

Introduction

Dans ce TD nous allons implémenter une méthode de compression d'image par segmentation. Dans un premier temps vous écrirez des fonctions de manipulation d'image, ensuite la construction récursive d'un arbre de partition de l'image, et enfin la compression de l'image à partir de l'arbre. Ce TD fera l'objet d'un rendu sous forme de rapport, à l'aide d'exemples d'images compressées et de tests de performance, à rendre sur Moodle.

Segmentation d'image par quadripartition récursive

La segmentation d'une image consiste à regrouper les pixels similaires en régions connexes. C'est une méthode très utilisée en compression d'images, où on attribue une même couleur aux pixels d'une même région. L'image résultante est alors visuellement proche de l'image d'origine, tout en stockant moins de couleurs distinctes de pixels. L'appartenance des pixels à une même région est définie par un critère d'homogénéité, qui peut se baser sur la couleur, la texture, la profondeur de champ, le mouvement, etc.

Les méthodes de segmentation sont généralement regroupées en trois grandes catégories :

1. Segmentation à base de pixels
2. Segmentation à base de contours
3. Segmentation à base de régions

La première catégorie utilise souvent les histogrammes de couleurs de l'image. On répartit toutes les couleurs rencontrées dans un nombre fini de classes (par exemple avec la méthode des K-moyennes), puis les pixels de chaque classe sont peints avec une même couleur. La deuxième catégorie utilise l'information de contours des objets pour délimiter des régions distinctes. La troisième catégorie correspond aux méthodes de croissance, décomposition ou fusion de régions. Dans le premier cas, on part d'un ensemble initial de régions (par exemple des couleurs extrêmes de l'image), qu'on fait croître en incorporant les pixels les plus similaires. Le deuxième cas est une approche *top-down* : on part de l'image entière que l'on va subdiviser récursivement en plus petites régions tant que ces régions ne sont pas suffisamment homogènes. Le troisième cas est une approche *bottom-up* : tous les pixels sont initialement couverts par de petites régions indépendantes, et on fusionne les régions voisines homogènes tant qu'un critère particulier n'est pas respecté (nombre de régions maximale, qualité visuelle de l'image). La décomposition peut être associée à une fusion, dans la méthode *split and merge* proposée par Pavlidis et Horowitz en 1974. Dans ce TD nous utiliserons la méthode de décomposition pour compresser des images.

La méthode de segmentation que nous allons utiliser se base sur des régions rectangulaires de l'image. La segmentation d'une image consiste en un pavage de celle-ci avec des rectangles, chacun stockant une couleur unique pour les pixels couverts. Pour construire ce pavage, nous utilisons un algorithme de quadripartition récursive. L'image est initialement couverte par un unique rectangle de taille maximale. Puis celui-ci est sous-divisé en quatre rectangles de tailles égales. Chaque sous-rectangle est à nouveau sous-divisé, et ainsi de suite, jusqu'à ce que les plus petits rectangles couvrent chacun

un pixel. La figure 1 montre une image en noir et blanc 4×4 pixels et le découpage correspondant en trois niveaux.

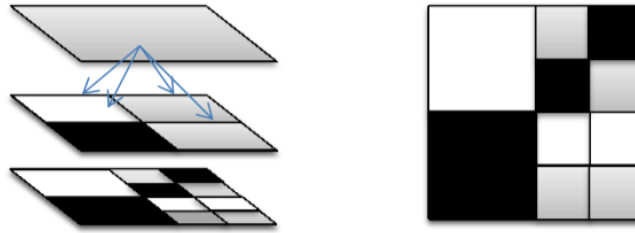


Figure 1: Découpage par quadripartition d'une image 4x4 pixels.

On associe une structure d'arbre à cette division récursive, dans laquelle chaque noeud est une région de l'image, et ses enfants sont les rectangles issus de sa sous-division en quatre. Les feuilles de l'arbre sont les régions homogènes de l'image, qu'on n'a plus besoin de sous-diviser. La structure d'arbre associée au découpage en exemple est illustrée en figure 2.

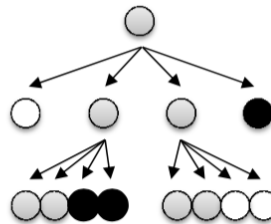


Figure 2: Arbre quaternaire à partir de l'image de la figure 1.

