Rapport du projet 2 : ULBMP

LI Min-Tchun

12/05/2024

Contents

1 Introduction

Pour le deuxième projet du cours INFO-F-106, nous avons dû implémenter un outil pour encoder ou decoder un fichier de format ULBMP, basé sur le format BMP, développé par Microsoft dans les années 80.

L'un des objectifs de ce projet est de pouvoir compresser une image. En effet, ce n'est pas efficace de stocker une image de manière brut car elle prendrait trop de place dans le stockage. Prenons l'exemple d'une image de dimension 360×480. Si chaque pixel est codé sur 3 octets (un octet pour chaque couleur), cela signifie que l'image prendrait $360 \times 480 \times 3 = 518400$ octets, soit 518.4 kilo-octets. Nous pourrions nous poser la question suivante : comment pouvons-nous réduire la taille de l'image tout en gardant une qualité acceptable ?

Afin de résoudre ce problème, on va procéder comme suit, nous allons créer une classe *Decoder* et *Encoder*. Le premier permettra, à partir d'un chemain d'accès du fichier, de lire chaque octets et de retourner une *image* (nous définierons plus tard plus précisemment ce que c'est). Le deuxième permettra, à partir d'une *image*, d'écrire dans un fichier les octets correspondants aux valeurs RGB de chaque pixel.

Par ailleurs, il est fondamental de maitriser la manipulation de fichier, de bits (notamment le shifting et le masking), ainsi que la compréhension de la représentation binaire et hexadécimal. En effet, comme nous devons utiliser des pixels, il est important de comprendre que ces derniers sont composées de 3 valeurs : Rouge, Vert, Bleu (RVB ou RGB en anglais). Chacune de ces valeurs est codée sur 8 bits, ce qui signifie que chaque valeur peut prendre $2^8 = 256$ valeurs différentes. De plus, il faut savoir que les octets du fichiers ULBMP sont encodés en little-endian, cela signifie que les bits de poids faible se trouve au début de la séquence de bits. Il était donc utile d'utiliser le programme `hexdump` afin de pouvoir visualiser les données hexadécimale.

2 Méthodes

Comme cité dans l'introduction, nous allons devoir implémenter deux classes : *Decoder* et *Encoder*. Mais avant ça, il va falloir définir les classes *Pixel* et *Image* car elles seront utiles pour la représentation des pixels et des images en Python.

2.1 Pixel et Image

La classe *Pixel* possède les attributs suivants : r, g, b (ces derniers correspondent aux valeurs RGB d'un pixel). Le constructeur *Image* prend en paramètre la hauteur, la largeur de l'image, ainsi qu'une liste de taille $hauteur \times largeur$ comprenant des instances de la classe Pixel. Par ailleurs, rappelons nous que chaque couleur du pixel est encodée sur 8 bits, ce qui signifie qu'il est impossible d'avoir des valeurs négatives ou supérieures à 255. De plus, une image ne peut pas posséder une quantité de pixel supérieure à la dimension de l'image. Par exemple, si une image a une hauteur de 5 pixels et une largeur de 3 pixels, il est impossible d'accéder au pixel d'indice (6, 2) car il n'existe pas.

Voilà pourquoi les constructeurs des classes *Pixel* et *Image* sont définis comme suit :

Pour la classe Pixel :

```
def __init__(self, r, g, b):
    self.red = r
    self.green = g
    self.blue = b
    if self.red < 0 or self.green < 0 or self.blue < 0:
        raise ValueError("RGB values must be positive.")
    self.pixel = (self.red, self.green, self.blue)
```

