# L'algorithme quadripartition appliqué pour la segmentation de l'image

CPE

Mai 2020

(Version élèves)

ASG

# I Segmentation

## I-*A* *Description du problème*

La segmentation d'image est le processus qui consiste à découper une image en régions connexes présentant une homogénéité selon un certain critère, comme par exemple la couleur, la texture, la profondeur, le mouvement, etc. L'image initiale est retrouvée par l'union de ces régions.

La segmentation est une étape importante dans l'extraction des informations qualitatives d'une image. Elle apporte une description de haut niveau sémantique. Les régions voisines sont connectées à travers un graphe ou 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 (réduite et) représentée par un graphe où des noeuds étiquetés contiennent presque toutes les informations utile au système. Les arcs de ce graphe peuvent être étiquetés (porter une valeur) 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 groupés en trois grandes catégories :

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

1. La première catégorie utilise souvent les histogrammes de l'image. Par seuillage, les différentes couleurs représentatives de l'image sont identifiée et l'algorithme construit ainsi des classes de couleurs qui sont ensuite projetées sur l'image.

2. La deuxième catégorie correspond aux algorithmes d'accroissement de régions ou de partitionnement de régions. La première de deux 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. La seconde 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 dans ce document.

3. 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 proche des techniques d'accroissement de régions fonctionnant sur la base des pixels sont en général limitées pour traiter des images complexes et bruitées.

# I-*B* *Approche du type "split and merge"*

L'algorithme *split and merge* a été propose par *Pavlidis* et *Horowitz* en 1974. Selon cet algorithme, deux étapes de *split*, partitionnement, et de *merge* (fusion), sont opérées.

On traitera des images en couleur. Chaque pixel d'une image en couleur est représenté par trois intensités en rouge, vert et bleu (dit *RGB*). Chaque intensité est codée sur un octet, sa valeur varie de 0 à 255.

## I-B-1 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.

Soit une région $R$ (toute ou une partie de l'image). Si $R$ vérifie un certain critère d'homogénéité de couleur (voir ci-dessous), l'algorithme appliquera la moyenne des couleurs de $R$ à tous ses pixels. Sinon, on découpe $R$ en quatre parties de la même taille et on applique récursivement cette procédure à chacune des quatre parties. La région initiale $R$ est considérée comme un noeud dans un graphe et les sous-régions de $R$ (si découpage en 4 de $R$) comme les quatre fils de ce noeud.

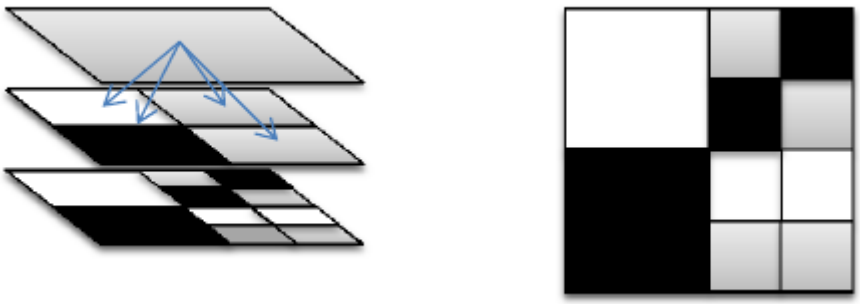


FIGURE 1 – Découpage par quadripartition d'une image 4x4 pixels

La Figure 1 montre une image et le découpage correspondant en plusieurs niveaux. La structure d'arbre associée à ce découpage est illustrée Figure 2.

Le critère d'homogénéité peut-être absolu. Une zone est dite (absolument) 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

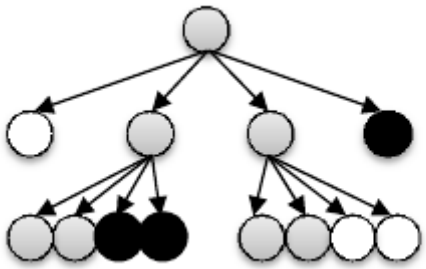


FIGURE 2 – Arbre quaternaire à partir de l'image de la Figure précédente

domine. Pour simplifier, le critère d'homogénéité considéré sera ici à base de l'écart type des couleurs d'une région. Ce critère sera fixé par un seuil.

On utilisera $\alpha$ =*Hétérogénéité* (*1- Homogénéité*). En dessous du seuil $\alpha$, la région est conservée telle quelle et constitue une feuille(noeud terminal) de l'arbre et on lui attribue alors la couleur de la moyenne des pixels la constituant. Au-dessus de ce seuil, la région est découpée en quatre.

## I-B-2 Merge

La procédure de découpage décrite précédemment aboutit à un nombre de régions parfois trop élevé.

La suite du traitement est appelée la phase de fusion (*merge*) et consiste à procéder à une fusion de régions après les découpages de la phase *Split*.

L'implantation la plus simple de la fusion consiste à rassembler les couples de régions adjacentes de couleur proche dans l'arbre issu de la phase *split*.

Différentes techniques existent pou rendre compte de ces adjacences. Par exemple :

- En plus d'un simple graphe qui rend compte des découpages, on peut tenir à jour une liste de voisinages 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 ou *Region Adjacency Graph*. Pour être efficace, ce graphe d'adjacence doit se construire en même temps que l'arbre de découpage. Ensuite, on marquera toutes les régions comme non-traitées, et et on choisit la première région R non traitée disponible.

  Pour une région $R$, les régions en contact avec elle 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 ainsi marquée comme traitée. Une fois la pile vide, l'algorithme choisit la prochaine région marquée comme non traitée et recommence, jusqu'à ce que toutes les régions soient traitées.
- Alternative à cette technique, le *quadtree* crée est un graphe (plus ou moins sophistiqué) codé de telle sorte que l'on puisse trouver facilement les adjacents d'un noeud (voir plus loin pour un implantation efficace).