Dans cet exemple, le critère d'homogénéité est absolu. Une zone est dite homogène si elle ne contient que des pixels de la même couleur (seuil d'homogénéité = 0). En pratique on est plus tolérant et considère qu'une zone est homogène dès que l'écart-type de ses couleurs est inférieur à un seuil σ_h . De manière plus générale, on va appliquer ce principe de réduction à des images colorées. Chaque pixel d'une image en couleur est représenté par trois intensités en rouge, vert et bleu. Chaque intensité est codée sur un octet, sa valeur varie de 0 à 255. L'écart-type d'une région se calculera par moyenne des écarts-types en rouge, vert et bleu pour la région. Il est homogène à une différence de couleur (donc compris entre 0 et 255). Lorsqu'il est inférieur au seuil σ_h , la région est considérée comme homogène et constitue une feuille, noeud terminal de l'arbre. On lui attribue alors la couleur de la moyenne des pixels la constituant. Au-dessus de σ_h , la région n'a pas de couleur et est découpée en quatre.

Un exemple d'image traitée par l'algorithme quadripartition est illustré par la figure 3. L'image originale est traitée avec des valeurs de seuil de plus en plus petites augmentant le nombre de régions détectées.

En haut à gauche, l'image originale non segmentée. Les suivantes avec des valeurs de seuil de plus

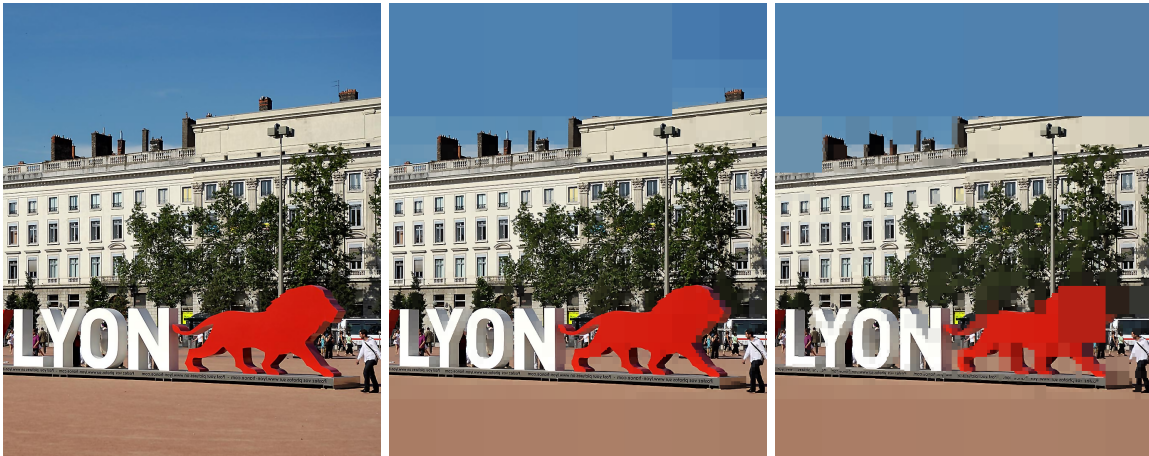


Figure 3: Image de Lyon et les résultats obtenus avec différentes valeurs de seuil.

en plus basses augmentant le nombre de régions.

1 Chargement d'une image et fonctions utilitaires

On souhaite réaliser l'algorithme *top-down* qui utilise la stratégie de quadripartition rectangulaire. Ceci nécessite l'utilisation de la librairie `PILLOW` fournie avec Anaconda :

```
from PIL import Image
```

Nous aurons ensuite besoin de lire une image à partir de son nom (à placer dans le même répertoire que le script Python) :

```
im = Image.open("lyon.png")
```

Pour importer les données des pixels sous forme d'une matrice `px` :

```
px = im.load()
```

Pour obtenir la taille de cette image :

```
W, H = im.size
```

Vous devez donc exécuter la séquence suivante au début de votre programme :

```
from PIL import Image # importation de la librairie d'image PILLOW
from math import sqrt, log10 # fonctions essentielles de la librairie math
im = Image.open("lyon.png") # ouverture du fichier d'image
px = im.load() # importation des pixels de l'image
W, H = im.size
```

