

Menguelti Mounir

---

**Projet 14 Algorithmique Avancée 2020-2021**  
**Compression Par simplification.**

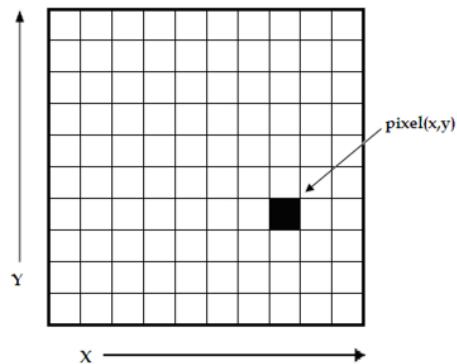
## Table des matières

<b>1</b>	<b>Parcours par fenêtre</b>	<b>2</b>
1.1	Structure d'une image : . . . . .	2
1.2	Parcours fenêtre par fenêtre : . . . . .	2
<b>2</b>	<b>Compression par simplification :</b>	<b>4</b>
2.1	Compression : . . . . .	4
2.2	Taux de compression : . . . . .	5
2.3	Décompression et affichage : . . . . .	5
2.4	Comparaison d'images : . . . . .	6

# 1 Parcours par fenêtre

## 1.1 Structure d'une image :

-Une image est représentée par une grille de plusieurs pixels.

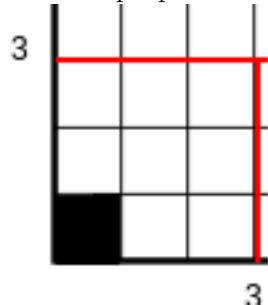


-Chaque pixel est définie par trois valeurs(R, G, B).

193	194	190
-----	-----	-----

## 1.2 Parcours fenêtre par fenêtre :

-Pour parcourir l'image par fenêtre, l'image est découpé en fenêtre (3\*3) ou (4\*4) puis je parcours chaque pixel de la fenêtre :



```
int tab3[] [2] =
{{0, 0},{1, 0},{2, 0},
{0, 1},{1, 1},{2, 1},
{0, 2},{1, 2},{2, 2}};

int tab4[] [2] =
{{0, 0},{1, 0},{2, 0},{3, 0},
{0, 1},{1, 1},{2, 1},{3, 1},
{0, 2},{1, 2},{2, 2},{3, 2},
{0, 3},{1, 3},{2, 3},{3, 3}};
```

-Pour parcourir tout les pixels de la fenêtre j'utilise un tableau d'indice qui permet de parcourir l'entièreté de la fenêtre.