## I-C *Exemple*

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


FIGURE 3 – Image avant et après la segmentation

L'image (de Lyon) est segmenté et les résultats (ci-dessous) sont obtenus avec différentes valeurs de seuil.


FIGURE 4 – Résultats avec des valeurs de seuil de plus en plus basses augmentant le nombre de régions.

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.

# II Travail de préparation à réaliser

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

• Le morceau de code ci-dessous montre comment ouvrir un fichier image et de charger en mémoire (par *load()*) la matrice de ses pixels.

☞ Si les fonctions de manipulation d'images ne sont pas disponibles, il faudra veiller à installer la librairie *PILLOW*. On sera probablement obligé de travailler avec une machine virtuelle (Virtual Desktop ? installée sur vos postes) ou sur un portable (personnel) pour pouvoir exécuter la commande *pip*).

```
from PIL import Image #Importation de la librairie d'image PIL
from math import sqrt

image_ = Image.open("Image8.bmp") #Ouverture du fichier d'image (voir plus bas pour d'autres formats)

# Importer les données des pixels sous forme d'une matrice px :
matrice_pixels = image_.load()       # chargement des pixels de l'image

# Obtenir la taille de cette image :
width,  height=image_.size

# Afficher l'image à l'écran
image_.show()

# Ecrire une image sur le disque
image_.save("nom_dur_disque.bmp")
```

• Pour manipuler un pixel de coordonnées $x, y$ (soit *matrice_pixels[x,y]*) :

```
# Lire le triplet RGB d'un pixel :
un_pixel=matrice_pixels[x, y]  # un triplet (r, g, b)

# Pour affecter une couleur r, g, b au pixel p[x, y]  :
matrice_pixels[x, y] = r, g, b    #  r, g et b contiennent les valeurs rouge,  vert,  bleu.
# On peut écrire
# matrice_pixels[x, y] = int(r), int(g), int(b)
# pour être sûr de placer des ints.
```

# II-*A* *Split en Python*

Pour respecter l'ordre des fonctions dans Python, on décrit ci-dessous les parties du code. Pour tester ce code, assemblez l'ensemble (dans l'ordre).

- Commençons par quelques fonctions utilitaires de manipulation d'une image chargé en mémoire :

```
# -*- coding: utf-8 -*-
"""
Split d'une image (Mono-Process)"""

from PIL import Image # Importation de la librairie d'image PIL
from math import sqrt # Importation de la fonction sqrt de la librairie math

image_ = None     # sera chargée dans la partie Mian (ici,  on la 'pré-déclare')
width, height = None,  None # respectivement,  la largeur et la hauteur de l'image

def GetPixel(x, y):  return matrice_pixels[x, y]

def PutPixel(x, y, r, g, b): matrice_pixels[x, y]= int(r),  int(g),  int(b) # Il faut des ints !

def PutRegion(x, y, width, height, triplet_color):
    for i in range(x,   x+width):
        for j in range(y,  y+height):
            PutPixel(i, j, triplet_color[0], triplet_color[1], triplet_color[2])
```

- Le calcule de la moyenne des couleurs d'une zone :

```
def Average(corner_x,  corner_y, region_w, region_h):
    sum_red,  sum_green,  sum_blue = 0, 0, 0    #Initialisation des compteurs
    area = region_w*region_h       #Calcul de la superficie de la région

    for i in range(corner_x,  corner_x+region_w):
        for j in range(corner_y,  corner_y+region_h):
            r, g, b=GetPixel(i, j)# Nous lisons les données r, v, b d'un pixel
            sum_red += r       # somme de chaque composant
            sum_green += g
            sum_blue += b
    #Normalisation
    sum_red/=area
    sum_green/=area
    sum_blue/=area

    return(sum_red,  sum_green, sum_blue)   #Retour des valeurs r, g, b moyennes
```

- Calcul de l'écart type des couleurs d'une région de l'image :

  ➛ On utilisera ici une formule alternative plus rapide lorsqu'on a la moyenne. Il s'agit de calculer l'écart type par la formule alternative $\sigma^2 = \frac{\sum x_i^2}{N} - \mu^2$. Penser à *abs()* en appliquant *sqrt()* : ce sera plus efficace.

```
def Mesures_Std_et_Mu(corner_x, corner_y, region_w, region_h):
    av_red, av_blue, av_green = Average(corner_x, corner_y, region_w, region_h)
    sum_red2, sum_green2, sum_blue2 = 0.0, 0.0, 0.0

    for i in range(corner_x, corner_x+region_w):
        for j in range(corner_y, corner_y+region_h):
            red, green, blue = GetPixel(i, j)
            sum_red2 += (red**2)
            sum_green2 += (green**2)
            sum_blue2 += (blue**2)

    area=region_w*region_h*1.0
    r, g, b=0, 0, 0
    r = sqrt(abs(sum_red2 / area - av_red**2))
    g= sqrt(abs(sum_green2 / area - av_green**2))
    b = sqrt(abs(sum_blue2 / area - av_blue**2))
    return ((av_red, av_blue, av_green), (r+b+g)/3.0)
```

- La fonction de découpage en quadrilles.

```
def Decouper_en4(x, y, width, height, threshold_alpha):
    if height*width < 4 : return  # rien à découper

    #Cas de région uniforme : une couleur uniforme est affectée à la partition
    color, rm = Mesures_Std_et_Mu(x, y, width, height)
    if rm < threshold_alpha: #Affectation de la couleur moyenne à la partition
        PutRegion(x, y, width, height, color)
    else: #Dans le cas contraire, la partition non-uniforme est coupée en 4 (récursivement)
        Decouper_en4(x, y, width//2, height//2, threshold_alpha)
        Decouper_en4(x + width//2, y, width//2, height//2, threshold_alpha)
        Decouper_en4(x, y + height//2, width//2, height//2, threshold_alpha)
        Decouper_en4(x+width//2, y+height//2, width//2, height//2, threshold_alpha)
```

