Introduction

Dans ce TD nous allons implémenter une méthode de compression d'image par segmentation. Dans un premier temps vous écrirez un programme de manipulation d'image, ensuite l'implémentation de l'algorithme de segmentation, et enfin l'optimisation de l'algorithme à partir de sa structure de données sous-jacente sous forme d'arbre. A noter que vous pouvez toujours utiliser la plateforme de tests (rappels d'utilisation en annexe) pour les deux premières parties de ce TD. La dernière partie devra cependant faire l'objet d'un rendu sous forme de rapport et d'exemples d'images compressées et tests de performance à rendre sur Moodle.

1 Compression d'image par segmentation

La méthode que nous allons utiliser pour la compression est basée sur la segmentation d'image, qui est un processus de découpe d'une image en régions connexes. La découpe n'est cependant pas toujours systématique, elle dépend de valeurs d'homogénéité des régions selon un certain critère, comme par exemple la couleur, la texture, la profondeur, le mouvement, etc. Ce critère est utilisé pour re-colorier les régions en une seule couleur, et l'image compressée est le résultat de l'union de ces régions re-coloriées.

D'un point de vue de structure de données, les régions de l'image sont connectées à travers un graphe (arbre) où chaque région porte une étiquette donnant des informations qualitatives discriminantes comme sa taille, sa couleur, sa forme, son mouvement ou encore son orientation. L'image est donc réduite à un graphe la représentant où des noeuds étiquetés contiennent presque toutes les informations utiles au système. Les arcs de ce graphe peuvent être valués précisant si deux régions connectées sont en simple contact ou si l'une est incluse dans l'autre. D'autres informations topologiques peuvent également être stockées comme par exemple le positionnement relatif : une région est au-dessous d'une autre, etc.

La construction de ce graphe peut être plus ou moins complexe et dépend des techniques de segmentation utilisées. Les algorithmes de segmentation sont généralement regroupés en trois grandes catégories :

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

La première catégorie utilise souvent les histogrammes de l'image. Par seuillage, les différentes couleurs représentatives de l'image sont identifiées, l'algorithme construit ainsi des classes de couleurs qui sont ensuite projetées sur l'image. La deuxième catégorie correspond aux algorithmes d'accroissement de régions ou de partitionnement de régions. Le premier est une approche *bottom-up* : on part d'un ensemble de petites régions uniformes dans l'image, d'un, voire de quelques pixels, et on regroupe les régions adjacentes d'une même couleur. Ce regroupement s'arrête quand aucun regroupement n'est plus possible. Le deuxième, est une méthode *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. Un mélange de ces deux méthodes est l'algorithme de *split and merge* que nous allons détailler et étudier par la suite. Finalement, la troisième catégorie utilise l'information de contours des objets.

Comme la plupart de ces algorithmes fonctionnent au niveau du pixel, ils sont considérés comme des algorithmes locaux. Ce type d'algorithme est formellement proche des techniques d'accroissement de régions fonctionnant au niveau du pixel et sont purement locales et en général limitées pour traiter des images complexes et bruitées.

1.1 Approche du type "split and merge"

L'algorithme split and merge a été proposé par Pavlidis et Horowitz en 1974. Selon cet algorithme, deux étapes de split, partionnement, et de merge, fusion, sont opérées.

Split. La méthode de découpage de l'image utilisée dans cet algorithme est basée sur la notion de quadripartition (quadtree en anglais). Une structure de données du type arbre quaternaire permet de stocker l'image à plusieurs niveaux de résolution. On part souvent de la totalité de l'image. Si cette image vérifie un certain critère d'homogénéité de couleur, l'algorithme s'arrête. Sinon, on découpe cette région en quatre parties de la même taille et on relance, récursivement, cette procédure dans chacune des quatre parties. La région initiale est considérée comme un noeud dans un graphe et les sous parties comme les quatre fils de ce noeud.

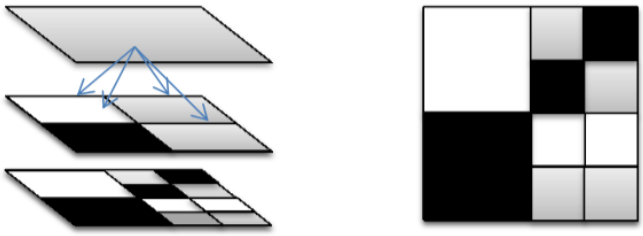


Figure 1: Découpage par quadripartition d'une image 4x4 pixels.

La figure 1 montre une image en noir et blanc 4×4 pixels et le découpage correspondant en trois niveaux. La structure d'arbre associée à ce découpage est illustrée Figure 2.