-Pour tester cette méthode je parcours une image où je recharge et j'affiche l'image par la suite pour vérifier que la méthode ne modifie pas les valeurs des pixels.

```
struct pixel_t{
    GLubyte r,g,b;
};

typedef struct pixel_t pixel_t;

pixel_t * pim = (pixel_t*) im;
pixel_t * tmp = (pixel_t*) ptmp;

for(y= 0; y < image->sizeY; y+=3)
{
    for(x = 0; x < image->sizeX ; x+=3)
    {
        for(j = 0 ; j < 9; ++j)
        {

            px = x + tab3[j] [0];
            py = y + tab3[j] [1];

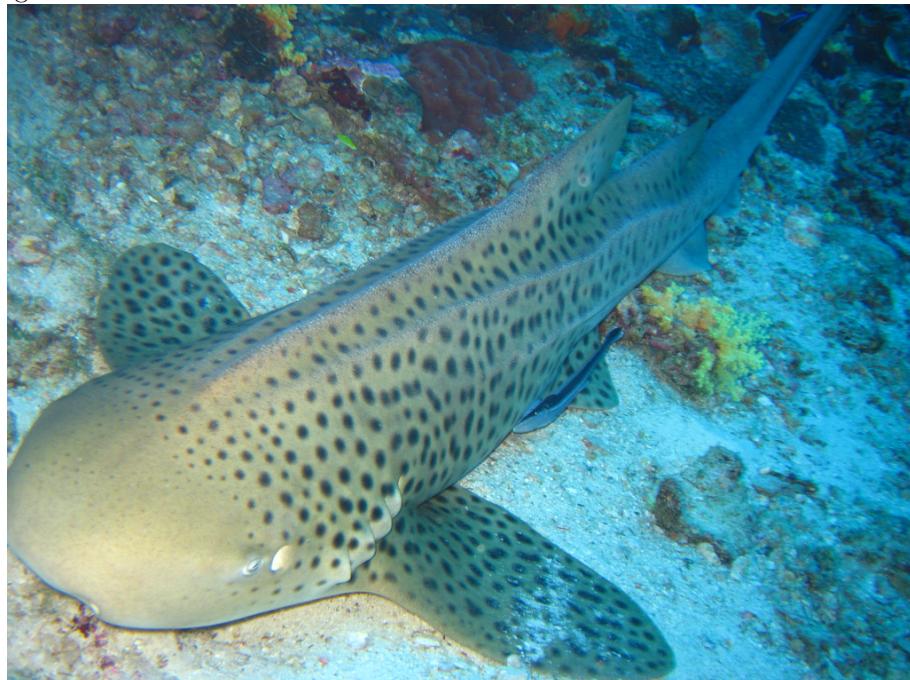
            i = py *image->sizeX + px;

            tmp[i].r=pim[py *image->sizeX+px].r;
            tmp[i].g=pim[py *image->sizeX+px].g;
            tmp[i].b=pim[py *image->sizeX+px].b;
        }
    }
}

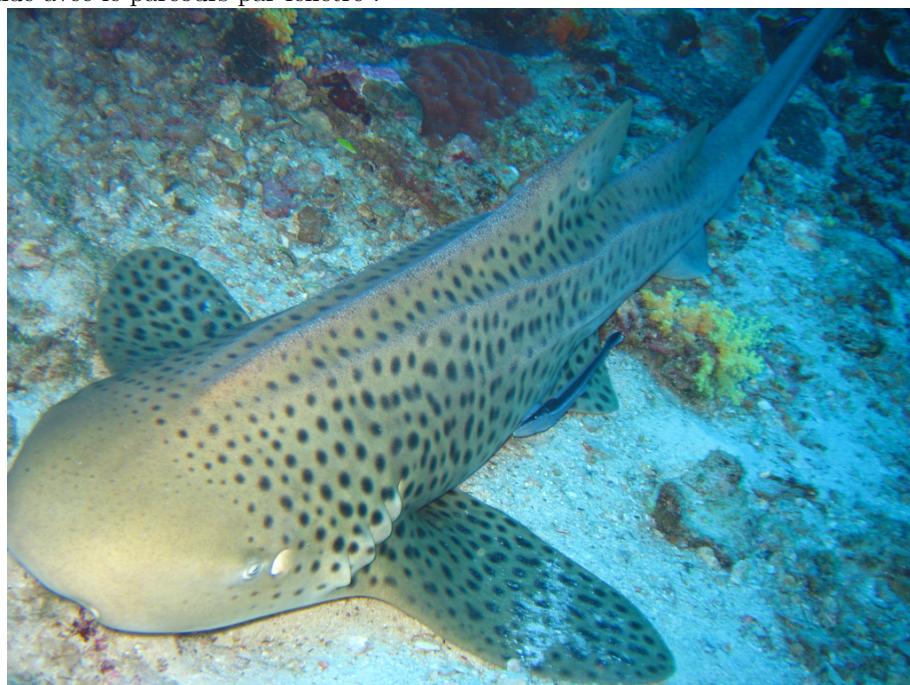
image->data=tmp;
```

-Les valeurs(R,G,B) de chaque pixel de la nouvelle image(tmp) reçoit la valeur du pixel de l'image original(pim).

-Image originale :



-Image obtenue avec le parcours par fenêtre :



## 2 Compression par simplification :

Pour toute fenêtre ( $n \times n$ ) de l'image, on trouve une valeur minimale de chacune des couleurs. Puis pour chaque pixel de la fenêtre, la couleur est donnée par un écart sous la forme de quelques chiffres binaires avec la couleur minimale. Par exemple 3 bits pour le vert, 3 pour le rouge et 2 pour le bleu. On a donc, par fenêtre, une couleur puis uniquement un octet pour chaque pixel.

### 2.1 Compression :

-Pour chaque fenêtre parcourue, je récupère le minimum de chaque couleur (R, G, B), qui sera par la suite ajouté à un tableau.

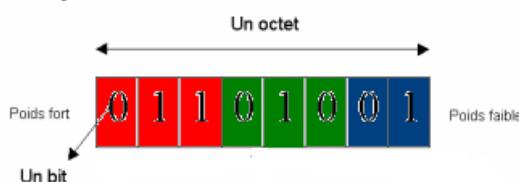
```
int tabRGB[cpt][3];

for (y= 0; y < image->sizeY ; y+=3)
{
    for(x = 0; x < image->sizeX ; x+=3)
    {
        i = y *image->sizeX + x;
        for(j = 0 ; j < 9; ++j)
        {
            px = x + tab[j][0];
            py = y + tab[j][1];
            if(j == 0)
            {
                rmin=pim[py *image->sizeX+px].r;
                gmin=pim[py *image->sizeX+px].g;
                bmin=pim[py *image->sizeX+px].b;
            }
            r=pim[py *image->sizeX+px].r;
            g=pim[py *image->sizeX+px].g;
            b=pim[py *image->sizeX+px].b;

            if(rmin>r){rmin = r;}
            if(gmin>g){gmin = g;}
            if(bmin>b){bmin = b;}
        }
        tabRGB[cpt][1] = rmin;
        tabRGB[cpt][2] = gmin;
        tabRGB[cpt][3] = bmin;
    }
}
```

-Ici cpt fait référence à l'indice de la fenêtre.