- Et la partie principale :

```
#-------------------- Principale -------------------------
if __name__ == '__main__':
    # nom_fic_image="Image_Lyon.bmp"

    dir_image="Images"
    nom_fic_image="steve.png"
    nom_fic_in=dir_image+'/'+nom_fic_image

    try :
        image_ = Image.open(nom_fic_in).convert("RGB") #  nécessaire pour une image "png"
    except :
        print("Problème avec le fichier ",nom_fic_in)
        quit(1)

    matrice_pixels = image_.load()  # Importation des pixels de l'image
    width, height=image_.size

    image_.show()  # Montrez l'image originale
```

```
  Decouper_en4(0, 0, width, height, 15) # tester avec les seuils différents 3,  10,  15,  20,  ...

  image_.show()

  # On sauvegarde le résultat
  nom_fic_out=dir_image+'/'+"out_"+nom_fic_image   # On construit le nom de l'image sauvegard
ée
  image_.save(nom_fic_out)
```

# III Travail à réaliser

1. Mettre en place et tester la version mono-processus. Vérifier qu'elle fonctionne !
2. Écrire une version multi-Processus de cette méthode.

   Indication : dans la fonction **decouper_En4**, s'il faut découper une région, affecter 3 processus à 3/4 de de la région. A chaque fois, sous-traiter 3 des 4 quarts et faire faire la 4e par la fonction elle-même. Cela éviter de conserver un processus qui se contente de sous-traiter seulement !

   ☞ Surveiller le nombre total de processus crées. Ne dépassez pas 12 ! Le mieux serait de moduler ce nombre par la taille de l'image traité.

   ☞ Par gagner en performances, on évitera d'utiliser *array*, *Array*, listes, etc. Utilisez *SharedArray* (SHM), *Pipe* ou *Queue*.
3. Comparez les temps (version Mono et Multi-Processus).
4. Pour la 2e étape (marge ou fusion des régions proches), il faudra créer un arbre quaternaire de voisinage.

# IV Tableau comparatif indicatif

• Un exemple de comparaison des performances par différentes méthodes de parallélisation (temps en **millisecondes**).

➛ On constate un gain de facteur 2 avec *submit* (dernière colonne).

| Fichier Image | Taille Image | Mono (ms.) | ThreadPool | (Multi) Process | Submit+SHM |
|---|---|---|---|---|---|
| steve.png | 40392 | 3232 | | | 1695 (NbProcessus=8)<br>1382 (NbProcessus=4) |
| trump.png | 614400 | 76304 | 43845 | 45501 | 20930 (NbProcessus=8)<br>24071 (NbProcessus=4)<br>25205 (NbProcessus=3) |
| … | … | … | … | … | … |

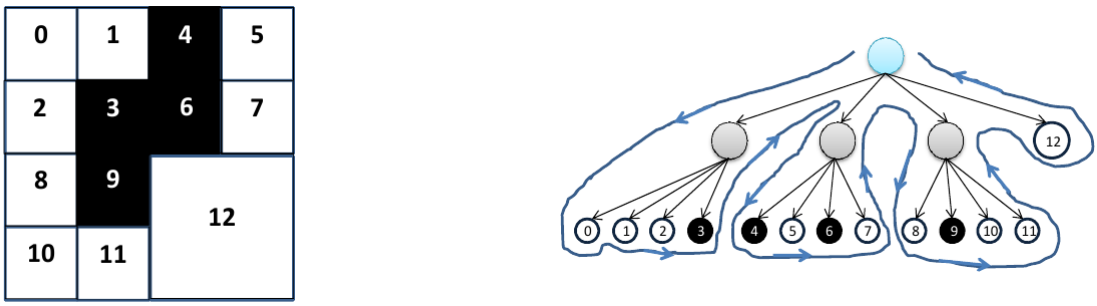
# V Seconde étape : partie merge

La partie fusion travaille sur une liste de régions (par exemple obtenue par l'algorithme de *split*). Il doit être capable de connaître les régions adjacentes (les voisines) de chaque région.

Pour ce faire, supposons disposer des régions sous forme d'un arbre. Cet arbre rassemble dans une structure de données les données des régions monoblocs de couleur homogène de l'image, obtenues à l'étape de *split*.

**Exemple :**

Dans la figure ci-dessous, à gauche, on a une image simple constituée de 16 pixels. A droit, on a l'arbre parcouru en appliquant la stratégie quadripartition.




Soit (les patchs de) l'image à gauche dans cette figure. En suivant la stratégie de quadripartition, nous parcourons en profondeur un arbre quaternaire dont les feuilles correspondent aux régions homogènes. Ces feuilles sont numérotées dans l'ordre de leur visite permettant ainsi de numéroter les régions correspondantes de l'image dont on retiendra pour chacune d'elle, par exemple, les coordonnées du coin gauche supérieur, sa largeur et sa hauteur dans notre cas, ainsi que sa couleur.

## V-A *Une approche possible*

Il faudra modifier notre version parallèle de *split* pour apporter cette fonctionnalité. A l'issu de cette étape de la réalisation, notre programme doit pouvoir retourner une structure (une liste par exemple appelée L_regions) de ces régions (mais auparavant, lire la suite).

Une fois les régions homogènes listées, nous devons retrouver pour chacune d'elle ses voisines, les régions adjacentes, et les placer dans une structure adéquate.

On pourra proposer un graphe (arbre) où deux noeuds sont reliés s'ils sont voisins. Chaque noeud représente une région.

☞ Nous allons avoir besoin de connaître les régions et leurs voisins. Deux solutions de principe se présentent à nous

1- Demander à *split* de stocker les régions (les patchs découpées) dans une liste. On s'occupera de connaître les voisins d'une région plus tard (en phase de fusion).

2- Demander à *split*, lorsqu'on décide de découper une région en 4, de conserver les liens de parenté entre une région et ses 4 fils ainsi que les liens de fratrie entre les 4.