Par la suite on peut accéder au pixel `px[x,y]` de coordonnées (x,y) par la commande suivante :

```
couleur = px[x, y]
```

La couleur étant un tuple (r,g,b) , on peut directement récupérer les composantes du pixel :

```
r, g, b = px[x, y]
```

Pour affecter une couleur (trois variables r, g et b) au pixel `p[x,y]`, on utilisera la commande :

```
px[x, y] = r, g, b
```

Enfin pour afficher l'image dans une nouvelle fenêtre, on utilise la commande :

```
im.show()
```

Si les commandes de lecture d'image et d'accès aux pixels ne sont pas disponibles, c'est que la librairie `PILLOW` n'est pas installée. Pour l'installer vous devez exécuter la commande suivante dans une fenêtre de terminal Anaconda (Menu démarrer → Anaconda 64bit → Anaconda PowerShell Prompt) : `pip3 install Pillow`.

Exercice 1.1 – Écrire une fonction permettant de peindre un rectangle de l'image avec une même couleur. Vous penserez à tester cette fonction avec des rectangles de tailles minimales/maximales et asymétriques en x/y (ex. 1×1 , 10×100 , $W \times H$).

Exercice 1.2 – Écrire une fonction calculant la moyenne (triplet (r,b,g)) des pixels d'une région rectangulaire. Dans l'image originale en figure 3, par exemple, on renverra $(125.9533571489017, 123.31529434994782, 125.42241302884867)$ pour toute l'image, ou $(70.3, 120.3, 171.3)$ pour le carré de 10 pixels de large en haut à gauche.

Exercice 1.3 – Écrire une fonction calculant l'écart-type (triplet (r,g,b)) des pixels d'une région rectangulaire. Pour rappel, l'écart type d'une composante est définie par :

$$\sigma = \sqrt{\frac{1}{n} \left(\sum_{i=1}^n x_i^2 \right) - \mu^2} \quad (1)$$

Dans l'image originale en figure 3, par exemple, on renverra :

- $(74.32282975727676, 67.51872047739333, 73.11702302391257)$ pour l'image entière.
- $(0.8426149773181971, 0.8426149773171178, 0.8426149773171178)$ pour le carré de 10 pixels de large en haut à gauche.

Exercice 1.4 – Proposez une fonction qui estime l'homogénéité des pixels d'une région rectangulaire de l'image. Elle prendra en entrée un paramètre de seuil sur l'écart-type des pixels de la région, et renverra un booléen indiquant si les pixels sont homogènes.

Exercice 1.5 – Proposez une fonction qui divise un rectangle d'entrée en quatre rectangles plus petits, comme illustré en figure 1. Elle recevra les coordonnées du rectangle à diviser, et renverra une liste de tuples, chacun contenant les coordonnées d'un rectangle plus petit. Comment gérez-vous les tailles impaires, comme 5×5 ? Comment gérez-vous les tailles minimales, comme 1×2 ?

2 Création d'arbre explicite et parcours

Dans cette section nous allons construire un arbre quaternaire à partir de l'image d'entrée, comme dans la figure 2. On utilisera la classe `Noeud` suivante :

```
class Noeud:
    def __init__(self, x, y, l, h, r, v, b, hg, hd, bg, bd):
        self.x = x
        self.y = y
        self.l = l
        self.h = h
        self.r = r
        self.v = v
        self.b = b
        self.hg = hg # haut-gauche
        self.hd = hd # haut-droite
        self.bg = bg # bas-gauche
        self.bd = bd # bas-droite
```

Exercice 2.1 – En utilisant des instances de cette classe, complétez le code des 5 premiers noeuds ci-dessous pour décrire l'arbre représenté en figure 2.

```
racine = Noeud(0, 0, 4, 4, 128, 128, 128,
               Noeud(0, 0, 2, 2, 255, 255, 255, None, None, None, None),
               Noeud(2, 0, 2, 2, 128, 128, 128, None, None, None, None),
               Noeud(2, 2, 2, 2, 128, 128, 128, None, None, None, None),
               Noeud(0, 2, 2, 2, 0, 0, 0, None, None, None, None))
```

