Compression d'une image couleur par l'algorithme des quadtrees

Jimmy Levacher (18901505)

29 Décembre 2021

1 Introduction

Ce projet consiste à encoder, décoder et compresser avec ou sans perte des images en utilisant des quadtrees. Les quadtrees consistent à découper récursivement une image par zones de quatre. Ainsi, une image sera représentée par un arbre, chaque feuille représente un pixel de l'image et donc, un noeud une zone de l'image.

2 Fonctionnement du quadtree

Dans ce projet, j'ai choisi d'implémenter la courbe de Peano en Z, c'est-à-dire, parcourir les zones découpées dans un ordre précis comme le montre l'image ci-dessous.

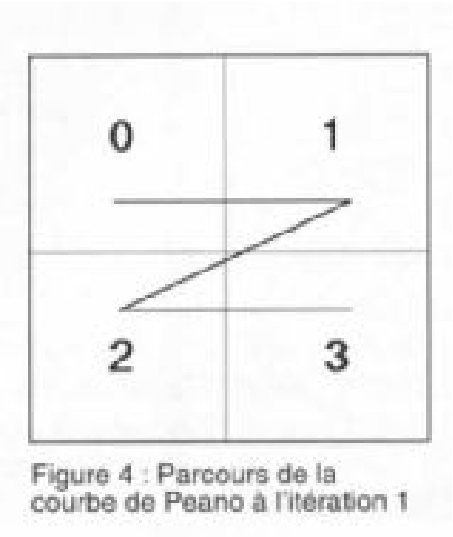


FIGURE 1 – Parcours d'une zone

L'arbre quadtree d'une image sera représenté comme ceci

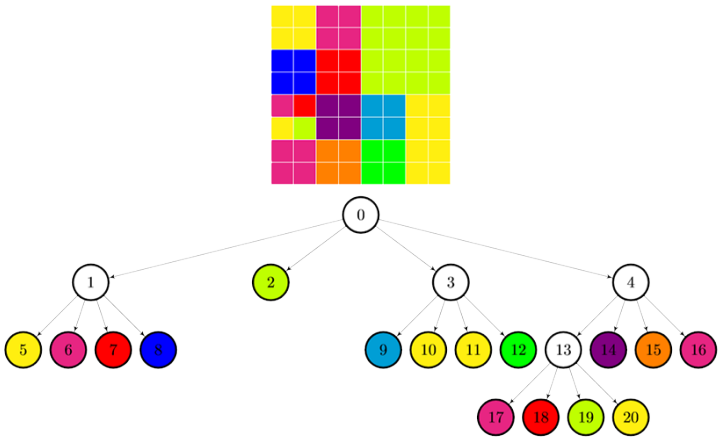
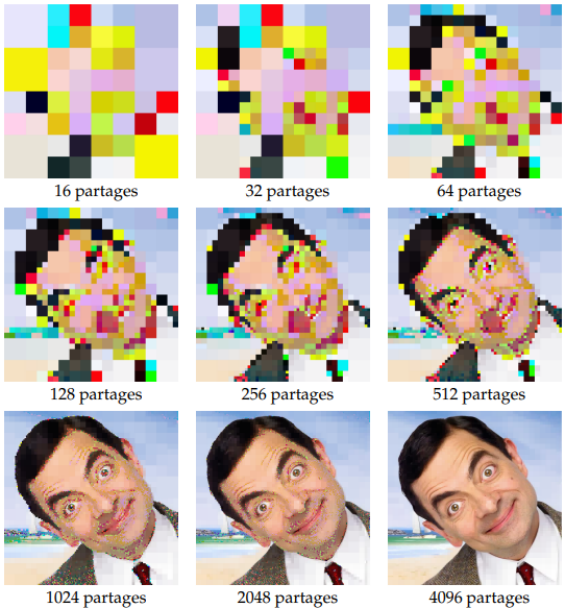


FIGURE 2 – Arbre d'une image

Le parcours de la courbe de Peano est répété pour toutes les récursions de toutes les zones de l'image jusqu'à son terme, le pixel. Durant ce parcours, on cherche la zone la plus loin de la réalité en calculant la somme des erreurs par rapport à la réalité sur chaque zone.