## V-*B* *Première méthode : peu efficace*

On se place dans le cas 1 (ci-dessus) et rappelons qu'on a demandé à *split* de stocker la liste des régions (soit *L_regions*).

Une solution simple et pratique pour représenter la relation d'adjacence des régions de l'image est d'utiliser un graphe. Chaque région apparaît comme noeud de ce graphe. Les liens entre les noeuds expriment la relation adjacence. Ce graphe peut être implanté par une Liste de Régions Adjacentes. Dans cette liste, [0 1 3] signifie que les régions (noeuds) 1 et 3 sont adjacentes à la région 0.

• Proposer un algorithme qui prend en entrée la liste *L_regions* et un indice *ind* d'une région dans cette liste et construit les une liste de régions limitrophes de la région *ind*.

L'étape suivante consiste à étendre le travail précédent pour réaliser, pour toute région, une liste des régions adjacente à celle-ci. On appellera cette liste (*RAL : Ragion Adjacency List*). Pour la figure (gauche) ci-dessus, on peut avoir (approche naïve mais simple) :

RAL = [[0 1 3][1 0 3][2 4 5][3 0 1][4 2 3 5] [5 2 4]]

L'étape suivante sera la réalisation de la fusion. Ici, la condition pour fusionner deux régions est qu'ils soient adjacentes et de même couleur, plus ou moins.

Nous partons d'une RAL et l'algorithme marque toutes les régions comme non-traitées. On choisit ensuite la première région *R* non traitée disponible. Les régions en contact avec *R* sont examinées les unes après les autres pour savoir si elles doivent fusionner avec *R* (en comparant leurs couleurs par exemple). Si c'est le cas, la couleur moyenne de *R* est mise à jour et les régions en contact avec la région fusionnée sont à leur tour comparées avec *R*. La région fusionnée est marquée comme traitée. Quand il n'y aura plus de régions en contact avec *R* et de couleur proche, l'algorithme choisi la prochaine région non-traitée et recommence, jusqu'à ce que toutes les régions soient traitées.

# V-C *Deuxième méthode : arbre plus sophistiqué*

Comme indiqué dans l'alternative 2 ci-dessus, ce sera pendant la phase *split* que les régions et leurs adjacentes seront construite.

Dans la méthode expliquée ci-dessous, l'arbre (le graphe) construit pendant *split* contient en lui-même toutes les informations sur une région et ses adjacents possibles.

La proposition suivante évite l'utilisation d'une pile des régions voisines car elles sont directement accessible.

Une alternative à l'arbre ci-dessous est d'utiliser une forêt d'arbres (comme dans un dictionnaire).

Cet arbre (structure abstraite) aura une représentation physique sous forme d'une liste ou tableau Python.

Dans cette liste, on utilisera les indices pour retrouver les 4 fils d'une même région (comme c'est le cas dans la représentation par tableaux des TAS et des ABOHs)

◦ La liste contiendra tous les noeuds (régions),

◦ La racine sera à l'indice 0,

◦ Pour un noeud d'indice $i$,

Le premier fils est à l'indice $4i + 1$, le 2e à l'indice $4i + 2$, le 3e à $4i + 3$ et le dernier à $4i + 4$.

◦ Pour retrouver le parent (`indice_pere`) d'un noeud d'indice *ind* :

```
indice_pere=ind // 4
if ind % 4 == 0 : indice_pere -= 1
```

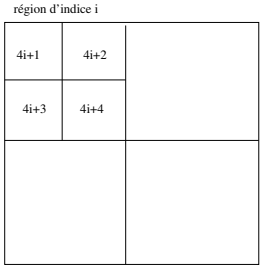
◦ Un noeud représente une région et si cette région est découpée (en 4), ses 4 fils contiendront le quadruplet. Le noeud parent est nécessaire pour la phase de fusion.

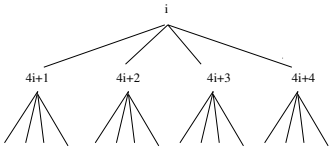
◦ Les informations stockées dans chaque noeud seront (le tuple) $(x, y, width, height, \sigma, \mu, merged)$ où $\mu = (r, v, b)$ est la (moyenne de) couleur de cette région.

➛ Le booléen *merged* initialisé à *False* recevra *True* si la région a déjà fait l'objet d'une fusion.

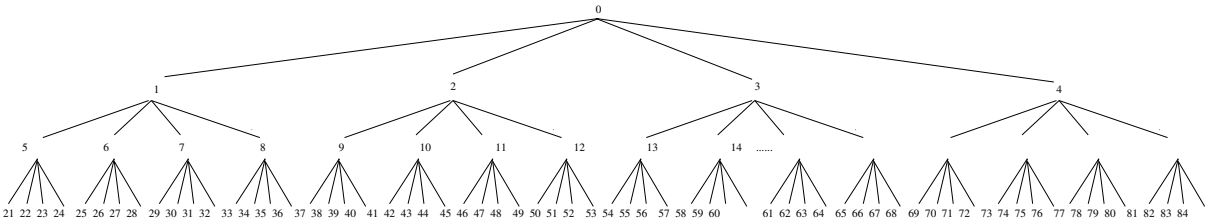
➛ La valeur de $\sigma$ peut-être utilisée pour décider de fusionner cette région ou pas

Par exemple (arbre partiel) :



Un exemple avec les indices :




## V-*D* *Une Implantation différente de l'arbre quaternaire*

L'implantation[1] de l'arbre quaternaire par un tableau / liste plate permet une utilisation facile mais peu efficace en termes d'espace mémoire. Cette inefficacité est dûe aux noeuds (régions d'image) des niveaux supérieurs qui n'ont pas eu besoin d'être découpées. Ces noeuds crée des "trous" dans l'arbre et il est nécessaire de prévoir des emplacements pour leurs descendants, comme si la région avait été découpée. Il les faut si on doit maintenir le principe d'accès (par $4i + j$) aux fils.

