Projet de Compression de données
JPEG

Colas Sébastien

Chapitre 1
Introduction

1.1 La compression d'images

Avec l'arrivée d'Internet il a fallu se pencher sur le problème de la compression de données transitant à travers ce réseau, en effet on peut difficilement imaginer de faire transiter toutes les données non compressées.

On a donc assisté à l'émergence de deux types de compressions de données: la compression conservative et la compression non conservative.

La compression conservative consiste à compresser des données et à les décompresser sans perte de données, c'est ce type de compression qui est utilisée pour compresser du texte par exemple.

La compression d'image non conservative quant à elle s'autorise à ne pas conserver intacte les données compressées. C'est ce type de compression que nous allons étudier dans ce rapport au travers de la compression JPEG. On peut notamment souligner le fait qu'avec la compression non conservative de données (dans notre cas d'images) on obtient des taux de compression beaucoup plus importants qu'avec une compression conservative.

1.2 But du projet

Le but de ce projet consiste à:

- Implémenter l'algorithme de compression non conservative JPEG en paramètrant autant que possible.
- Réaliser une interface utilisateur agréable.
- Comparer le performances en terme de temps de codage, de temps de décodage, d'occupation mémoire, de taux de compression. Toutes ces comparaisons seront faites sur des fichiers images.

Lors de cette réalisation on essayera autant que possible de justifier le choix de structures ainsi que de procédure.

Chapitre 2
L'algorithme JPEG et son implémentation

2.1 Introduction

Le CCITT et l'ISO ont débuté dès les années 1970 leur recherche sur la compression de données. Ce n'est qu'en 1990 que le groupe JPEG "Joint Photographic Experts Group" créé par ces deux organisations produit les spécifications de ce qui va devenir la norme graphique JPEG.

L'algorithme de compression JPEG va suivre les trois étapes suivantes:

- La transformation en cosinus discrète (DCT)
- La quantification
- La compression conservative

Ce sont ces trois étapes que nous allons étudier dans les chapitres suivants.

2.2 La DCT

2.2.1 La formule

DCT veut dire en anglais Dicrete Cosinus Transformation ce qui est traduit en français par transformée de cosinus discrète.

Cette transformée est la clef de voûte de la compression JPEG. En effet cette transformée permet de transformer une courbe analogique ou digitale en sa représentation sous forme de fréquences. Il est important de noter que cette transformation est réversible (on parle de DCT inverse) et qu'elle est conservative (il n'y a pas de dégradation des données).

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

$$C(x) = \frac{1}{\sqrt{2}} \ si\ x = 0,\ et\ 1\ si\ x > 0$$

La transformée de cosinus discrète

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

$$C(x) = \frac{1}{\sqrt{2}} \ si\ x = 0,\ et\ 1\ si\ x > 0$$

La transformée de cosinus discrète inverse

2.2.2 Le choix du bloc

Lorsque l'on étudie l'algorithme on se rend compte que l'algorithme est en $O(N^2)$ ce qui rend impossible le calcul d'une image entière. Il a donc fallu décomposé l'image en blocs plus petits pour éviter l'explosion combinatoire. On a donc dû choisir la taille d'un bloc. La solution qui a été retenue est un bloc de 8x8, en effet avec un bloc plus petit le gain du taux de compression n'est pas optimal et avec un bloc plus grand on obtiendrai un plus grand taux mais avec une mauvaise cohérence entre les blocs.

2.2.3 Amélioration et implémentation

Une manière plus efficace de réaliser la DCT est d'utiliser les opérations de matrices. Pour effectuer cette opération il faut d'abord créer une matrice NxN, c'est la matrice de transformée de cosinus C. Une fois cette matrice construite il nous faudra aussi construire la matrice transposée pour que la formule devienne: DCT=C*Pixels*Ct.