3 Implémentation du quadtree

Pour ce faire, l'arbre est représenté comme ceci

```
 // ___________________________
 // |            |            |
 // |            |            |
 // |     NO     |     NE     |
 // |            |            |
 // |____________|____________|
 // |            |            |
 // |            |            |
 // |     SO     |     SE     |
 // |            |            |
 // |____________|____________|
typedef struct Quadtree
{
    Pixel color;

    struct Quadtree *NO, *NE;
    struct Quadtree *SO, *SE;
 } *Quadtree;
```

FIGURE 3 – Structure du quadtree

Et un pixel comme ceci

```
typedef struct pixel
{
    unsigned char r;
    unsigned char g;
    unsigned char b;
} Pixel;
```

FIGURE 4 – Structure d'un pixel

En effet, le type `unsigned char` est capable de représenter les nombres de `[0;255]`, comme une couleur `RGB`, d'où ce choix.

Une image, quant à elle, est réprésentéé comme ceci

```
typedef struct image
{
    int w, h;
    Pixel **data;
} Image;
```

FIGURE 5 – Structure d'une image

Les données de l'image, c'est-à-dire ses pixels, seront stockées dans une matrice de pixels, `data`.

4 Lecture d'une image

Pour ce projet, j'ai décidé de traiter uniquement les images de type `ppm` car vu en cours de `Programmation Graphique`.

```
Image *read(char * path)
{
    [...]

    img = (Image *)malloc(sizeof(Image)); assert(img);

    /* PPM Header */
    fscanf(f, "%s"   , magicNum);    /*MagicNum*/
    fscanf(f, "%d "  , &img->w);     /*Largeur*/
    fscanf(f, "%d\n" , &img->h);     /*Hauteur*/
    fscanf(f, "%d"   , &color);        /*Color*/

    while (fgetc(f) != '\n'); //données inutiles

    img->data = (Pixel **)malloc(img->w * img->h * sizeof(Pixel));
    assert(img->data);

    for(int i = 0; i < img->h; i++)
    {
        img->data[i] = malloc(img->w * sizeof(Pixel));
        assert(img->data[i]);
    }

    for(int i = 0; i < img->h; i++)
        for(int j = 0; j < img->w; j++)
        {
            fread(&img->data[i][j].r, sizeof(unsigned char), 1, f);
            fread(&img->data[i][j].g, sizeof(unsigned char), 1, f);
            fread(&img->data[i][j].b, sizeof(unsigned char), 1, f);
        }

    fclose(f);

    return img;
}
```

FIGURE 6 – Lecture d'une image ppm

5 Écriture d'une image

```c
void write(char * path, Image * img)
{
    FILE *f;
    f = fopen(path, "wb");
    FILEcheck(f);

    /* PPM Header */
    fprintf(f, "P6\n");                                  /*MagicNum*/
    fprintf(f, "%d %d\n", img->w, img->h);              /*Size*/
    fprintf(f, "%d\n", 255);                           /*Color*/

    for(int i = 0; i < img->h; i++)
        for(int j = 0; j < img->w; j++)
        {
            fwrite(&img->data[i][j].r, sizeof(unsigned char), 1, f);
            fwrite(&img->data[i][j].g, sizeof(unsigned char), 1, f);
            fwrite(&img->data[i][j].b, sizeof(unsigned char), 1, f);
        }

    fclose(f);
}
```

FIGURE 7 – Écriture d'une image ppm

6 Découpage d'une image

Comme dit plus haut, l'image doit être découpée en quatre zones (NO/NE/SO/SE) de manière récursive. En cours de `Programmation Graphique` nous avons pu voir que pour faire une recherche en profondeur dans un arbre par exemple, la récursivité sur chaque branche était idéale.
C'est donc cette méthode qui a été implémentée.

```
1  decoupage(img, &(*noeud)->NO, getCoord(NO, c.x, c.y, h, w), ...);
2  decoupage(img, &(*noeud)->NE, getCoord(NE, c.x, c.y, h, w), ...);
3  decoupage(img, &(*noeud)->SO, getCoord(SO, c.x, c.y, h, w), ...);
4  decoupage(img, &(*noeud)->SE, getCoord(SE, c.x, c.y, h, w), ...);
```

