Rapport de Travaux dirigés

UE Informatique - INF tc1

Compression d'image

Groupe

GUEYE Khady

MIRANDA GOMES Vitor

Chargé de BE :

CHEN Liming

Année 2021 Mai

1. Introduction

L'objectif de ce BE est d'appréhender quelques techniques de compression et d'optimisation d'image. Nous travaillerons dans la partie 3 de ce TP avec L'image suivante. Elle est totalement libre de droit (sans attribution nécessaire, référence en annexe) :

Figure 1 - Exemple d'arbre

2. Réponses aux questions

2.1. Question 1.5: Comment gérez-vous les tailles impaires, comme 5x5? Comment gérez-vous les tailles minimales comme 1x2?

Pour gérer les tailles impaires on applique une floor division par 2 pour les dimensions maximales du rectangle, c'est-à-dire w et h, comme cela, on aura une division qui engendre quatre rectangles de tailles différentes.

```python
    #quadripartition d'un rectangle
    def div_rect(self,rect):
        try:
            x=rect[0]
            y=rect[1]
            w=rect[2]
            h=rect[3]

            #gestion du cas w ou h<=1
            if w<=1 or h<=1:
                return []

            #gestion des cas impairs

            new_rect1=[x,y,w//2,h//2]
            new_rect2=[x+w//2,y,w-w//2,h//2]
            new_rect3=[x,y+h//2,w//2,h-h//2]
            new_rect4=[x+w//2,y+h//2,w-w//2,h-h//2]

            return [new_rect1,new_rect2,new_rect3,new_rect4]

        except:
            return "erreur"
```

Figure 2 - Code pour la question 1.5

On va diviser un rectangle 5x5 par exemple. Le code nos renvoie les dimensions suivantes: ([0, 0, 2, 2], [2, 0, 3, 2], [0, 2, 2, 3], [2, 2, 3, 3]), ce que nous amène à la division suivante:

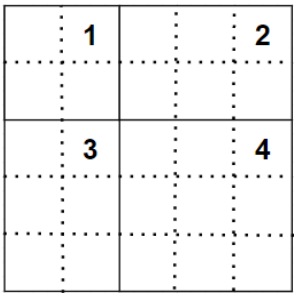

Figure 3 - Division d'un rectangle 5x5

En plus, le cas limite c'est quand on trouve une des dimensions égale à un, ce qui est visiblement indivisible. Pour gérer ce problème on a mis une

condition *if* que retourne une liste vide quand on trouve une dimension inférieure ou égale à un.

3. Partie 3

On a réfléchi à plusieurs solutions pour la réalisation d'un arbre implicite et nous avons abouti à une optimale que nous allons détailler après avoir expliqué pourquoi les autres méthodes ne marchaient pas.

3.1. Première méthode:

Il s'agirait de créer une liste où l'on stockerait les nœuds les uns après les autres et en définissant une équation pour retrouver le parent et les enfants. Pour trouver les formules permettant de calculer pour un nœud à l'indice *i* les indices de son parent et de ses quatre enfants, l'opération serait *(i-1)//4* pour l'indice du parent, c'est-à-dire, on prend celui qu'on analyse et on l'applique une *floor division (//)*. Par exemple, pour 13 on trouve 3. D'un autre côté, pour les enfants on utilise la formule *(4i + 1 ou 2 ou 3 ou 4).*

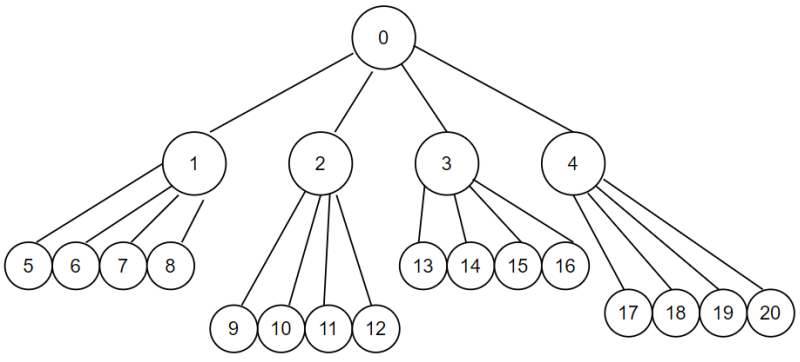

Figure 3 - Exemple d'arbre

Le problème avec cette méthode, c'est que lorsqu'il y a un nœud terminal, puisque celui-là n'a pas d'enfants, l'ordre des nœuds change et ne suit donc plus l'équation que nous avons définie, et cela peu importe l'équation. Pour régler ce problème, on peut supposer que chaque nœud