Pour la classe Image :

```
 def __init__(self, width: int, height: int, pixels: list[Pixel]):
    self.w = width
    self.h = height
    self.img = pixels
    if self.w * self.h != len(self.img):
      raise ValueError("Quantity of pixels doesn't match the image's dimension.")
    for i in pixels:
        if not isinstance(i, Pixel):
                raise ValueError("List's elements aren't Pixel instances.")

def __getitem__(self, pos: tuple[int, int]) -> Pixel:
    x = pos[0]
    y = pos[1]
    if not (x <= self.w and y <= self.h):
        raise IndexError("Not enough pixels.")
    return self.img[y * self.w + x]

def __setitem__(self, pos: tuple[int, int], pix)-> None:
    x = pos[0]
    y = pos[1]
    if x < self.w or y < self.h:
        self.img[x + y * self.w] = pix
    else:
        raise IndexError("Invalid position.")
```

Certaines fonctions spécifiques à la construction de classes doivent être créé pour faciliter la manipulation des pixels et des images. Par exemple, la fonction `__getitem__` de la classe *Image* permet de récupérer un pixel à partir de ses coordonnées (x, y). Ou bien la fonction `__setitem__` qui permet de modifier un pixel à partir de ses coordonnées (x, y).

2.2 Encoder

La classe *Encoder* possède les attributs suivants : une instance de la classe *Image*, la version du compresseur que nous voulons utiliser (mis par défaut à 1), et **kwargs. Ce dernier permet de passer des arguments optionnels, tels que la profondeur et une une valeur booléene déterminant l'utilisation de l'encodage RLE (passer à True si nous souhaitons utiliser la encodage RLE, False sinon). Cette méthode sera utiliser pour enregistrer une image.

La première version de l'*Encoder* permet seulement de convertir des valeurs entières en bits un à un et ensuite l'écrire dans un fichier. Cela est faisable grâce à la fonction `int.to_bytes`.[1]

```
bpixels = b''.join([pixel.get_red(),
        pixel.get_blue(), pixel.get_green()]
        for pixel in pixels)
```

Cependant, cette méthode n'est pas optimale car elle ne permet pas de compresser l'image. En effet, cette version correspond à l'exemple utilisé dans l'introduction. Pour une image de dimension 360×480, il y a 172800 pixels, soit $172800 \times 3 = 518400$ bytes. Nous allons donc utiliser l'encodage RLE, notion cité précemment. Ce dernier consiste à remplacer une suite de bits identiques par un seul bit qui correspondra au nombre de répétitions du bit. Dans notre cas, si nous avons une suite de 16 pixels identiques, nous utiliserons un byte qui correspondra à 16. Ainsi, au lieu d'encoder les pixels sur 3 bytes, nous les encoderons sur 4 bytes

[1] `pixels` est une liste contenant des éléments de la classe `Pixel`. `get_red(), get_blue() et get_green()` sont des méthodes de la classe Pixel qui permettent de récupérer les valeurs RGB d'un pixel.

dont le premier représentera le nombre de répétitions des bits suivants. De cette manière, nous réduisons la taille de l'image d'un facteur 12. En effet, à la place d'avoir une image contenant $16 \times 3 = 48$ octets, nous n'aurons besoin que de $3 + 1 = 4$ bytes pour encoder les couleurs.

Néamoins, il y a une limite concernant l'efficacité de l'encodage RLE. Dans le cas où une image possède beaucoup de couleurs différentes, il y aura tout de même 4 bits utilisés pour un seul pixel. Donc si nous reprenons l'exemple utilisé précedemment, il y aurait $480 \times 360 \times 4 = 691200$ octets utilisés.

Voilà pourquoi, la quatrième version permet de choisir la profondeur(ou bits par pixel), elle correspond au nombre de bits utilisés pour encoder chaque couleur d'un pixel. Par exemple, si la profondeur est de 4, cela signifie que chaque couleur d'un pixel est encodée sur 4 bits. Dans ce projet, nous considérerons les profondeurs suivantes : 1, 2, 4, 8 et 24.