FIGURE 8 – Découpage d'une image

A chaque itération, la taille de l'image est divisée par 2. Il faut donc indiquer des coordonnées `x,y` afin de savoir dans quelle partie d'une zone nous nous situons.

```
1  Coord getCoord(Direction d, int x, int y, int w, int h)
2  {
3      Coord c;
4      switch (d)
5      {
6      case NO:
7          c.x = x;
8          c.y = y;
9          return c;
10     case NE:
11         c.x = x + (w / 2);
12         c.y = y;
13         return c;
14     case SO:
15         c.x = x;
16         c.y = y + (h / 2);
17         return c;
18     case SE:
19         c.x = x + (w / 2);
20         c.y = y + (h / 2);
21         return c;
22     default:
23         c.x = c.y = 0;
24         return c;
25     }
26 }
```

FIGURE 9 – Retourne les coordonnées x,y

7 Couleur moyenne

Afin de calculer la somme des erreurs par rapport à la réalité sur chaque zone, il faut déterminer la moyenne RGB de chaque zone. Il suffit de parcourir la zone actuelle dans la matrice et de faire une simple moyenne sur chaque couleur.

```
Pixel moyenneRGB(Image * img, Coord c, int h, int w)
{
    Pixel p;
     unsigned long long r = 0, g = 0, b = 0;

    /* Parcours de la matrice */
    for(int i = c.y; i < c.y + h; i++)
        for(int j = c.x; j < c.x + w; j++)
        {
            r += img->data[i][j].r;
            g += img->data[i][j].g;
            b += img->data[i][j].b;
        }

    p.r = r / (h * w);
    p.g = g / (h * w);
    p.b = b / (h * w);

    return p;
}
```

Une fois la moyenne obtenue, elle est copiée dans l'arbre.

```
void ccRgb2tree(Quadtree *noeud, Pixel color)
{
    (*noeud)->color.r = color.r;
    (*noeud)->color.g = color.g;
    (*noeud)->color.b = color.b;
}
```

8 Calculs des erreurs

Pour calculer la somme des erreurs dans une zone afin d'avoir un rendu correct, j'ai fais face à deux méthodes.

$$mean = \frac{\sum_{i=x}^{x+size}\sum_{j=y}^{y+size}(red - grid[i][j].red)^2 + (green - grid[i][j].green)^2 + (blue - grid[i][j].blue)^2}{3 \cdot size \cdot size}$$

FIGURE 10 – Première méthode

$$dist(p_1, p_2) = \sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2 + (a_1 - a_2)^2}$$

FIGURE 11 – Deuxième méthode

La différence se porte sur la division par la taille de l'image.
En effet, avec la deuxième méthode la précision est mieux mais la plage de "réglage" dans le terminal est plus grande.
J'ai finalement opté pour la seconde méthode.

```c
double somErrorQuadtree(Image * img, Pixel mRGB, Coord c, int h, int w)
{
    double d = 0;

    /* Parcours de la matrice */
    for(int i = c.y; i < c.y + h; i++)
        for(int j = c.x; j < c.x + w; j++)
            d += dist(mRGB, img->data[i][j]);

    return d;
}
```

```c
double dist(Pixel p1, Pixel p2)
{
    return sqrt(pow(p1.r - p2.r, 2) + pow(p1.g - p2.g, 2) + pow(p1.b -
    ↪  p2.b, 2));
}
```