On peut envisager une implantation de ces mêmes arbres par une structure dispersée qui évite ces gaspillages en espace mémoire.

☞ Les détails de cette implantation en annexes.

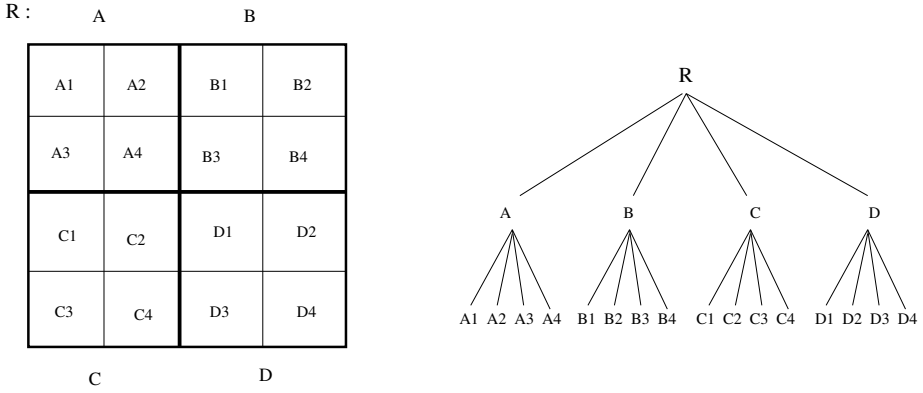
## V-*E* *Détails d'une fusion*

Soit un noeud (une région) découpée en *A,B,C,D*, lesquelles sont redécoupées en 4.

Pour un noeud (région) $A$ :

◦ Fusionner les files de $A$ : fusion des **frères** $A1, A2, A3, A4$ (voir *frères* ci-dessous)

◦ Remonter au père de $A$ (ici $R$, on a vu ci-dessus comment remonter au père à partir d'un indice) pour traiter les **neveux** dans les 4 familles A,B,C,D (voir *neveux* ci-dessous).




1. Ou en Anglais "implementation" que l'on voit utilisé par abus en Fraçais

Bien entendu, les fusion des couleurs se feront par rapport à un seuil fixé. Voir la section remarques plus loin pour les critères de fusion et ses conséquences.

Parcours en **profondeur de gauche à droite** : on fusionne au niveau le plus bas avant de remonter.

Soit la région R (un noeud) découpée en $A, B, C, D$

0. merge(R) :

1. **merge les frères** dans $R$ :
   (a) **merge les neveux dans** $R$ :
   - ➛ Entre $A$ et $B$ : considérer $(A2 \times B1)$, $(A4 \times B3)$
   - ➛ Entre $A$ et $C$ : considérer $(A3 \times C1)$, $(A4 \times C2)$
   - ➛ Entre $B$ et $D$ : considérer $(B3 \times D1)$, $(B4 \times D2)$
   - ➛ Entre $C$ et $D$ : considérer $(C2 \times D1)$, $(C4 \times D3)$

   (b) On fusionne maintenant entre les 4 régions $A, B, C, D$ (sont des frères dans $R$ = fils de $R$) :
   $A \times B$, $A \times C$, $B \times D$ et $C \times D$.

On peut décider, alternativement, de fusionner d'abord ce qui peut l'être au sein de $A$ : entre $A1, A2, A3, A4$. Ceci se fera par un appel récursif à $merge(A)$ donne lieu à une simple fusion potentielle entre $A1, A2, A3, A4$. Ensuite, on va examiner les neveux de $A$ (qui se trouvent dans $B$ et $C$). Enfin, on s'attaquera à la fusion des 4 région dans $R$. Cette fusion entre $A, B, C, D$ sera traitée au niveau du père ($R$), dans les appels récursifs.

☞ Pour fusionner les frères, dans chaque quadruplet de frères, prendre le première fils (le cadet) comme repère et envisager les fusions dans l'ordre $1, 2, 3, 4$ ci-dessus pour éviter de re-fusionner deux patchs déjà traités.

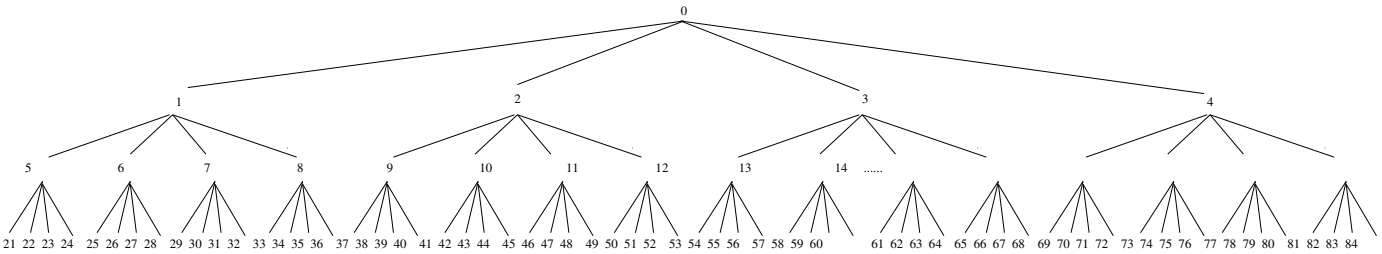
**Trace :**

```
0 : merge(R) :
 1 : merge_fils_de(R) :
  2 : merge(A) :
   3 : merge_fils_de(A) :
    4 : marge(A1) :--> <-- rien à faire
    4 : marge(A2) :--> <--
    4 : marge(A3) :--> <--
    4 : marge(A4) :--> <--
   3 : merge_freres_dans(A) :
    4 : merge_neveux_dans(A) : --> <--
    4 : Envisager la fusion de A1 x A2, A1 x A3, A2 x A4, A3 x A4
  2 : merge(B) :
   3 : merge_fils_de(B) :
    4 : marge(B1) : --> <-- rien à faire
    4 : marge(B2) : --> <--
    4 : marge(B3) : --> <--
    4 : marge(B4) : --> <--
   3 : merge_freres_dans(B) :
    4 : merge_neveux_dans(B) : --> <--
    4 : Envisager la fusion de B1 x B2, B1 x B3, B2 x B4, B3 x B4
  2 : merge(C) :
```

```
    3 : merge_fils_de(C) :
      4 : marge(C1) : --> <--
      4 : marge(C2) : --> <--
      4 : marge(C3) : --> <--
      4 : marge(C4) : --> <--
    3 : merge_freres_dans(C) :
     4 : merge_neveux_dans(C) :  --> <--
     4 : Envisager la fusion de C1 x C2, C1 x C3, C2 x C4, C3 x C4
  2 : merge(D) :
    3 : merge_fils_de(D) :
      4 : marge(D1) : --> <--
      4 : marge(D2) : --> <--
      4 : marge(D3) : --> <--
      4 : marge(D4) : --> <--
    3 : merge_freres_dans(D) :
     4 : merge_neveux_dans(D) :  --> <--
     4 : Envisager la fusion de D1 x D2, D1 x D3, D2 x D4, D3 x D4
1 : merge_freres_dans(R) :
  2 : merge_neveux_dans(R) :
    3 : Envisages les 8 paires de neveux dans R
  2 : Envisager merge entre A, B, C et D
```