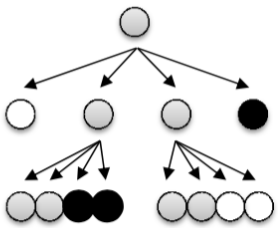


Figure 2: Arbre quaternaire à partir de l'image de la figure 1.

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 = 100%). En pratique on est plus tolérant et considère qu'une zone est homogène dès que plus de 75% d'une couleur domine. 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. Le critère d'homogénéité est fixé par un seuil. En dessous de ce seuil, la région est conservée 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 ce seuil, la région est découpée en quatre.

Merge. La procédure de découpage décrite précédemment aboutit à un nombre de régions parfois trop élevé. Il se peut qu'elle coupe de cette façon une zone homogène en deux ou quatre parties. La solution, qui correspond à la phase fusion (merge en anglais) de l'algorithme, est de procéder à une fusion de régions après le découpage. L'implémentation la plus simple de cette fusion consiste à rassembler, fusionner, les couples de régions adjacentes repérées, de couleur proche, dans l'arbre issu de la phase split. Pour réaliser cette fusion, il faut d'abord tenir à jour une liste des contacts entre régions (chaque région dispose d'une liste de régions avec lesquelles elle est en contact). On obtient ainsi un graphe d'adjacence de régions (Region Adjacency Graph en anglais). Ce graphe de contact doit se construire en même temps que l'arbre de découpage.

Ensuite, l'algorithme va marquer toutes les régions comme non traitées, et choisir la première région R non traitée disponible. Les régions en contact avec R sont empilées et examinées les unes après les autres pour savoir si elles doivent fusionner avec R. Si c'est le cas, la couleur moyenne de R est mise à jour et les régions en contact avec la région fusionnée sont ajoutées à la pile des régions à comparer avec R. La région fusionnée est marquée comme traitée. Une fois la pile vide, l'algorithme choisi la prochaine région marquée comme non traitée et recommence, jusqu'à ce que toutes les régions soient traitées.

1.2 Exemple

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 en plus basses augmentant le nombre de régions.

2 Chargement d'une image et structure de données

On souhaite réaliser l'algorithme de split qui utilise la stratégie quadripartition. Ceci nécessite l'utilisation de la librairie `PILLOW` fournie avec Anaconda. Nous avons notamment besoin de lire une image à partir de son nom dans le répertoire courant :

```
im = Image.open("Image8.bmp")
```

Importer les données des pixels sous forme d'une matrice px :

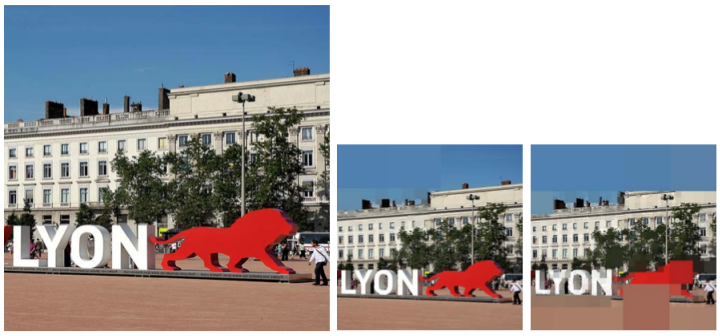


Figure 3: Image de Lyon et les résultats obtenus avec différentes valeurs de seuil.

```
px = im.load() # Importation des pixels de l'image
```

Obtenir la taille de cette image :

```
w,h = im.size
```

Utiliser, et donc importer, la fonction mathématique racine carrée (`sqrt`), qui se trouve dans la bibliothèque `math` :

```
from math import sqrt
```

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

```
from PIL import Image #Importation de la librairie d'image PIL
im = Image.open("Image8.bmp") #Ouverture du fichier d'image
px = im.load() #Importation des pixels de l'image

from math import sqrt #Importation de la fonction sqrt de la librairie math

w,h = im.size
```

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

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

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

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

où r,g et b sont des variables.

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 `pip install Pillow`.

Exercice 1.1 – Ecrire une fonction permettant de lire la valeur d'un pixel.

Exercice 1.2 – Ecrire une fonction permettant d'affecter une couleur à un pixel.

Exercice 1.3 – Ecrire une fonction permettant d'affecter une couleur à une région rectangulaire de l'image.

Exercice 1.4 – Proposez une fonction qui estime l'homogénéité des pixels d'une région rectangulaire de l'image. Il faudra tout d'abord réaliser une fonction qui calcule la moyenne pour chacune des trois couleurs primaires dans la région rectangulaire. Ensuite l'homogénéité basée sur l'écart-type des pixels dans la région rectangulaire définie.