9 Remplissage de la matrice

Une fois l'arbre construit, il faut remplir la matrice avec ses valeurs.
Il faut donc parcourir l'arbre récursivement comme la méthode de découpage, et lorsqu'il s'agit d'une feuille, donc d'un pixel, on le met dans la matrice.

```
1  void remplissage(Quadtree noeud, Image *img, Coord c, int h, int w)
2  {
3      if(!feuille(noeud))
4      {
5          remplissage(noeud->NO, img, getCoord(NO, c.x, c.y, h, w), ...);
6          remplissage(noeud->NE, img, getCoord(NE, c.x, c.y, h, w), ...);
7          remplissage(noeud->SO, img, getCoord(SO, c.x, c.y, h, w), ...);
8          remplissage(noeud->SE, img, getCoord(SE, c.x, c.y, h, w), ...);
9      }
10     else ccTree2matrix(noeud, img, c, h, w);
11 }
```

```
1  void ccTree2matrix(Quadtree noeud, Image *img, Coord c, int h, int w)
2  {
3      for(int i = c.y; i < c.y + h; i++)
4          for(int j = c.x; j < c.x + w; j++)
5          {
6              img->data[i][j].r = noeud->color.r;
7              img->data[i][j].g = noeud->color.g;
8              img->data[i][j].b = noeud->color.b;
9          }
10 }
```

```
1  int feuille(Quadtree noeud)
2  {
3      return (noeud->NO == NULL && noeud->NE == NULL && noeud->SO == NULL
       ↪  && noeud->SE == NULL);
4  }
```

10 Utilisation

Pour compiler il suffit d'ouvrir un terminal dans le dossier du projet et d'exécuter la commande `make`, puis `./quadtree 0 test/test5.ppm out.ppm`
Le premier argument est le niveau d'erreur, plus il est haut plus l'image est compressée. Le second, l'image d'entrée, et le dernier le chemin de sortie.

FIGURE 12 – Taux 5000

```
lecture...ok
decoupage...ok
remplissage...ok
ecriture...ok
clear...ok
Temps : 2534 ms
```

FIGURE 13 – Sortie terminal

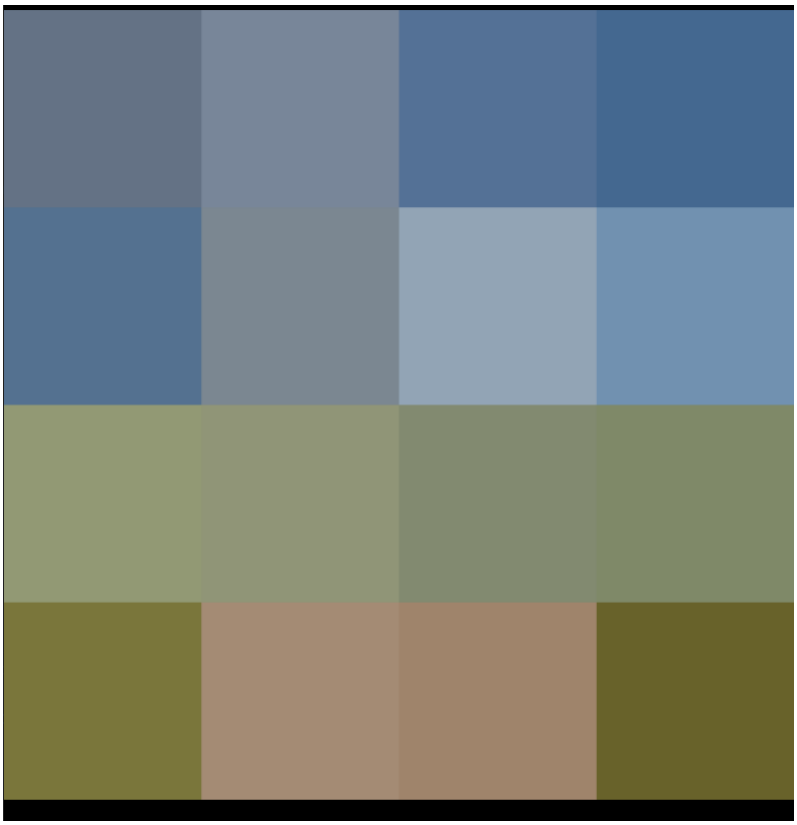


FIGURE 14 – Taux 1000000

11 Problèmes rencontrés

Malgré l'aspect visuel fonctionnel, je n'ai pas réussi à réduire la taille de sortie de l'image.

12 Documentations

Pour réaliser ce projet, je me suis beaucoup aidé de ce `PDF` de l'Université de Marne la Vallée, ce dernier explique concrètement comment fonctionne le quadtree mais également certaines méthodes de calculs que j'ai repris.
Mais également de cette `page`

Plusieurs ancien projets m'ont également aidé, comme `Fortress` en L2 ainsi que divers TP en `Programmation Graphique`.