## V-*F* *D'autres exemples*

**Exemple 1 :**




La trace en annexe est celle d'un parcours en profondeur et de droite à gauche de l'arbre ci-dessus associé à la petite image (et le résultat de l'exécution) :

Il est plus cohérent de procéder à un parcours en profondeur mais de droite à gauche.

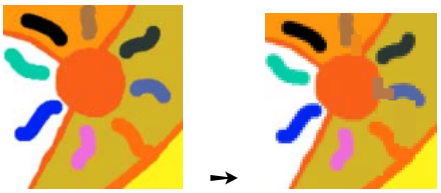
La démarche est strictement identique mais le bon ordre (dans le code) est de droite à gauche car pour traiter les frères, puisqu'on s'appuie sur le frère cadet (qui est traité en premier dans un parcours de gauche à droite), il faudra que les frères aînés soient traités en premier avant de fusionner l'ensemble des frères.

Il n'y a pas d'erreur à traiter de gauche à droite mais il est plus "logique" de traiter les fils / frères de droite à gauche pour fusionner de plus grands régions qui sont déjà fusionnées par petits bouts.

**Exemple 2** :

**Exemple 3 :**

A gauche, l'image d'origine et à droite le résultat de la fusion (seuil de *split=merge=3*)



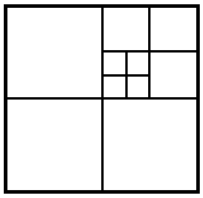
# VI Travail à réaliser

- Compléter *Split* en y ajoutant le stockage des régions.
- Réaliser le code Python parallèle de Fusion de ces régions.

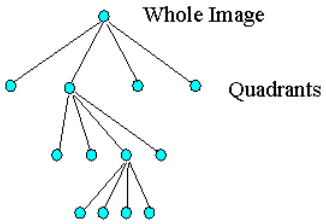
# VII Aller plus loin

## VII-*A* *Remarques*

L'arbre utilisé ne sera pas forcément équilibré et compact comme le montre la figure d'exemple ci-dessous (l'arbre correspond au découpage actuel de l'image à gauche) car pour un noeud donné, le *split* n'a lieu que si la région est considérée comme non homogène.



IMAGE




QUADTREE

Etant donnée la structure de l'arbre (représenté par une liste Python), les "trous" dans la liste contiendront *None*.

Ce que l'on peut reprocher à cet arbre creuse et déséquilibré dans sa représentation plate (présentée ci-dessus) est la présence obligatoire de trous (pour les régions non découpées, les noeuds sans fils) qui en font une structure par endroit creuse.

Cependant, avec une représentation par pointeurs (dans un autre langage de programmation), l'équivalent d'une *absence* décrite par *None* en Python (par exemple NULL en C/C++/JAva, NIL en Pascal, etc) coûterait autant. Néanmoins, plus cet arbre sera compact et équilibré, moins il y aura des trous et la mémoire sera mieux utilisée.

Une autre alternative sera de représentée l'arbre par une matrice d'adjacence. d'ordre est les quadruples (appartenance ensembliste).

## VII-*B* *Critères de fusion*

Les critères de fusion de 2 régions $R1$ et $R2$ (avec respectivement $color_1, \mu_1, \sigma_1$ et $color_2, \mu_2, \sigma_2$) peuvent être diverses :

◦ D'une manière générale : fusion si $|\mathcal{N}(\mu_1, \sigma_1) - \mathcal{N}(\mu_2, \sigma_2)| < seuil$ où $\mathcal{N}(\mu_1, \sigma_1)$ est la distribution *Normale* des couleurs de $R_1$ (resp. $R_2$).

Cette différence peut se décliner de différentes manières (simples et surtout rapide) :

◦ Fusion si $|\mu_1 - \mu_2| < seuil$

◦ Fusion si $|\sigma_1 - \sigma_2| = 0$ (ou $\simeq 0$)

◦ Fusion si $\sqrt{|\sigma_1^2 - \sigma_2^2|} = 0$ (ou $\simeq 0$)

◦ Fusion si $|\mu_1 - \mu_2| + \sqrt{|\sigma_1^2 - \sigma_2^2|} = 0$ (ou $\simeq 0$)

◦ Fusion si $\frac{|\mu_1 - \mu_2|}{\sqrt{|\sigma_1^2 + \sigma_2^2|}} < seuil$

◦ Etc.