$$C_{i,j} = \frac{1}{\sqrt{N}} \ si\ i = 0$$

$$C_{i,j} = \sqrt{\frac{2}{N}}\cos[\frac{(2j+1)i\pi}{2N}] \ si\ i > 0$$

Matrice de transformée de cosinus

Voici donc le code permettant de calculer C et Ct. Ce code se situe dans la fonction `init_C_Ct()`:

```
for (j=0;j<block;j++)
{
  *(C +0+j*block)=1.0/sqrt(block);
  *(Ct+j+0*block)=*(C +0+j*block);
}

for (i=1;i<block;i++)
```

```
  {
    for (j=0;j<block;j++)
    {
      *(C +i+j*block)=sqrt(2.0/block)*cos((2*j+1)*i*pi/(2.0*block));
      *(Ct+j+i*block)=*(C+i+j*block);
    }
  }
```

Pour conclure on peut écrire le calcul de la DCT de la manière suivante:

```
void DCT(int *DCT,unsigned char *pixel)
{ double tmp[block][block];
  double tmp1;
  int i,j,k;


  for(i=0;i<block;i++)
  {
    for (j=0;j<block;j++)
    {
      tmp[i][j]=0;
      for (k=0;k<block;k++)
      {
        tmp[i][j]+=(*(pixel+i+k*block)-128)**(Ct+k+j*block);
      }
    }
  }

  for (i=0;i<block;i++)
  {
    for (j=0;j<block;j++)
    {
      tmp1=0;
      for (k=0;k<block;k++)
      {
        tmp1+=*(C+i+k*block)*tmp[k][j];
      }
      *(DCT+i+j*block)=tmp1;
    }
  }
}
```

2.2.4 Sortie de DCT

Le point le plus important de la DCT est le résultat produit par celle-ci. En effet pour la matrice résultante le coefficient situé en (0,0) appelé "coefficient continu" représente une moyenne de la grandeur d'ensemble de la matrice d'entrée. Ce coefficient continu est plus grand d'un ordre de grandeur à toute autre valeur de la matrice de la DCT. Plus on s'éloigne du coefficient continu moins les valeurs apportent une contribution à l'image.

Donc cela signifie qu'en effectuant la DCT sur les données d'entrées, nous avons concentré la représentation de l'image dans les coefficients en haut à gauche de la matrice de sortie.

2.3 La quantification

La DCT a pris en entrée une matrice contenant des valeurs de 0 à 255 et a produit une matrice contenant des valeurs en -1024 et 1024. Donc on se rend compte qu'au lieu de gagner de l'espace on en a plutôt perdu. C'est la quantification qui va permettre d'obtenir de bons taux de compression.

La quantification est le processus qui va réduire le nombre de bits nécessaire au stockage en diminuant la précision de l'entier codé. Comme nous l'avons vu précédemment plus on s'éloigne du point (0,0) moins l'élément contribue à l'image. Donc plus on s'éloigne du point (0,0) moins la précision sur la valeur a d'importance. Nous allons donc utiliser une matrice de quantification pour compresser la sortie de la DCT. La matrice de quantification aura pour valeur le pas, c'est-à-dire la précision avec laquelle on va coder la valeur de la sortie de la DCT.

Les éléments les plus importants pour l'image seront codés avec une taille de pas petite, la taille 1 donnant la meilleure précision. Les valeurs deviennent donc de plus en plus grande au fur et à mesure que l'on s'éloigne de (0,0). On obtient donc la formule suivante:

$$\text{Valeur quantifiee}(i,j) = \frac{\text{DCT}(i,j)}{\text{Quantum}(i,j)}$$

On voit facilement que l'opération s'inverse facilement lors de la dé-quantification:

$$\text{DCT}(i,j) = \text{Valeur quantifiee}(i,j) * \text{Quantum}(i,j)$$

Bien sûr la variable `Valeur quantifiee` est une valeur entière, on perd donc un peut d'information mais on gagne en compression.

Dans notre implémentation nous avons pris une matrice qualité avec un algorithme très simple. Pour déterminer la valeur de la taille des pas de quantum l'utilisateur doit donner une valeur de "facteur de qualité" qui doit être dans l'intervalle de 1 à 25. Les valeurs au dessus de 25 seront acceptées mais l'image sera suffisamment dégradée pour ne pas dépasser cette valeur. Voici l'algorithme utilisé dans la procédure `init_matrice_qualite(int qualite)`:

```
for (i=0;i<block;i++)
  for (j=0;j<block;j++)
    *(quantum+i+j*block)=1+((1+i+j)*qualite);
```

Bien sûr cet algorithme constitue un choix important dans l'algorithme JPEG, il ne faut donc pas négliger cet algorithme.