terminal sera affecté dans la liste d'une valeur nulle, *None,* il aura aucune valeur, par contre, il maintiendra l'ordre pour que la formule soit encore applicable. Finalement, celle-là se montre une option pas du tout optimale parce qu'on trouvera énormément de *None*, occupant de l'espace dans la mémoire de l'ordinateur alors qu'ils pourraient être inexistants dans la liste.

3.2. Notre solution:

La première méthode n'étant pas optimale, nous avons réfléchi à une manière de stocker les nœuds permettant à la fois d'optimiser l'espace mais aussi de gérer la nomenclature des parents et enfants. On a donc utilisé un dictionnaire comme structure de données pour organiser l'arbre.

La nomenclature serait maîtrisée peu importe l'équation que nous choisissons pour identifier les nœuds puisque on gère maintenant leurs indices. On a choisit une nomenclature [ij] , i etant l'indice du parent et j celui de l'enfant comme le montre la figure suivante. Nous avons de cette manière géré la nomenclature des indices mais aussi optimisé notre espace comme nous le verrons dans la partie 3.3.

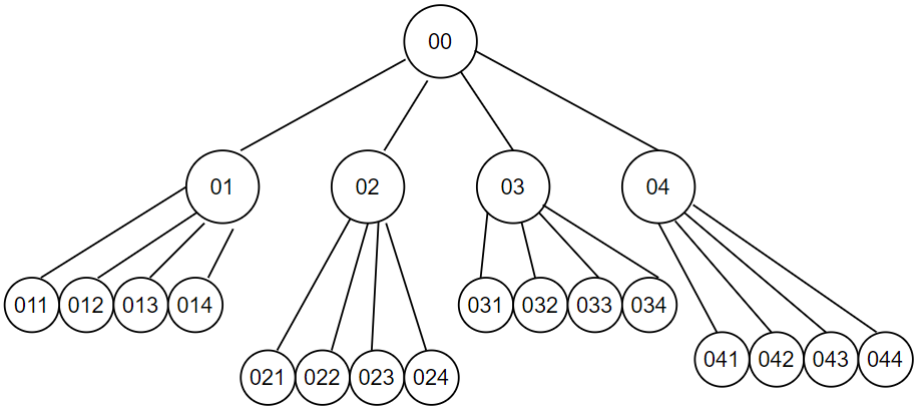

Figure 3 - Exemple d'arbre avec le dictionnaire

4. Partie 3.2.

Pour implementer notre solution, nous avons redefini la classe noeud en enlevant toutes les references explicites au parent et à l'enfant.

```python
class Noeud:
    def __init__(self, x, y, w, h, r, g, b):
        self.x = x
        self.y = y
        self.w = w
        self.h = h
        self.red = r
        self.green = g
        self.blue = b
```

Par la suite nous avons réécrit la fonction permettant de créer les noeuds.

Elle va maintenant stocker les Noeuds créés dans un attribut appelé self.arbre, qui est un dictionnaire qui va stocker l'ensemble des noeuds créés selon la nomenclature définie dans la partie 3.1

```python
#Cree un noeud à partir d'un rectangle et retourne le noeud correspondant à l'arbre de descendance
def createNoeud(self,rect,indice_parent):
    try:
        #creation du noeud racine de l'arbre en fonction du rectangle
        self.arbre[indice_parent]=Noeud(rect[0], rect[1], rect[2], rect[3], 128, 128, 128)

        #quadripartition du rectangle
        kids=self.div_rect(rect)

        #la liste de noeuds fils
        kid=[]

        #creation des noeuds pour les enfants
        if kids!=[]:
            kid.append(Noeud(kids[0][0],kids[0][1],kids[0][2],kids[0][3],128, 128, 128))
            kid.append(Noeud(kids[1][0],kids[1][1],kids[1][2],kids[1][3],128, 128, 128))
            kid.append(Noeud(kids[2][0],kids[2][1],kids[2][2],kids[2][3],128, 128, 128))
            kid.append(Noeud(kids[3][0],kids[3][1],kids[3][2],kids[3][3],128, 128, 128))


            #affectation des premieres branches de l'arbre
            self.arbre[indice_parent+"1"]=Noeud(kids[0][0],kids[0][1],kids[0][2],kids[0][3],128, 128, 128)
            self.arbre[indice_parent+"2"]=Noeud(kids[1][0],kids[1][1],kids[1][2],kids[1][3],128, 128, 128)
            self.arbre[indice_parent+"3"]=Noeud(kids[2][0],kids[2][1],kids[2][2],kids[2][3],128, 128, 128)
            self.arbre[indice_parent+"4"]=Noeud(kids[3][0],kids[3][1],kids[3][2],kids[3][3],128, 128, 128)

            #appel recursif de la fonction sur les enfants
            self.createNoeud(kids[0],indice_parent+"1")
            self.createNoeud(kids[1],indice_parent+"2")
            self.createNoeud(kids[2],indice_parent+"3")
            self.createNoeud(kids[3],indice_parent+"4")

        return self.arbre

    except:
        return "erreur ici"
```