## VII-C *Action de fusion et ses conséquences*

Diverses méthodes (dont le résultat sera affecté aux 2 régions fusionnées) :

◦ Opérer un bitwise-OR (ou un bitwise-AND, bitwise-XOR) sur $color_1$ et $color_2$

◦ Attribuer la moyenne de $color_1$ et $color_2$ aux 2 régions.

◦ La région englobante les deux fusionnées (avec $\mathcal{N}(\mu_1, \sigma_1)$ et $\mathcal{N}(\mu_2, \sigma_2)$) aura une moyenne de couleurs $\mu = \sum x_i$ et une variance $\sigma^2 = \frac{\sigma_1^2+\sigma_2^2+\mu_1^2+\mu_2^2}{2} - \mu^2$

◦ Etc.

☞ Si une des régions a déjà fait l'objet d'une fusion (l'attribut *merged=True*), alors

◦ propager sa couleur aux deux,

◦ Refaire une moyenne et propager aux régions en contact avec l'une des deux régions,

◦ etc.

## VII-D *Encore plus loin !*

La compression d'image et de vidéo fait appel à de nombreux algorithmes permettant de compresser les données des pixels dans une image.

On parle alors de la *compression spatiale*, y compris pour des images successives d'une séquence d'image. Dans ce dernier cas, on parle de la *compression spatio-temporelle*.

Ces compressions se font en supprimant les données redondantes dans une image, voire des images successives dans une séquence.

Pour cela, la première norme de compression, MPEG1, considère une image divisée en bloc de 8x8 pixels. Dans un tel bloc, les pixels sont dans la plupart des cas similaires.

Si la norme MPEG1 visait les ordinateurs multimédia, MPEG2 vise un contexte plus large, notamment, la télévision numérique, DVD, mais aussi les formats hautes résolutions.

MPEG4 chamboule l'ordre en proposant une compression par objet. Chaque objet dans une image est segmenté et subit une compression. Il est par ailleurs possible de composer les objets au moment de diffusion.

Dernièrement, le standard HEVC (High Efficiency Video Coding) proposé par ISO/IEC Moving Picture Experts Group (MPEG), marque une nouvelle étape dans la capacité de compression des images animées.