Voici une fonction qui permet de convertir une liste de tuple d'entiers en une chaine de bits grâce à la liste `padding` qui est la palette de couleurs utilisées.

```
def split_string(grouped_pixels: list[str], padding:list[str]) -> bytes:
        color_to_index = {color: index for index, color in enumerate(padding)}
        indices = []
        for pixel in grouped_pixels:
                indices.append(padding[pixel])
        bit_string = ''.join(str(bit) for bit in indices)
        if len(bit_string) != 8:
                shift = 8 - len(bit_string)
        shifter = '0' * shift
        shifted_bit_string = bit_string + shifter
        decimal_value = int(shifted_bit_string, 2)
        hex_value = bytes([decimal_value])
        return hex_value
```

Nous avons également dû implémenter une quatrième version de l'*Encoder*. L'idée est de regrouper les pixels en fonction de leur différence de couleur. Prenons pour exemple 2 pixels : $P_1 = (R_1, G_1, B_1)$ et $P_2 = (R_2, G_2, B_2)$, la différence de couleur entre ces deux pixels est définie comme suit :

$$\Delta_R = R_2 - R_1$$
$$\Delta_G = G_2 - G_1$$
$$\Delta_B = B_2 - B_1$$

Ainsi, en fonction des intervalles de valeurs de Δ_R, Δ_G et Δ_B, nous allons regrouper les pixels en fonction de leur différence de couleur. Plus la différence sera grande, plus on aura besoin de byte pour encoder les pixels. En effet, si la différence n'est pas grande (c'est-à-dire entre -2 et 1), la couleur des pixels sont assez proche pour qu'on ne voit pas la différence à l'oeil nu. Ainsi, on peut se permettre de les encoder sur moins de bytes.

2.3 Decoder

La classe *Decoder* se repose sur la méthode statique `load_from` qui prend en paramètre le chemin d'accès du fichier à lire. Cette méthode permet de lire les bytes du fichier et de les convertir en une instance de la classe *Image*. Cette fonction sera utile lorsque l'utilisateur voudra ouvrir une image.

Bien évidemment, en fonction de la version du fichier encodé, il va falloir adapter la méthode `load_from`. Mais l'idée principale reste tout de même de convertir une série de bytes précise étant donné que les fichiers

ULBMP sont encodés de cette manière. Cela est faisable grâce à la fonction `int.from_bytes` qui permet de convertir une série de bytes en entier. Par la suite, nous stockerons ces valeurs dans une liste, type accepté par la classe `Encoder`.

Nous nous attarderons brièvement concernant l'analyse de la version 1.0 et 2.0 du `Decoder` puisqu'elle consiste seulement à lire les bytes un intervalle précis. En effet, les valeurs RGB des pixels sont encodées sur 3 bytes dans la première version du fichier ULBMP. Ainsi, il suffit de lire les bytes 3 par 3 pour obtenir les valeurs RGB de chaque pixel. Quant à la version 2.0, les valeurs RGB sont encodées sur 4 bytes (d'abord le nombre de répétition du pixel, ensuite les valeurs RGB), ce qui signifie que nous devons lire les bytes 4 par 4.

```
if version.to_bytes(1, "little") == b'\x01':
        list_pixels = [(int.from_bytes(pixels[i:i+1]),
                int.from_bytes(pixels[i+1:i+2]),
                int.from_bytes(pixels[i+2:i+3]))
                for i in range(0, len(pixels), 3)]

        img = [Pixel.get_pixel(p) for p in img]

if version.to_bytes(1, "little") == b'\x02':
        list_pixels = [(int.from_bytes(pixels[i:i+]),
                int.from_bytes(pixels[i+1:i+2]),
                int.from_bytes(pixels[i+2:i+3]))
                int.from_bytes(pixels[i+3:i+4])
                for i in range(0, len(pixels), 4)]

        img = [Pixel.get_pixel(p) for p in img]
```

Concernant la troisième version du `Decoder`, elle dépend du paramètre RLE. Si ce dernier est à `True`, il suffit de lire les bytes 4 par 4, comme la version 2.0. Sinon, il faut séparer la séquence de bytes qui correspond aux valeurs RGB des pixels en fonction de la profondeur précisée dans le header. Pour cela, une fonction appelée `split_into_pixels` a été créée, elle permet de séparer les bytes en fonction de la profondeur. Ele prend en paramètre une chaine de caractère correspondant au bytes et un entier représentant la profondeur.

```
def split_into_pixels(bits: str, depth: int) -> list[bytes]:
    pixels = []
    for i in range(0, len(bits), depth):
        pixels.append(bits[i:i+depth])
    return pixels
```