5. Partie 3.3.

Question 3.3: Quels effets a eu l'introduction d'une structure implicite sur la performance de votre programme?

Nous avons réalisé plusieurs exécutions du programme avec la structure explicite puis avec la structure implicite. Les résultats sont le suivants:

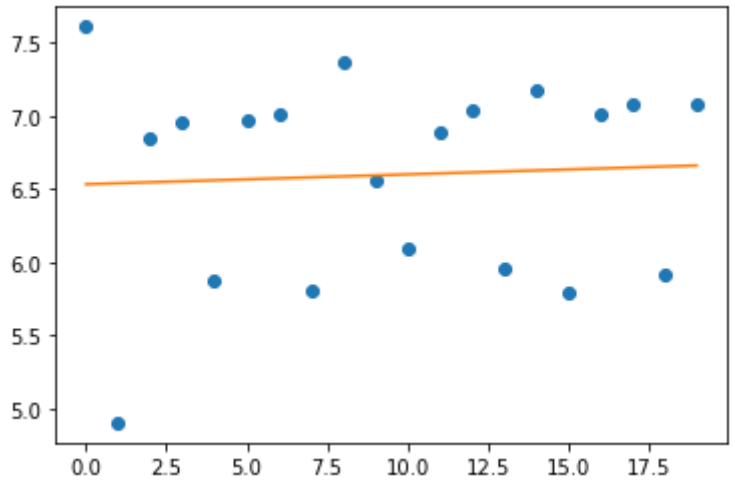

Figure 4 : ***Programme 1 (structure explicite): Moyenne = 6.55 secondes***

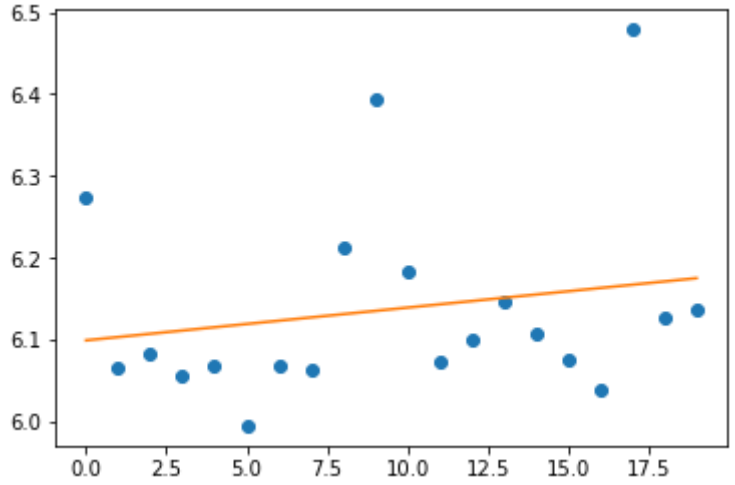

Figure 5 : ***Programme 2(structure implicite):*** **Moyenne = 6.09 secondes**

On peut remarquer ici une différence de temps d'exécution pour les deux programmes , qui, avec le nombre de nœuds à créer, se creuse de plus en

plus. La structure implicite a donc considérablement diminué le temps d'exécution du programme.