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

Exercice 1.5 – Proposez un algorithme récursif permettant de reproduire la stratégie quadripartition et remplissant les régions homogènes avec une même couleur. La récursion permettra de réaliser la quadripartition sans faire appel à une structure arbre. La récursivité est suffisante pour cette étape qui permet d'obtenir des régions homogènes qui seraient des feuilles d'un arbre quaternaire.

3 Création d'arbre explicite et parcours

Dans la section précédente nous avions créé une structure de données d'arbre de manière implicite au fil des appels récursifs. Nous souhaitons désormais expliciter cette structure de données afin de permettre de nouveaux types de parcours, ainsi que de calculer des statistiques dessus (ex : plus long chemin racine/feuille, ..). Vous devez désormais créer votre propre structure de données d'arbre (sous forme de classe).

Exercice 2.1 – Créez une structure d'arbre explicite qui réalise une quadripartition récursive de l'image. Vous modifierez votre fonction de parcours récursif pour renvoyer la racine de l'arbre et ne plus peindre directement l'image.

Exercice 2.2 – Écrire une fonction récursive qui parcourt l'arbre et pour chaque feuille peint la région couverte avec sa couleur.

Exercice 2.3 – Proposez une fonction récursive qui parcourt l'arbre et pour chaque feuille peint la région couverte d'une couleur proportionnelle à sa profondeur dans l'arbre.

Exercice 2.4 – 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 d'homogénéisation. 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 :

$$PSNR = 20 \cdot log_{10}(255) - 10 \cdot log_{10}\left(\frac{EQ}{3 \cdot W \cdot H}\right)$$

$$EQ = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} \left[(I_r[x,y] - I_r^0[x,y])^2 + (I_g[x,y] - I_g^0[x,y])^2 + (I_b[x,y] - I_b^0[x,y])^2\right]$$

Exercice 2.5 – Écrire une fonction récursive qui parcourt l'arbre et pour chaque feuille imprime une ligne avec sa région couverte et sa couleur, dans n'importe quel ordre, au format `x y width height red green blue`. Cette fonction sera utilisée pour la validation en ligne. Par exemple, pour une feuille couvrant en rouge un carré de taille 4x4 au coin en haut à gauche (0, 0) de l'image, la ligne à imprimer serait `0 0 4 4 255 0 0`.

Exercice 2.6 – Pour comparer différents formats de compression ou différents encodeurs pour un même format, on impose généralement une même taille de fichier (ex. 1Mo, 100Ko, 10Ko) avant de les classer par PSNR. Ici on souhaite comparer différentes quadripartition sur un maximum de 10000 feuilles de l'arbre. Trouvez un seuil d'homogénéité qui fasse que votre quadripartition génère environ 10000 *feuilles* (sans compter les noeuds non terminaux). Vous pourrez évaluer votre programme en ligne sur :

http://eclope.mi90.ec-lyon.fr/problem/tc1td3x1

4 Optimisation de la compression

L'objectif de cette section est d'apporter de nouveaux cas d'étude à votre code. Pour la validation en ligne sur la plateforme de tests, celui-ci risque de devenir excessivement lent. Nous vous recommandons d'optimiser au préalable votre programme, en stockant les données sur les nœuds pour ne jamais les calculer deux fois (ex. moyenne des pixels, erreur quadratique).

Exercice 3.1 – Toutes les zones d'une image ne se prêtent pas nécessairement à une quadripartition en couleurs. En pratique les formats vidéo modernes définissent des *types de nœuds terminaux* qui étendent les possibilités de partitions. Nous nous basons sur le format AV1 (https://en.wikipedia.org/wiki/AV1), qui définit 9 types de nœuds terminaux (le dixième en rouge étant la quadripartition récursive).

Chacun de ces types divise le nœud terminal en rectangles auxquels il faut associer une couleur. Modifiez votre programme pour stocker le type de partition et les couleurs sur un nœud.

Exercice 3.2 – Écrire une fonction récursive qui imprime pour chaque feuille son type (selon la figure ci-dessus), sa région couverte et la liste de ses couleurs (de haut en bas, de gauche à droite). Par exemple, pour une feuille de type 6 couvrant un carré de taille 4x4 au coin en haut à gauche (0,0) avec un carré 2x2 rouge en haut à gauche, un carré 2x2 vert en bas à gauche et un rectangle 2x4 bleu à droite, la ligne à imprimer serait `6 0 0 4 4 255 0 0 0 255 0 0 0 255`.

