` qui convertit une image `ppm`, enregistrée dans la structure pointée par le paramètre `image`, en une image au format `pgm`. Les valeurs de la composante Y seront arrondies à l'entier le plus proche à l'aide de la fonction `round` définie dans `math.h`.
5. Créer une fonction `void write_picture(picture *image, char *file_name, int binary)` qui enregistre l'image `image` au format `pgm` ou `ppm` dans un fichier dont le nom `file_name` est passé en paramètre. Le paramètre `binary` permettra de spécifier si l'enregistrement se fait en ASCII ou en binaire.
6. Créer un programme qui convertit une image enregistrée au format `ppm` vers le format `pgm`. Votre programme prendra le nom du fichier contenant l'image en paramètre et enregistrera le nouveau fichier sous le même nom. Par exemple l'exécution de : `./main lena.ppm` créera un fichier `lena.pgm`.

2 Compression d'une image au format PGM

De manière générale la compression JPEG permet de compresser des images couleurs. Par souci de simplicité, nous nous restreindrons ici à la compression de la composante Y d'une telle image, c'est à dire d'une image au format `pgm`.

La première étape de la compression consiste à subdiviser l'image en bloc de 8×8 pixels et d'appliquer une transformée en cosinus discrète à chaque bloc. On rappelle ci dessous la formule de la transformée en cosinus discrète en deux dimensions pour un bloc de taille $N \times N$:

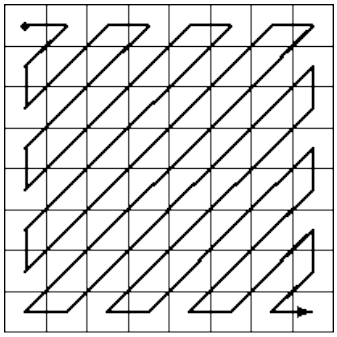

FIGURE 1 – Extraction en zigzag des valeurs d'un bloc 8×8

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

avec $C(z) = 1$ pour $z > 0$ et $C(0) = 1/\sqrt{2}$. Une fois la transformée appliquée, on applique l'étape de *quantification* qui consiste à diviser le coefficient $\texttt{DCT}(i,j)$ d'un bloc par le coefficient $Q_{i,j}$ de la matrice de quantification Q donnée ci dessous :

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

Le résultat de la division sera arrondi à l'entier le plus proche. À partir de ce moment là, les seules valeurs non nulles seront présentes dans la partie supérieure gauche du bloc 8×8. Afin de pouvoir réaliser une compression RLE du résultat, on extrait les valeurs du bloc en zigzag comme indiqué dans la Figure 1.

de sorte à obtenir une liste de 64 entiers dont les dernières valeurs sont toutes nulles. Ces dernières valeurs seront compressées à l'aide d'un algorithme RLE. Dans cette partie on supposera que les images passées en paramètres ont des dimensions (largeur et hauteur) divisibles par 8.

7. Créer une fonction `void extract_bloc(picture *image, double bloc[8][8], int i, int j)` qui extrait le bloc 8×8 formé de la composante Y de l'image `ppm` pointée par `image` dont le coin supérieur gauche se trouve aux coordonnées (i, j). Ce bloc sera sauvegardé dans le tableau de double `bloc` passé en paramètre.
8. Créer une fonction `void DCT(double bloc[8][8])` qui applique la transformée en cosinus discrète bi-dimensionnelle à un tableau `bloc` de taille 8×8. Pensez à vérifier le bon fonctionnement de votre fonction avec l'exemple du cours.
9. Créer la fonction `void quantify(double bloc[8][8])` qui quantifie le bloc `bloc` passé en paramètre. La matrice de quantification Q sera déclarée comme une variable globale.
10. Créer la fonction `void zigzag_extraction(double bloc[8][8], int zigzag[64])` qui extrait les 64 nombres contenus dans le bloc `bloc` de taille 8×8 dans l'ordre donné par la Figure 1. Les valeurs de `bloc` seront arrondies à l'entier le plus proche avant d'être stockées dans le tableau `zigzag` passé en paramètre.
11. Créez la fonction `void compress_RLE(FILE *f, int zigzag[64])` qui écrit les entiers contenus dans le tableau `zigzag` dans le fichier pointé par `f`. On supposera que le flux donné par `f` aura été ouvert préalablement. Chaque entier sera écrit sur une ligne différente et une séquence de n 0 sera codée par `@`n dès que $n \geq 2$.
12. À l'aide des fonctions précédentes, créer une fonction `void jpeg_compression(picture *image, char *file_name)` qui compresse l'image `pgm` pointée par `image` en utilisant l'algorithme de compression JPEG et stocke le résultat dans un fichier dont le nom est donné par `file_name`. Le fichier compressé respectera le format suivant :

```
JPEG
largeur hauteur
valeur 0
```

```
valeur 1
valeur 2
...
...
```

où `largeur` et `hauteur` correspondront à la largeur et la hauteur de l'image compressée.

13. Créez une fonction `unsigned int file_size(char *file_name)` qui renvoit la taille en octets du fichier nommé `file_name`. On supposera que chaque caractère du fichier `file_name` est codé sur 1 octet. Vous pourrez vérifiez le résultat de votre fonction en le comparant avec le résultat de la commande : `wc -c file_name`
14. Complétez votre programme précédent de sorte à ce qu'il compresse avec l'algorithme JPEG l'image `pgm` extraite du fichier `ppm` passé en paramètre. Le fichier compressé portera le même nom que le fichier passé en paramètre. Par exemple le fichier `lena.ppm` sera compressé dans un fichier `lena.jpeg`. Votre programme affichera également la taille du fichier `ppm` créé, la taille du fichier `jpeg` ainsi que le taux de compression. Par exemple l'exécution de votre programme sur le fichier `lena.ppm` fourni avec le TP donnera le résultat suivant :

```
./main lena.ppm
Le fichier lena.pgm fait 968860 octets.
Le fichier lena.jpeg fait 115257 octets.
Le taux de compression de ce fichier est de 0.12.
```

Remarque : évidemment nous avons seulement implémenté une version simplifiée et non standard de l'algorithme JPEG, ainsi le fichier `jpeg` que nous avons créé ne pourra pas être décompressé par un lecteur d'image traditionnel. En revanche les personnes intéressées sont libres d'implémenter l'algorithme de décompression correspondant à notre algorithme et observer la différence de qualité entre l'image de départ et l'image décompressée.