# VIII Annexes

Le code de l'arbre quaternaire (avec tests).

```
from collections import OrderedDict
TRACE1=False

class Quadternaire(OrderedDict):
    def __init__(self) :
        super().__init__()

    #-------------------------
    # redéf de la fonction d'ajout par '[]'
    # Je redéf car je doit dire si la clé existe ?
    def __setitem__(self, ind_node, infos):# poru ajouter des élés avec []
        try : # obligé car l'accès à une clef inexistante provoque une err.
            if self[ind_node] :
                if TRACE1 : print("add_node : la cle", ind_node, "existe déjà !, on la remplace...")
        except : # builtins.KeyError ? si clé n'existe pas
            pass

        super().__setitem__(ind_node, infos)
    #-------------------------
    def __getitem__(self, ind_node):
        return self.get(ind_node, None)

    #-------------------------
    def __repr__(self) :
        self.afficher()
        return "" # __repr__ doit tjs envoyer un str.
    #-------------------------
    def supprimer_entry(self, ind_node) :
        try : # obligé car l'accès à un e clef inexistante provoque une err.
            self.pop(ind_node) # supprime une entrée si clé existe et renvoie sa value
        except : # builtins.KeyError ? si clé n'existe pas
            print("__getitem__ : la cle", ind_node, "n'existe pas ! ...")

    #-------------------------
    def add_node(self, ind_node, infos_node) :
        self[ind_node]=infos_node

    #-------------------------
    def afficher(self, nb_regions_a_afficher=-1) :
        if nb_regions_a_afficher==-1 :
            for cple in self.items() : print(cple)
            return
        nb_ecrits=0
        for cple in self.items() :
            if nb_ecrits==nb_regions_a_afficher : return
            print(cple)
            nb_ecrits+=1
    #-------------------------
    def afficher_largeur(self, profondeur=3) :
        print('0 :',self[0])
        indice_max_a_afficher=0
        for i in range(1,profondeur+1) :
            indice_max_a_afficher+=4**i
        for ind in range(1,indice_max_a_afficher+1) :
            print(ind,':', end='')
            info=self.get(ind, None)
            print(info); input("1000")
            x,y,width,height,(mu_r, mu_g, mu_b),ecart_type,merged_or_not=info
            print("%3d, %3d, %3d, %3d, %3d, %3d, %3d, %.2f" %(x,y,width,height,mu_r, mu_g, mu_b,ecart_type))



    #-------------------------
    def get_infos(self, ind_node) :
        #assert(ind_node in self.keys())
        try : # obligé car l'accès à un e clef inexistante provoque une err.
            return self[ind_node]
        except : # builtins.KeyError ? si clé n'existe pas
            print("__getitem__ : la cle", ind_node, "n'existe pas ! ...")
            return      None
    #-------------------------
    def indices_fils_de_region(self, ind_node) :
        assert(ind_node in self.keys())
        return [get_infos(ind_node*4+1), get_infos(ind_node*4+2), get_infos(ind_node*4+3), get_infos(ind_node*4+4)]
    #-------------------------
    def indices_fils_de_region(self, ind_node) :
        assert(ind_node in self.keys())
        return [ind_node*4+1, ind_node*4+2, ind_node*4+3, ind_node*4+4]
    #-------------------------
    def region_a_des_fils(self, ind_node) : # est-ce que cette région a été découpé
        assert(ind_node in self.keys())
        return ind_node*4+1 in self.keys() # un seul fils suffit pour savoir si cette région est coupée en 4
    #-------------------------
    def get_ind_mon_pere(self, my_ind_node) :
        ind_mon_pere, reste=divmod(my_ind_node,4)
        if reste==0 : ind_mon_pere -= 1
        return ind_mon_pere
    #-------------------------
    # Les frere dans ma propre région (chez moi)
    def indices_de_mes_2_freres_adjecents(self, my_ind_node) :
        # assert(my_ind_node in self.keys())
        if my_ind_node not in self.keys() : return(None, None)
        if my_ind_node==0 : return(None, None)

        ind_mon_pere=self.get_ind_mon_pere(my_ind_node)
        #print("le pere de", my_ind_node, "est", ind_mon_pere)
        if my_ind_node==ind_mon_pere*4+1  : return (my_ind_node+1, my_ind_node+3)
        if my_ind_node==ind_mon_pere*4+2 : return (my_ind_node-1, my_ind_node+1)
        if my_ind_node==ind_mon_pere*4+3: return (my_ind_node-1, my_ind_node+1)
```

```
        if my_ind_node==ind_mon_pere*4+4: return (my_ind_node-1, my_ind_node-3)
        raise ValueError

        #infos_du_pere = self.get_infos(ind_mon_pere)
    #-------------------------
    def indices_de_mes_neveux(self, my_ind_node) :
        assert(my_ind_node in self.keys())
        if my_ind_node==0 : return(None, None)
        ind_mon_pere=self.get_ind_mon_pere(my_ind_node)

        mon_oncle1, mon_oncle2=self.indices_de_mes_2_freres_adjecents(ind_mon_pere)
    #-------------------------
    def indices_de_mes_neveux_adjecents(self, my_ind_node) :
        pass

#-----------------------------------------------------------------------
def test_simple_du_dico() :
    print("Go")
    arbre=Quadternaire()
    arbre.add_node(0, [0,2,"a", "b"])
    arbre.afficher()
    print('-'*40)
    arbre.add_node(5, [5,2,"cc", "b"])
    arbre.add_node(3, [3,2,"cc", "b"])
    arbre.afficher()
    print('-'*40)

def tester_arbre_creer_regions_fictives() :
    global qt_Arbre
    arbre=Quadternaire()

    def ajouter_au_dico_4_fils_pour_un_noeud(indice) :
        for i in range(1,5) :
            arbre.add_node(indice*4+i, [indice*4+i,"x", "y", "w","h", "mu", "sigma"])

    arbre.add_node(0, [0,"x", "y", "w","h", "mu", "sigma"])


    ajouter_au_dico_4_fils_pour_un_noeud(0)

    ajouter_au_dico_4_fils_pour_un_noeud(1)
    ajouter_au_dico_4_fils_pour_un_noeud(6)
    ajouter_au_dico_4_fils_pour_un_noeud(27)
    ajouter_au_dico_4_fils_pour_un_noeud(7) # mis après "27" pour tester l'ordre !

    arbre.afficher()
    print('-'*40)

    # On test différentes fonctions
    print("indices_de_mes_2_freres_adjecents(5) :", arbre.indices_de_mes_2_freres_adjecents(5))
    print("indices_de_mes_2_freres_adjecents(6) :", arbre.indices_de_mes_2_freres_adjecents(6))
    print("indices_de_mes_2_freres_adjecents(7) :", arbre.indices_de_mes_2_freres_adjecents(7))
    print("indices_de_mes_2_freres_adjecents(8) :", arbre.indices_de_mes_2_freres_adjecents(8))

    print("indices_de_mes_2_freres_adjecents(22) :", arbre.indices_de_mes_2_freres_adjecents(22))
    print("indices_de_mes_2_freres_adjecents(110) :", arbre.indices_de_mes_2_freres_adjecents(110))
    print("indices_de_mes_2_freres_adjecents(0) :", arbre.indices_de_mes_2_freres_adjecents(0))

    #---------------------------------
    # d'autres tests
    #-------------------------
    print('-'*50)
    ind_node=5
    print ("indices potentils des fils de region de ", ind_node, ' : ' ,arbre.indices_fils_de_region(ind_node) )
    print ("est-ce que des fils de la région ont été crées ? ", ind_node, ' : ' , arbre.region_a_des_fils(ind_node) )
    print ("ind du pere de ", ind_node, ' : ' , arbre.get_ind_mon_pere(ind_node))
    #-----------------------------
    print('-'*50)
    ind_node=6
    print ("indices potentils des fils de region de ", ind_node, ' : ' ,arbre.indices_fils_de_region(ind_node) )
    print ("est-ce que des fils de la région ont été crées ? ", ind_node, ' : ' , arbre.region_a_des_fils(ind_node) )
    print ("ind du pere de ", ind_node, ' : ' , arbre.get_ind_mon_pere(ind_node))

if __name__ == "__main__" :
    tester_arbre_creer_regions_fictives()
```

## Trace :

```
(0, [0, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(1, [1, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(2, [2, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(3, [3, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(4, [4, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(5, [5, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(6, [6, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(7, [7, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(8, [8, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(25, [25, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(26, [26, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(27, [27, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(28, [28, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(109, [109, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(110, [110, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(111, [111, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(112, [112, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(29, [29, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(30, [30, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(31, [31, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
(32, [32, 'x', 'y', 'w', 'h', 'mu', 'sigma'])
----------------------------------------
indices_de_mes_2_freres_adjecents(5) : (6, 8)
indices_de_mes_2_freres_adjecents(6) : (5, 7)
indices_de_mes_2_freres_adjecents(7) : (6, 8)
```

```
indices_de_mes_2_freres_adjecents(8) : (7, 5)
indices_de_mes_2_freres_adjecents(22) : (None, None)
indices_de_mes_2_freres_adjecents(110) : (109, 111)
indices_de_mes_2_freres_adjecents(0) : (None, None)
_____________________________________________________
indices potentils des fils de region de  5  :  [21, 22, 23, 24]
est-ce que des fils de la région ont été crées ?  5  :  False
ind du pere de  5  :  1
_____________________________________________________
indices potentils des fils de region de  6  :  [25, 26, 27, 28]
est-ce que des fils de la région ont été crées ?  6  :  True
ind du pere de  6  :  1
```