Exercice 2.2 – Créez une fonction qui reçoive en entrée une portion de l'image (représentée par un rectangle) ainsi qu'un seuil d'homogénéité, et renvoie le `Noeud` correspondant. Si les pixels de la portion sont homogènes, le `Noeud` sera terminal (pas d'enfants) et sa couleur sera la moyenne des pixels de la portion. S'ils ne sont pas homogènes, le `Noeud` aura au plus 4 enfants, résultats de l'appel récursif de la fonction sur la quadripartition générée à l'exercice 1.5. Le `Noeud` sera donc non-terminal, et sa couleur pourra être fixée à une valeur arbitraire (sans importance).

Exercice 2.3 – Proposez une fonction récursive, qui permette de compter le nombre d'enfants (sous-enfants, sous-sous-enfants, etc.) d'un `Noeud`, en le comptant également. Pour la racine de la figure 2 elle renverrait par exemple 13.

Exercice 2.4 – Proposez une fonction récursive qui parcourt l'arbre depuis sa racine, et peint chaque noeud terminal selon la couleur qui lui a été assignée.

Exercice 2.5 – Écrivez une fonction récursive qui peint les noeuds terminaux d'un arbre d'une couleur proportionnelle à leur profondeur dans l'arbre.

Exercice 2.6 – Pour évaluer la qualité visuelle de l'image dégradée par rapport à l'originale, on utilise une *mesure de distorsion* qui va nous permettre de comparer les résultats de différents critères de compression. Écrivez une fonction qui calcule la mesure Peak Signal to Noise Ratio (*PSNR*) pour l'image complète, en calculant l'Erreur Quadratique (*EQ*) de manière récursive :

$$PSNR = 20 \cdot \log_{10}(255) - 10 \cdot \log_{10} \left(\frac{EQ}{3 \cdot W \cdot H} \right)$$
$$EQ = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} [(I_r[x, y] - I_r^0[x, y])^2 + (I_g[x, y] - I_g^0[x, y])^2 + (I_b[x, y] - I_b^0[x, y])^2]$$

3 Optimisation de la compression (Rendu du TD)

L'objectif de cette section est d'apporter de nouveaux cas d'étude à votre code.

Exercice 3.1 – Dans le cas de grandes images, les arbres de quadripartition risquent d'atteindre beaucoup de noeuds, et la limite de mémoire disponible pourrait être un problème. On souhaite utiliser une structure d'arbre implicite, comme le tas vu au TD 2, pour stocker les arbres de manière plus compacte. Pour rappel, il s'agit de stocker tous les noeuds dans une liste, de déterminer leur position dans l'arbre uniquement par leur position dans la liste, et de n'inclure aucune référence explicite entre noeuds. Donnez d'abord les formules permettant de calculer pour un noeud à l'indice i les indices de son parent et de ses 4 enfants.

Exercice 3.2 – Modifiez maintenant la classe `Noeud` ainsi que votre code pour générer un arbre quaternaire sous forme implicite.

Exercice 3.3 – Quels effets a eu l'introduction d'une structure implicite sur la performance de votre programme ? Vous pourrez par exemple mesurer la vitesse d'exécution et la consommation de mémoire de votre application. Des graphiques seront appréciés ici.

Exercice 3.4 – Le critère d'homogénéité basé sur l'écart-type des pixels est relativement simple à calculer, mais ne prend pas en compte la sensibilité plus ou moins importante de l'œil à différentes caractéristiques de l'image (ex. vert vs bleu, zones de forts contrastes, zone au centre de l'écran). Proposez un nouveau critère d'homogénéité qui donne plus d'importance aux éléments auxquels l'œil est plus sensible (voir par exemple https://fr.wikipedia.org/wiki/Vision_des_couleurs).

Exercice 3.5 – À l'aide d'une image de votre choix, montrez l'effet de votre nouveau critère d'homogénéité sur les zones qui seront dégradées ou préservées en priorité dans l'image. Quelle est son influence sur le PSNR par rapport au critère initial ? Justifiez.

4 Rendu du TD

Vous serez évalués sur un rendu de ce TD via un rapport dont les modalités sont décrites dans le fichier `td3-modalite-rendu.pdf`.