Pas encore de Figure

Figure 2.1: Exemple de compression/décompression

2.4 La compression conservative

Une fois tout ceci effectué il ne nous reste plus qu'a coder les données ainsi produites. Nous allons voir que le reste du codage va s'effectuer en deux étapes:

- la séquence zigzag
- le codage RLE

2.4.1 La séquence zigzag

Comme nous l'avons vu précédemment grâce à la quantification un grand nombre de valeurs de la matrice se trouvent réduite à 0. Le fait qu'il y ait beaucoup de valeurs mises à 0 va nous permettre de compresser encore mieux grâce à la compression RLE que nous verrons plus tard.

Une manière d'augmenter la séquence de 0 (et donc de gagner en compression) est de modifier l'ordre de codage de la matrice. En effet si on utilise une séquence zigzag on augmente considérablement le nombre de 0 qui se suivent.

Voici le code utilisé pour générer le parcours zigzag:

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc,char *argv[])
{ int x=0,y=0;
  int max;

  max=atoi(argv[1])-1;

  for (x=0; x<=max; x++)
    for (y=0; y<=x; y++) {
      printf("(%d,%d)",x%2?x-y:y,x%2?y:x-y);
    }
  for (x=1; x<=max; x++)
    for (y=max; y>=x; y--) {
      printf("(%d,%d)",x%2?x+max-y:y,x%2?y:x+max-y);
    }
}
```

Le codage se fera donc grâce à la procédure suivante:

```
void code_zigzag(int *DCT,BIT_FILE *Fichier)
{ int i,x,y;
  double tmp;
```

```
  for (i=0;i<(block*block);i++)
  {
    x=zz[i].x;
    y=zz[i].y;
    tmp=*(DCT+x+y*block)/ *(quantum+y*block+x);
    encode(ROUND(tmp),(i==block*block-1),Fichier);
  }
}
```

Pas encore de Figure
Figure 2.2: Le parcours zigzag

2.4.2 Le codage RLE

Ce codage est basé sur le fait que l'on va coder une grande quantité de 0. Voici son principe:

- Tout d'abord on code sur 4 bits la taille en nombre de bits du prochain élément. Si c'est un 0 on code 0.
- Si on a précédemment codé un 0 alors on donne le nombre de 0 consécutifs. Ce nombre est codé sur 7 bits (0-127).
- Sinon on code sur 1 bit le signe de l'élément codé puis sa valeur absolue codé sur le nombre de bits donné précédemment.

Voici donc notre implémentation:

```
void encode(int code,int end,BIT_FILE *Fichier)
{ static int nb_zero=0;
  int abs_code,nb_bits=1;
  int signe,tmp;

  if (code==0)
  {
    nb_zero++;
    if (end)
    {
      fwrite_n_bit(0,nb_bit_size,Fichier);
      fwrite_n_bit(nb_zero,nb_bit_zero,Fichier);
      nb_zero=0;
    }
    return;
  }

  if (nb_zero!=0)
  {
    fwrite_n_bit(0,nb_bit_size,Fichier);
    fwrite_n_bit(nb_zero,nb_bit_zero,Fichier);
    nb_zero=0;
  }
```