-Pour la suite chaque pixel doit être représenté par un octet où la couleur est calculé par une différence entre la couleur du pixel et les couleurs rmin gmin bmin.



B	R,G
00	000
01	001
10	010
11	011
XX	100
XX	101
XX	110
XX	111

-Les valeurs possibles pour R et G sont 8 et 4 pour B.

-Pour ne pas manipuler l'octet avec des décalages, un tableau contenant les valeurs possibles pour chaque couleur sera utilisé pour définir l'état des bits correspondant à chaque couleur.

```
unsigned char tabr[] =
{0x20,0x40,0x60,0x80,0xa0,0xc0,0xe0};
```

```
unsigned char tabg[] =
{0x04,0x08,0x0c,0x10,0x14,0x18,0x1c};
```

```
unsigned char tabb[] =
{0x00,0x01,0x02,0x03};
```

-Le but est de pouvoir changer la valeur des bits de chaque couleur selon la valeur de la différence entre R, G et B du pixel et rmin, gmin et bmin.

```
for(j = 0 ; j < 9; ++j)
{
    c = 0x00;
    px = x + tab[j][0];
    py = y + tab[j][1];

    r=pim[py *image->sizeX+px].r-rmin;
    g=pim[py *image->sizeX+px].g-gmin;
    b=pim[py *image->sizeX+px].b-bmin;

    c= print_byte_r(r,c);
    c= print_byte_g(g,c);
    c= print_byte_b(b,c);
    fprintf(img,"%c",c);
    tabPx[cpt][j] = c;
}
```

-La fonction :

```
unsigned char print_byte_r(int val, unsigned char c)
```

-Prend en paramètre la différence entre R du pixel et Rmin de la fenêtre, par la suite elle

change la valeur des trois bits de la couleur rouge avec un ou exclusif binaire et renvoie l'octet.

```
unsigned char print_byte_r(int val,
    unsigned char c)
unsigned char tabr[] =
{0x20,0x40,0x60,0x80,0xa0,0xc0,0xe0};

if (val<32){
    c = c | tabr[0];}

if (val>=32 && val<64){
    c = c | tabr[1];}

if (val>=64 && val<96){
    c = c | tabr[2];}

if (val>=96 && val<128){
    c = c | tabr[3];}

if (val>=128 && val<160){
    c = c | tabr[4];}

if (val>=160 && val<192){
    c = c | tabr[5];}

if (val>=192 && val<224){
    c = c | tabr[6];}

if (val>=224 && val<256){
    c = c | tabr[7];}

return c;
}
```

-Pour la couleur vert ça sera le même procédé juste que la tableau sera :

```
unsigned char tabg[] =
{0x04,0x08,0x0c,0x10,0x14,0x18,0x1c};
```

-Pour le bleu comme on a que 2 bits donc moins de possibilité :

```
unsigned char print_byte_b(int val,
    unsigned char c) {
unsigned char tabb[]={0x00,0x01,0x02,0x03};

if (val<64){
    c = c | tabb[0];}

if (val>=64 && val<128){
    c = c | tabb[1];}

if (val>=128 && val<192){
    c = c | tabb[2];}
```

```
if (val>=192 && val<256){
    c = c | tabb[3];}
return c;
}
```

-Chaque octet sera ajouté à un tableau d'octet à l'indice de chaque fenêtre :

```
unsigned char tabPx[cpt][9];
```

## 2.2 Taux de compression :

-Les données obtenu à partir de la compression seront sauvegarder dans un fichier texte pour pouvoir comparer la taille initiale de l'image et la taille des données obtenue à l'aide de la compression avec la formule  $\frac{\text{Taille,finale}}{\text{Taille,initiale}}$ .