Exercice 3.3 – Écrire une fonction récursive qui réalise une quadripartition de l'image en prenant en compte ces nouveaux types de partitions (tout en limitant le nombre de nœuds terminaux à 10000). Vous pourrez évaluer votre programme en ligne sur :

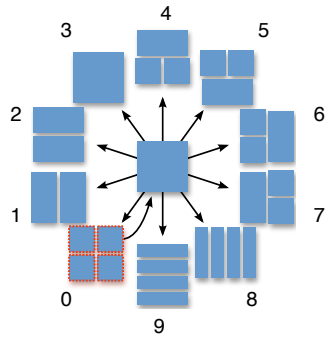


Figure 4: 9 types de partitions terminales d'un nœud, plus la quadripartition récursive (en rouge)

http://eclope.mi90.ec-lyon.fr/problem/tc1td3x2

Exercice 3.4 – Le PSNR 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 une mesure de distorsion qui évalue la *qualité visuelle* de l'image, en donnant plus d'importance aux caractéristiques auxquelles l'œil est le plus sensible.

Exercice 3.5 – Il convient maintenant de choisir une stratégie de quadripartition qui maximise votre mesure de distortion. Proposez de nouveaux critères d'homogénéisation et comparez leurs résultats pour différents seuils d'homogénéité, mesurés selon le PSNR et votre propre mesure de distorsion. Vous représenterez ces résultats sur un graphe, et discuterez des forces et faiblesses de chaque critère proposé.

Annexe 1 : Modalité et calendrier de rendu par groupe

Le rendu sera à déposer sur Moodle 2 semaines après le dernier TD. Ce rendu devra comporter :

- Un code fonctionnel et les tests appropriés
- Des tests avec plusieurs types d'images démontrant l'efficacité de votre approche. Vous êtes invité à fournir les images de test et leur résultat.
- Un rapport qui comprend :
  - Les réponses aux questions
  - La description de parties de code difficiles
  - Tout soucis ou point bloquant dans votre code
  - Les diagrammes nécessaires
  - Un discussion sur les différents tests, en particulier votre choix de valeur de seuil
- Bonus : la partie bonus et la réponse aux questions sur la plateforme de test

Créez une archive (zip, rar, etc.) nommée `nom1-nom2-inf-tc1-td3.zip` que vous transmettez via Moodle.

Annexe 2 : Rappel d'utilisation de la plateforme de tests

Une plateforme de test de code Python est accessible à l'adresse suivante :

http://eclope.mi90.ec-lyon.fr/

La plateforme fonctionne comme indiqué ci-dessous. Attention ! Il faudra modifier la manière dont vous lisez les données en entrée !

Étape 1 : Ouvrez la page web du site ci-dessus dans un navigateur et identifiez vous :

Login : votre nom CAS (1ere lettre du prénom + 7 premières lettres du nom de famille)
Pass : identique au nom CAS (mot de passe provisoire)

Étape 2 : Actions importantes à réaliser sur votre profil
→ https://eclope.mi90.ec-lyon.fr/edit/profile/
1) changez votre mot de passe.
2) sélectionnez votre groupe de TD dans la liste des organisations.
Sinon vous n'apparaitrez pas dans la liste des étudiants
→ exemple pour `d1a` https://eclope.mi90.ec-lyon.fr/organization/2-D1a

Étape 3 : Cliquez sur l'onglet problème pour obtenir la liste des questions de ce TD
→ https://eclope.mi90.ec-lyon.fr/problems/

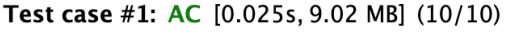
Étape 4 : Sélectionnez une question par exemple un test d'addition
→ https://eclope.mi90.ec-lyon.fr/problem/aplusb

Étape 5 : Cliquez sur `Submit solution` et copiez-collez votre code dans le champ texte
→ https://eclope.mi90.ec-lyon.fr/problem/aplusb/submit

Étape 6 : Selectionnez Python 3 et enfin cliquez sur `Submit` et le serveur exécute votre code avec différents jeux de données (via `input()`)

Étape 7 : Le serveur compare les résultats de votre code (produits par `print()`) à ceux attendus

Étape 8 : Une page affiche `Excecution Results` avec le bouton "Abort" il faut rafraichir cette page ! Votre code est correct s'il passe tous les tests et votre score est affiché.



Étape 9 : Comparez votre performance avec les autres étudiants de la promo ou de votre groupe
→ https://eclope.mi90.ec-lyon.fr/problem/aplusb/rank/