Concernant la mémoire, le premier programme va stocker un nœud de taille 40 pour chaque fils. On a donc multiplié la taille d'un nœud par le nombre de nœuds existant ce qui nous donne la consommation mémoire du système nodal.

```
nbrFils=image.nombreFils(noeud,1)
print(sys.getsizeof(noeud)*nbrFils)
```

Figure 6 :

Pour le deuxième programme nous avons juste cherché la taille du tableau stockant les valeurs :

```
print(sys.getsizeof(noeud))
```

Figure 7 :

Nous avons réalisé ces tests sur un carré de 10x10 pour minimiser le temps d'attente de création des nœuds et le résultat montre une très grande différence.

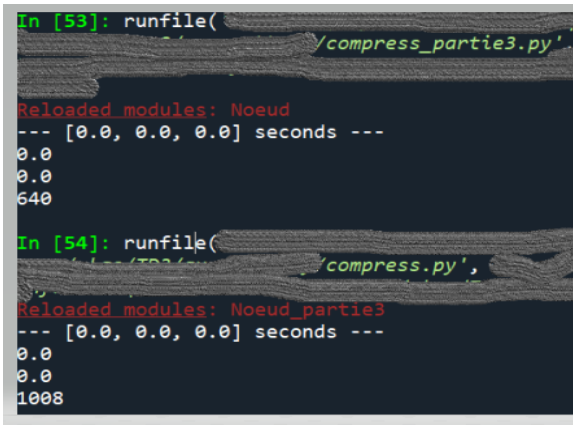

Figure 8 : comparaison de l'occupation mémoire des structures explicite et implicite

Nous avons 1008 bytes consommés pour le système explicite contre 640 bytes pour le système implicite. Sur une plus grande plage, la différence est d'autant plus remarquable.

On peut donc dire que la structure implicite a permis d'optimiser largement notre programme.

6. Question 3.4 : Système PSNR basé sur de nouveaux critères

Nous avons ici réfléchi à des critères permettant d'optimiser la qualité de compression de notre image et après plusieurs recherches nous en avons retenu quelques un: la sensibilité de l'œil au vert et aux couleurs environnantes ainsi que la sensibilité à la luminosité. Un autre critère qu'il serait intéressant d'exploiter est la compression selon les différences de contraste. Nous allons expliquer chaque méthode dans la partie qui suit.

1e et 2e critères: exploitation de la sensibilité de l'oeil au vert et à la luminosité

Plusieurs articles consultés nous expliquent que l'oeil humain ne perçoit pas toutes les couleurs de la même manière. Sur une image ayant 95% de vert et 5% de bleu, l'oeil est susceptible de ne pas distinguer la couleur bleu, par contre sur une image avec 95% de bleu et 5% de vert, le vert sera naturellement aperçu. Ce qui veut dire que l'oeil est naturellement plus sensible au vert qu'aux autres couleurs.

En exploitant cela, on pourrait faire en sorte de négliger les détails dans les zones de l'image ayant un taux de vert élevé car elles seront de toute façon perçues et d'augmenter la précision dans les zones les moins perçues.