-Pour un parcours (3\*3) le taux est de  $\simeq 62\%$   
-Pour un parcours (4\*4) le taux est de  $\simeq 50\%$

## 2.3 Décompression et affichage :

-À partir des deux tableaux qui contiennent les valeurs rmin, gmin et bmin de chaque et fenêtre, aussi les octets correspondant aux pixels de chaque fenêtre on pourra décompresser l'image et l'afficher :

```
int tabRGB[cpt][3];
unsigned char tabPx[cpt][9];

for (y= 0; y < image->sizeY; y+=9)
{
    for(x = 0; x < image->sizeX; x+=9)
    {

        rmin= tabRGB[cpt][1];
        gmin= tabRGB[cpt][2];
        bmin= tabRGB[cpt][3];

        for(j = 0 ; j < 9; ++j)
        {
            px = x + tab[j][0];
            py = y + tab[j][1];

            i = py *image->sizeX + px;

            rdiff = 0xe0 & tabPx[cpt][j] ;
            gdiff = 0x1c & tabPx[cpt][j] ;
            bdiff = 0x03 & tabPx[cpt][j] ;

            r = byte_to_intR(rdiff) +rmin;
            if(r>255){r = 255;}
```

```

g = byte_to_intG(gdiff) +gmin;
if(g>255){g = 255;}
b = byte_to_intB(bdiff) +bmin;
if(b>255){b = 255;}

tmp[i].r = (GLubyte)r;
tmp[i].g = (GLubyte)g;
tmp[i].b = (GLubyte)b;

}
cpt++;
}
}

image->data=ptmp;

```

-Pour récupérer les valeurs des bits de chaque couleurs et retrouver la valeur initiale de la couleur, avec un et binaire entre l'octet et la valeur(0xe0 == 11100000) les bits du vert et du bleu sont mis à zéro pareil pour le vert avec la valeur(0x1c == 00011100) et le bleu (0x03 == 00000011).

-La fonction :

```
int byte_to_intR(unsigned char c)
```

-Prend en paramètre l'octet du pixel pour renvoyé un entier qui correspond à une couleur selon l'état des bits.

```

int byte_to_intR(unsigned char c)
{
int val =0 ;
unsigned char tabr[] =
{0x20,0x40,0x60,0x80,0xa0,0xc0,0xe0};

if (c<tabr[0]){
    val = 0;}

if (c>=tabr[0] && c<tabr[1]){
    val = 32;}

if (c>=tabr[1] && c<tabr[2]){
    val = 64;}

if (c>=tabr[2] && c<tabr[3]){
    val = 96;}

if (c>=tabr[3] && c<tabr[4]){
    val = 128;}

if (c>=tabr[4] && c<tabr[5]){
    val = 160;}

if (c>=tabr[5] && c<tabr[6]){
    val = 192;}

```

```

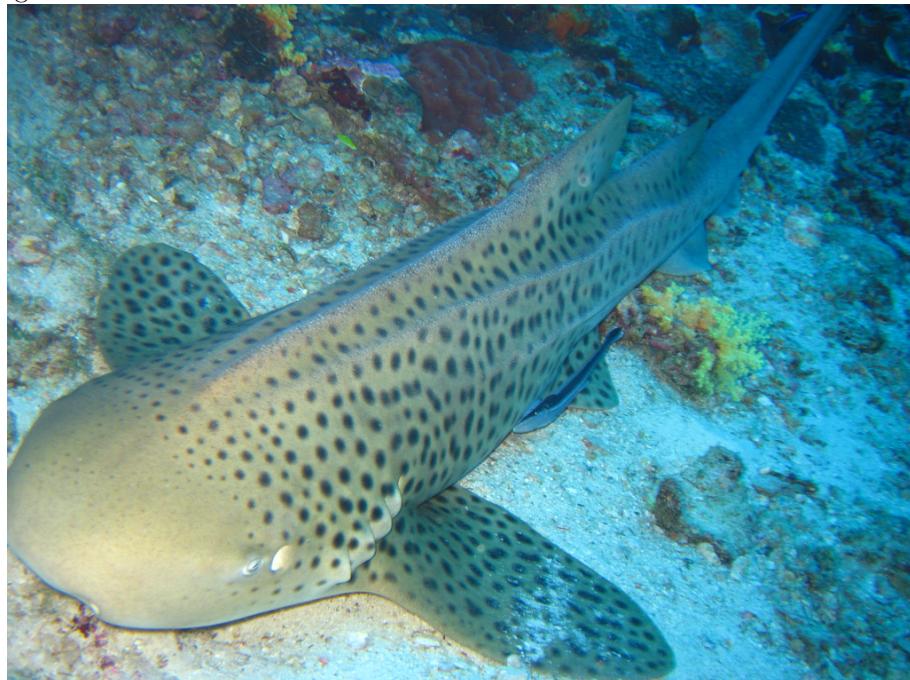
if (c>=tabr[6] && c<tabr[7]){
    val = 224;}

return val;
}
```

## 2.4 Comparaison d'images :

-On remarque une perte de détails et une couleur plus froide.

-Image originale :



-Image après décompression :