Enfin, la quatrième version du `Decoder` est la plus complexe. En effet, il faut déterminer les intervalles de valeurs de Δ_R, Δ_G et Δ_B pour regrouper les pixels en fonction de leur différence de couleur. Pour cela, une fonction `delta_colors` qui prend en paramètre deux `Pixel` a été créée. Elle permet de calculer la différence de couleur entre deux pixels. De plus, plusieurs fonction ont été créées pour vérifier si tel pixel est dans un intervalle de valeurs précis. Par exemple, la fonction `is_big_difference_g`, qui prend en paramètre la valeurs RGB de la diffférence calculée précédemment, retourne `True` si la Δ_G est entre -128 et 127 et si la $\Delta_R G$ et la $\Delta_B G$ sont entre -32 et 31.[2]

```
def is_big_difference_g(r: int, g: int, b: int) -> bool:
        BIG_DIFF_RANGE = range(-128, 128)
        BIG_RELATIVE_DIFF_RANGE = range(-32, 32)
        return g in BIG_DIFF_RANGE
                and r in BIG_RELATIVE_DIFF_RANGE
```

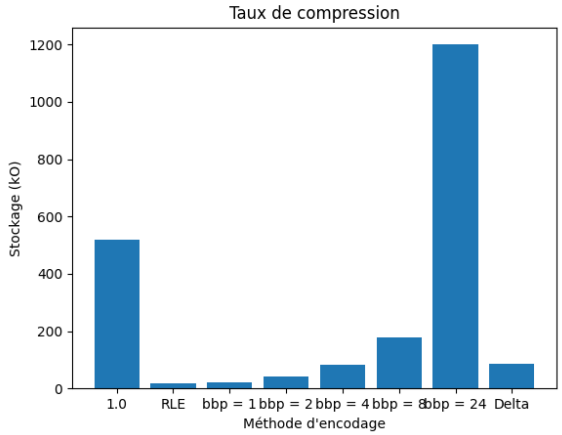
[2]RG et BG signifient qu'il s'agit de la différence entre la couleur rouge et la couleur verte, et entre la couleur bleu et la couleur verte respectivement.

```
                and b in BIG_RELATIVE_DIFF_RANGE

def delta_colors(color1: Pixel, color2: Pixel) -> tuple[int, int, int]:
        DR, DG, DB = color2.red - color1.red,
                color2.green - color1.green, color2.blue - color1.blue
        return DR, DG, DB
```

3 Résultats

Naturellement, il est interéssant de comparer les différentes versions du compresseur. Pour ce faire, nous avons utilisé une image de dimension 360×480 pixels. Nous avons ensuite compressé cette image en utilisant les différentes versions du compresseur. Et voici les résultats obtenus pour l'image suivante appelé lines.ulbmp :



Comme nous pouvons le constater, la version 1.0 est la moins efficace en terme de compression. En effet, elle ne compresse pas l'image, elle se contente de l'écrire dans un fichier. De plus, lorsque nous choisissons de compresser en utilisant une profondeur de 24 bits par pixels, la taille du fichier est plus grande que si nous n'utilisions pas de compression.

4 Conclusion

Ce projet nous a permis de mieux comprendre le fonctionnement des images et des pixels. Nous avons pu voir que les images sont composées de pixels qui sont eux-mêmes composés de valeurs RGB. Nous avons également pu voir que les images peuvent être compressées en utilisant l'encodage RLE. Cela permet de réduire la taille de l'image tout en gardant une qualité acceptable. Enfin, nous avons pu voir que la profondeur des pixels est un paramètre important à prendre en compte lors de la compression d'une image. En effet, plus la profondeur est grande, plus la taille de l'image sera grande.