```
  if (code<0)
  {
    abs_code=-code;
    signe=1;
  }
  else
  {
    abs_code=code;
    signe=0;
  }

  tmp=abs_code;

  while(tmp!=1)
  {
    nb_bits++;
    tmp/=2;
  }

  fwrite_n_bit(nb_bits,nb_bit_size,Fichier);
  fwrite_n_bit(signe,nb_bit_sign,Fichier);
  fwrite_n_bit(abs_code,nb_bits,Fichier);
}
```

Le décodage se fait de manière similaire.

Chapitre 3
L'interface utilisateur

3.1 Introduction

Pour pouvoir comparer efficacement la compression d'image JPEG il a fallu choisir un autre type de fichier image qui constituera notre référence. Notre choix c'est porté vers le le fichier BitMap (.bmp). En effet celui-ci est constitué d'une entête de longueur variable. Pour simplifier la tâche nous nous sommes volontairement limité aux fichiers couleurs et en niveaux de gris de taille 640x480. La taille 640x480 est une taille fréquemment utilisé c'est pourquoi nous l'avons choisi. Le format du fichier est le suivant:

```
 octet     fichier couleur            fichier niveaux de gris

     0     debut entete               debut entete
     .     .                          .
     .     .                          .
```

```
    53     fin entete                .
    54     debut donnees             .
     .     . un octet pour Rouge     .
     .     . un octet pour Vert      .
  1077     . un octet pour Bleu      fin entete
  1078     .                         debut donnees
     .     .                         . chaque octet represente
     .     .                         . un niveau de gris (0-255)
308278     .                         fin du fichier
     .     .
     .     .
921654     fin du fichier
```

3.2 Lecture et écriture de fichier BMP

La lecture du fichier BMP se fait donc de la manière suivante (fonction `lit_fichier_BMP(char *fich)`):

```
  for (i=0;i<taille_entete;i++) fgetc(in);
  for (y=0;y<480;y++)
  {
    for (x=0;x<640;x++)
    {
      ecran_r[x][y]=(unsigned char)fgetc(in);
      if (COLOR==1)
      {
        ecran_v[x][y]=(unsigned char)fgetc(in);
        ecran_b[x][y]=(unsigned char)fgetc(in);
      }
    }
  }
```

L'écriture se fait de manière analogue.

3.3 Lecture et écriture de fichier JPEG

Pour ce qui est du fichier JPEG que nous allons généré, nous avons déjà expliqué le codage dans le chapitre précédent. Il ne nous reste donc plus qu'à découper l'image en petits blocs de 8x8 et à effectuer l'encodage comme le montre le code de la fonction `ecrit_fichier_JPEG(char *fich)` :

```
  fwrite_n_bit(COLOR,1,Fichier);
  fwrite_n_bit(QUALITE,7,Fichier);
  fwrite_n_bit(block,8,Fichier);

  for (y=0;y<Y;y++)
  {
    for (x=0;x<X;x++)
```

```
    {
      for (yy=0;yy<block;yy++)
      {
        for (xx=0;xx<block;xx++)
        {
          *(tmp_r+xx+yy*block)=ecran_r[x*block+xx][y*block+yy];
          if (COLOR==1)
          {
            *(tmp_v+xx+yy*block)=ecran_v[x*block+xx][y*block+yy];
            *(tmp_b+xx+yy*block)=ecran_b[x*block+xx][y*block+yy];
          }
        }
      }
      DCT(dct,tmp_r);
      code_zigzag(dct,Fichier);
      if (COLOR==1)
      {
        DCT(dct,tmp_v);
        code_zigzag(dct,Fichier);
        DCT(dct,tmp_b);
        code_zigzag(dct,Fichier);
      }
    }
  }
  fwrite_bit_end(Fichier);
```

Comme précédemment la lecture se fait de fichier analogue.

3.4 Convertisseur BMP vers JPEG

Grâce aux procédures que nous avons décrite précédemment il est maintenant aisé de créer un programme convertissant un fichier BMP vers un fichier JPEG. Voici donc le code du programme `bmp2jpeg`:

```
int main(int argc,char *argv[])
{
  init(8);                            /* Initialisation des variables         */
  init_matrice_qualite(atoi(argv[3])); /* Initialisation de la matrice qualite */
  lit_fichier_BMP(argv[1]);           /* Lecture du fichier BMP               */
  ecrit_fichier_JPEG(argv[2]);        /* Ecriture du fichier JPEG             */
  return(0);
}
```

L'utilisation du convertisseur est très simple on passe comme arguments à la commande le fichier BMP source et le fichier JPEG destination ainsi que le facteur de qualité comme le montre l'exemple suivant:

```
bmp2jpeg Exemples/starwars.bmp Exemples/starwars.jpeg 4
```

3.5 Convertisseur JPEG vers BMP

Comme précédemment la conversion d'un fichier JPEG en fichier BMP se fait très facilement comme le montre le code du programme `jpeg2bmp`:

```
int main(int argc,char *argv[])
{
  init(8);                      /* Initialisation des variables */
  lit_fichier_JPEG(argv[1]);    /* Lecture du fichier JPEG      */
  ecrit_fichier_BMP(argv[2]);   /* Ecriture du fichier BMP      */
  return(0);
}
```

L'utilisation de ce convertisseur est très simple, il faut lui passer en arguments le fichier JPEG source ainsi que le fichier BMP destination comme le montre l'exemple qui suit:

```
jpeg2bmp Exemples/starwars.jpeg Exemples/starwars.bmp
```

3.6 L'afficheur

Notre projet serait incomplet si on ne pouvait pas visualiser nos fichier JPEG. C'est pourquoi nous avons aussi développé un programme permettant de visualiser les fichiers BMP et JPEG ainsi que pouvant dégrader une image et convertir une image BMP vers JPEG.

Comme la réalisation d'un afficheur d'image s'éloigne un peu de la compression nous ne décriront pas ici le code de ce programme. Nous pouvons simplement préciser que cet afficheur a été réalisé en Xlib et qu'il n'affiche que les images en niveaux de gris.

Pour exécuter l'afficheur il suffit de lui passer en argument le nom du fichier BMP ou JPEG a afficher:

```
viewer Exemple/starwars.bmp
```

Un fois l'afficheur lancé on peut effectuer les action suivantes:

- Quitter (touche `q`)
- Recharger le fichier (touche `r`)
- Sauvegarder le fichier en JPEG (touche `s`)
- Dégrader l'image (touche `0-9`)

Figure 3.1: L'afficheur

3.7 L'interface graphique

Les programmes précédents n'ayant pas tous une interface graphique, nous avons développé une interface graphique simple pour tous les utilisateurs ne souhaitant pas taper de lignes de commande. Cette interface a été réalisée en TCL/TK à l'aide de Visual TCL version 1.2.

Cette interface permet de réaliser toutes les opérations décrites précédemment le tout à l'aide d'une interface simple d'utilisation. En voici une rapide description:

- Deux boutons `...` un pour chaque type de fichier BMP et JPEG. Ces bouton permettent d'ouvrir une boîte de dialogue permettant de sélectionner un fichier.
- Deux boutons `visualiser` un pour chaque type de fichier. Ces boutons permettent de d'afficher le fichier BMP ou JPEG correspondant.
- Un bouton permettant de convertir un fichier BMP en fichier JPEG.
- Un bouton permettant de convertir un fichier JPEG en fichier BMP.
- Une scalebar permettant de choisir le coefficient de dégradation.
- Un bouton `Quitter`.

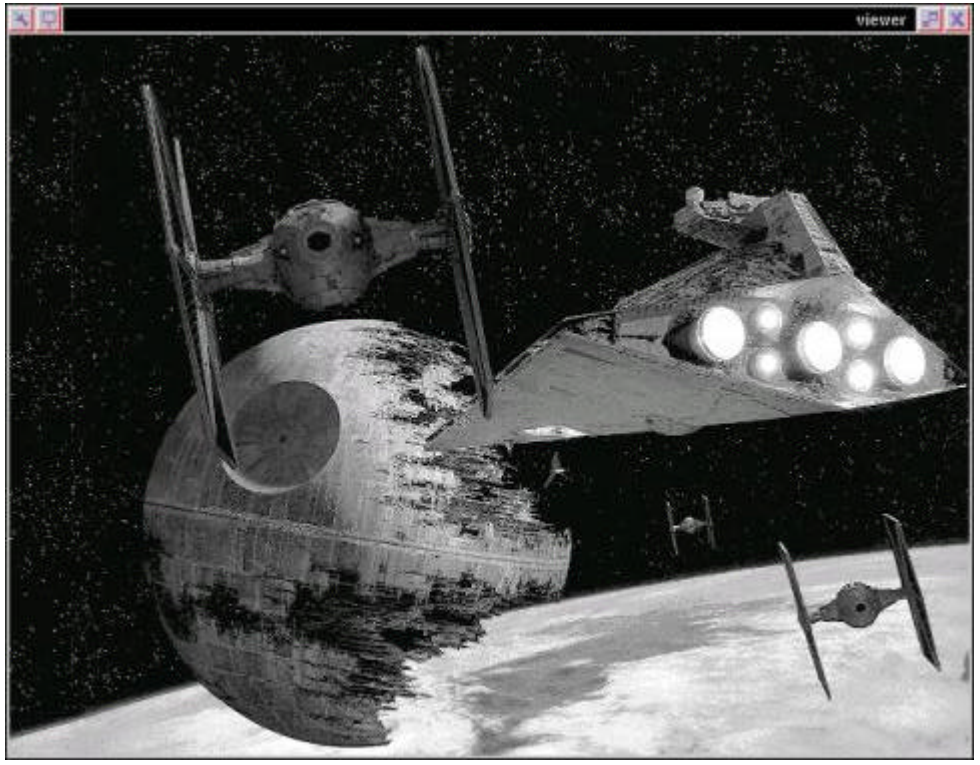

Figure 3.2: L'interface graphique

Chapitre 4
Performances

4.1 L'occupation mémoire

On constate tout d'abord que l'occupation mémoire est la même lors de la compression et lors de la décompression. On note aussi que la mémoire ne varie pas en fonction de l'image ce qui est normal car nous travaillons sur des images de même taille. Par contre une image couleur prendra deux fois plus de place en mémoire qu'une image en niveau de gris. Voici les résultats obtenus:

```
  PID USER     PRI  NI  SIZE  RSS SHARE STAT %MEM   COMMAND
 1346 seb       16   0  1304 1304   340 R     2.0   bmp2jpeg
 1370 seb       13   0  1300 1300   336 R     2.0   jpeg2bmp
```

Pour la couleur

```
  PID USER     PRI  NI  SIZE  RSS SHARE STAT %MEM   COMMAND
 1351 seb       20   0   712  712   340 R     1.1   bmp2jpeg
 1360 seb       20   0   708  708   336 R     1.1   jpeg2bmp
```

Pour les niveaux de gris

4.2 Temps de compression/décompression

Nous avons constaté lors de nos tests que le temps de compression et le temps de décompression ne variait pas en fonction des images, ce qui nous assure donc un temps de compression/décompression constant. Les résultats que nous allons maintenant produire dépendent beaucoup du type de machine utilisée, ce qui est important c'est le rapport entre les différents résultats.

Voici les résultats que nous avons obtenus:

- Pour la compression:
  - 1,6 secondes pour les niveau de gris
  - 4,8 secondes pour couleur

- Pour la décompression
  - 1,7 secondes pour les niveaux de gris
  - 5,2 secondes pour couleur

On peut noter que la décompression est un peut plus lente que la compression. On remarque aussi qu'en moyenne le temps de compression/décompression d'une image couleur est 3 fois plus grand que la compression/décompression de la même image en niveaux de gris ce qui s'explique par le fait qu'une image couleur est 3 fois plus importante en terme de taille que la même image en niveaux de gris.

4.3 Taux de compression

Lors de nos test on obtient le même taux de compression avec la même image en couleur et en niveaux de gris.

Ci dessus nous avons présenté sous forme d'un graphe les résultats expérimentaux que nous avons obtenus.

Pas encore de Figure
Figure 4.1: Taux de compression en fonction de l'indice de dégradation

4.4 Différences visuelles

Sur la page suivante nous montrons la dégradation de l'image lors d'une forte compression JPEG.

Figure 4.2: Une image BMP comparée à une image JPEG (fort indice de dégradation)

Annexe A
Les sources

Cliquer ici pour enregistrer les sources (25 ko)

Retour au menu précédent...