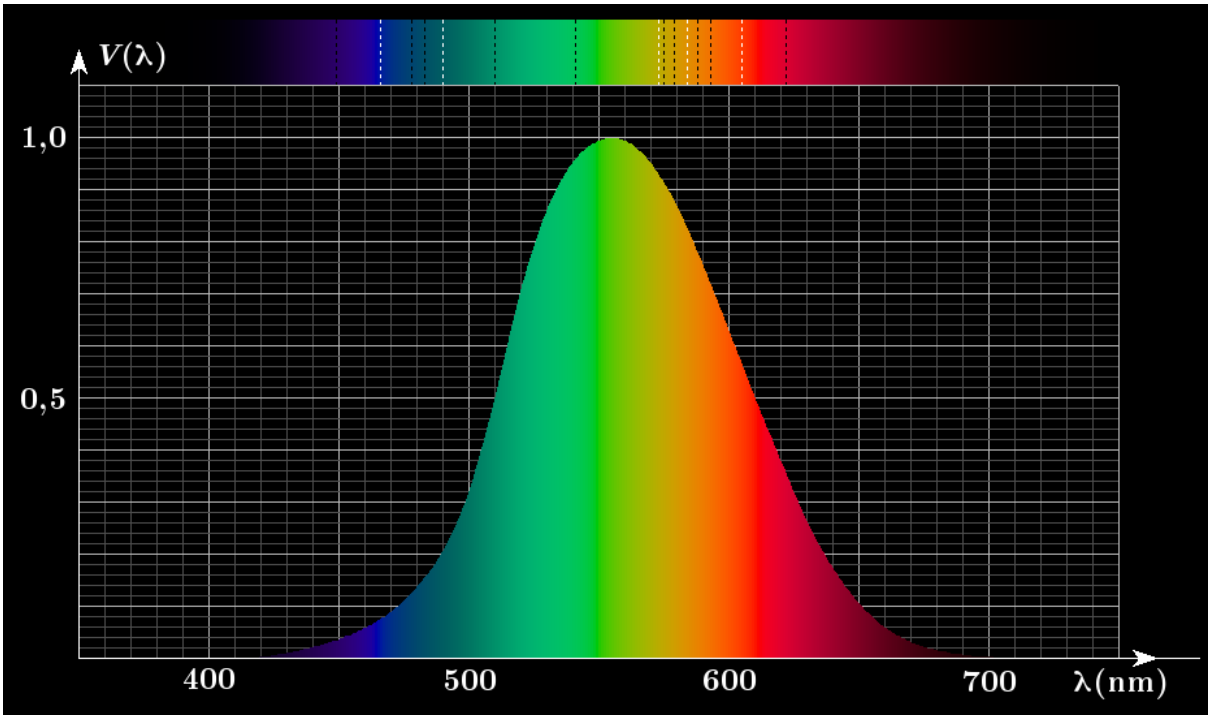

Figure 9 : sensibilité de l'oeil aux differentes ondes spectrales de couleur

On remarque sur la figure que les radiations de lumière verte viennent stimuler fortement les cônes de la rétine : c'est ainsi que la vision humaine s'avère beaucoup plus sensible et nuancée dans les tons verts.

On va donc dans notre mesure de distorsion chercher à augmenter l'erreur quand celle-ci porte sur la couleur verte et la diminuer (car moins importante pour l'œil humain) pour la couleur bleue.

On fera de même pour la sensibilité à la lumière car comme nous l'avons vu dans les articles de recherche (en annexe), les détails sont moins bien perçus dans les zones à forte luminosité par rapport aux zones à luminosité moyenne. Nous allons donc réduire l'erreur quadratique ou l'augmenter en fonction de la luminosité des différentes zones. Nous allons donc nous baser sur un seuil "luminosité forte", qui définit un seuil à partir duquel on considère la couleur comme claire ou sombre.

```python
def EQ(self):
 if self.couleurs != None :
    eq = 0
    for i in range(len(self.couleurs)):
        R,G,B = self.couleurs[i]
        x,y,w,h=self.zones[i]
        for j in range(y,y+h):
            for i in range(x,x+w):
                r,g,b = self.px[i,j]
                eq += (R-r)**2 + accent_vert*(G-g)**2 + (2-accent_vert)*(B-b)**2  # on ajoute un coefficient pour augmenter
                if x>centre_image_w and x<W-centre_image_w and y>centre_image_h and y<H-centre_image_h: # si la zone est da
                    eq = accent_centre*eq # on augmente l'erreur
                else :
                    eq = (2-accent_centre)*eq # on diminue l'erreur
                if (R+G+B)/3 > zone_luminosité_forte: # si la couleur de la zone est relativement lumineuse/éclairée
                    eq = accent_luminosité*eq # on augmente l'erreur
                else :
                    eq = (2-accent_luminosité)*eq # on diminue l'erreur
    return eq
 else :
    eq = 0
    for n in self.enfants:
        eq += n.EQ_modifié()
    return eq

def PSNR(self):
    return 20*log10(255) - 10*log10(self.EQ() / 3 / self.w / self.h)
```

Le résultat après compression est le suivant:

La qualité globale de l'image est améliorée et pour un seuil d'homogénéité plus grand, l'influence du PSNR modifié est plus visible.

Conclusion

Nous avons donc pu tester la compression d'image selon différents critères et donc en apprendre plus sur les méthodes possibles et réalisables.

7. Annexe

Source image :
https://www.pexels.com/photo/mirror-lake-reflecting-wooden-house-in-middle-of-lake-overlooking-mountain-ranges-147411/

source recherches :

https://fr.wikipedia.org/wiki/Vision_des_couleurs

https://fr.wikipedia.org/wiki/Efficacit%C3%A9_lumineuse_spectrale

https://fr.wikipedia.org/wiki/Vision_des_couleurs