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 compresseur d'image appelé 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 bytes (un byte pour chaque couleur), cela signifie que l'image prendrait $360 \times 480 \times 3 = 518400$ bytes, soit 518.4 kilo-octets. Nous pourrions se 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 bytes 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 bytes 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 bytes 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 à 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):
    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]):
    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):
    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).

La première version de l'*Encoder* permettait seulement d'analyser les bits un à un et ensuite l'écrire dans un fichier. Cependant, cette méthode n'est pas optimale car elle ne permet pas de compresser l'image. C'est pourquoi, il est important de créer une deuxième version de l'*Encoder* qui permettera de compresser l'image.

Nous avons cité précedemment la notion de encodage RLE. 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 10 pixels identiques, nous utiliserons un byte qui correspondra à 10. Ainsi, au lieu d'encoder les pixels sur 3 bytes, nous les encoderons sur 4 bytes dont le premier représentera le nombre de répétitions des bits suivants.

Quant à la profondeur, 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. Cette méthode d'encodage sera utiliser pour la version 3.0 du compresseur.

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. Si par exemple, soit deux 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.

Comme cité précédemment, les valeurs d'un pixel ne peuvent pas être négatives. Il va falloir décaler les valeurs de différences car il est possible que ces dernières soient négatives. De plus, le calcul de différence se fera toujours à partir du pixel actif et le pixel précédent.

Comme il fallait implémenter plusieurs versions de l'*Encoder*, il était important de créer une méthode `get_header` qui permet de retourner une suite de bytes correspondant au header contenant la constante ULBMP (c'est ce qui permet d'identifier nous sommes en train d'analyser le format correct du fichier), la hauteur, la largeur, la taille du header (mis par défaut à 12 car elle varie seulement pour la version 3) et la version. En fonction de la valeur du paramètre *version*, nous ajouterons des informations supplémentaires dont la profondeur et l'encodage RLE. À ceux-là s'ajoute la palette, utilisé pour la version 3.0.

La palette est une liste contenant seulement les couleurs se trouvant sur l'image. Ainsi, nous stockerons les couleurs et leurs indices dans un dictionnaire pour pouvoir faire appel à ces derniers plus rapidement.

```
1  list_pixels = []
2  for i in pixels:
3          list_pixels.append((int.to_bytes(pixel.get_red(), byteorder='little'),
4                              int.to_bytes(pixel.get_green(), byteorder='little'),
5                              int.to_bytes(pixel.get_blue(), byteorder='little'))
6  padding = list(set(list_pixels))
7  grouped_pixels = [b''.join(element) for element in list_pixels]
8  result = [b''.join(element) for element in padding]
9  color_to_index = {color: index for index, color in enumerate(result)}
10
11 indices = []
12 for pixel in grouped_pixels:
13         indices.append(color_to_index[pixel])
14
15 bit_string = ''.join(str(bit) for bit in indices)
16
17 if len(bit_string) != 8:
18         shift = 8 - len(bit_string)
19 shifter = '0' * shift
20 shifted_bit_string = bit_string + shifter
21 decimal_value = int(shifted_bit_string, 2)
```

La variable `pixels` est obtenu grace à la méthode de la classe `Image` `get_pixels` qui retourne l'attribut `pixels`

À la ligne 6, les fonctions `list` et `set` permettent de supprimer les doublons car c'est possible d'avoir plusieurs fois la meme couleur

La fin du code consiste à parcourir la liste de pixels et trouver les indices correspondants dans la palette. Et enfin, nous convertissons les bits en entier en fonction de la profondeur.

des couleurs utilisé des bytes pour encoder les indices des couleurs de la palette. Par exemple, plutôt que d'encoder 2^k. Cela permet de réduire la taille de l'image car on n'a plus besoin de stocker les valeurs RGB de chaque pixel.

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 quant à 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':
        img = [(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':
        img = [(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):
    pixels = []
    for i in range(0, len(bits), depth):
        pixels.append(bits[i:i+depth])
    return pixels
```

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 :

4 Conclusion