

# Segmentation d'images

TRIBOULET Corentin – D1b – DELLANDREA

Introduction

Choix et description de l'image

Partie 3: Optimisation de la compression

3.1 Formules reliant les parents aux enfants

3.2 Transformation en arbre quaternaire implicite

Fonctions implémentées

3.3 Comparaison entre les deux formes d'arbres

3.4 Nouveau critère d'homogénéité

3.5 Effets du nouveau critère d'homogénéité

On fait varier seulement le `seuil_lum` en maintenant `seuil_couleur=50`

On fait varier seulement le `seuil_lum` en maintenant `seuil_couleur=255`

## Introduction

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.

## Choix et description de l'image

La partie haute de l'image est principalement colorée en bleu donc un contraste extrêmement faible. Il sera facile de compresser cette partie.

Ensuite, sur le bâtiment il y a des cheminées. Il sera facile encore à comprimer cette partie car il y a peu de couleur différente.

Les fenêtres sont beaucoup plus précises, ainsi pour garder un rendu correct, il faudra sacrifier de la mémoire.



Les feuilles de l'arbre sont fines, difficile de représenter un arbre sans feuille. Il faudra là encore sacrifier de l'espace.

Le logo LYON est principalement, cela ne posera pas de problème pour la compression, de même pour le lion. Ce dernier est cependant peu géométrique, ainsi il sera difficile de garder une forme précise.

L'homme à droite de l'affiche sera aussi bien difficile à garder pendant la compression.

Le bas de l'image est peu contrasté, une seul couleur réside.

## Partie 3: Optimisation de la compression

### 3.1 Formules reliant les parents aux enfants

Grâce aux formules des arbres binaires, on obtient les formules des arbres quaternaires assez facilement:

Soit un parent d'indice  $i$ , alors ces 4 fils seront les éléments d'indices :

$$4i + 1$$

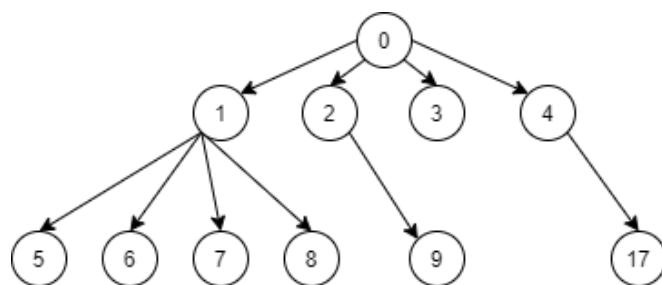
$$4i + 2$$

$$4i + 3$$

$$4i + 4$$

Soit un élément d'indice  $i$  alors son parent sera l'élément d'indice :

$E((i - 1)/4)$  soit en code `int((i-1)/4)`



Exemple:

4 est le parent de 17 car  $E((17 - 1)/4) = E(4) = 4$

9 est le fils de 2 car  $9 = 4 * 2 + 1$

## 3.2 Transformation en arbre quaternaire implicite

L'idée maintenant est de transformer l'arbre pour que l'on puisse accéder aux éléments grâce à leur indice puis pouvoir remonter aux enfants/parent grâce à ces indices. Par la suite, on utilisera un dictionnaire pour enregistrer les données.

### Fonctions implémentées

```
def a_fils(matrice,x,y,l,h,seuil_couleur,seuil_lum):
    a=False
    if h>1 and l>1:
        if est_homogene2(matrice,x,y,l,h,seuil_couleur,seuil_lum)==False :
            a=True
    return(a)
```

La fonction `a_fils` vérifie si un noeud possède des fils

```
def ajout_fils(matrice,dico,indice,seuil_couleur,seuil_lum):
    if indice in dico:
        if a_fils(matrice,*dico[indice],seuil_couleur,seuil_lum):
            hg,hd,bg,bd=partition(*dico[indice])
            dico[4*indice+1]=hg; dico[4*indice+2]=hd; dico[4*indice+3]=bg; dico[4*indice+4]=bd
```

La fonction `ajout_fils` permet de les ajouter dans le cas où le noeud a des enfants

```
def arbre(matrice,seuil_couleur,seuil_lum,x=0,y=0,l=W,h=H):
    dico={0:[x,y,l,h]}
    indice=0
    while indice<=4*len(dico)+4:
        ajout_fils(matrice,dico,indice,seuil_couleur,seuil_lum)
        indice=indice+1
    return(dico)
```

La fonction `arbre` se sert de `ajout_fils` pour créer un arbre sous forme implicite

## 3.3 Comparaison entre les deux formes d'arbres

Arbre explicite:

Arbre implicite:

On décide de bloquer le seuil de luminosité à 255 pour pas qu'il interfère dans la rapidité du programme avec arbre implicite.

```
def trace_memoire_temps():
    Duree=[]
    Seuil=[ ]
```

```
def trace_memoire_temps(matrice):
    Duree=[ ]
    Seuil=[ ]
```

```

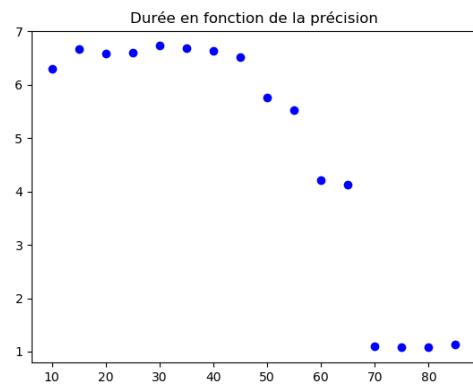
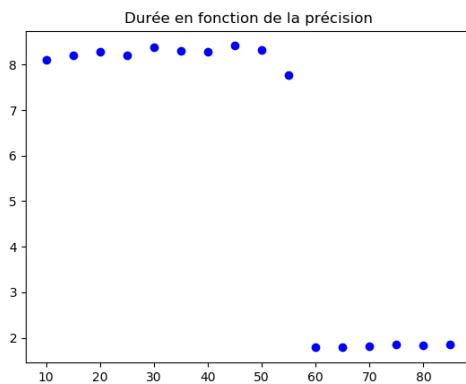
for seuil in range(10,90,5):
    deb=time.time()
    racine = arbre2(0,0,W,H,seuil)
    peindre_arbre(racine)
    fin=time.time()
    Duree.append(fin-deb)
    Seuil.append(seuil)
plt.plot(Seuil,Duree,'bo')
plt.title('Durée en fonction de la précision')
plt.show()

```

```

for seuil in range(10,90,5):
    deb=time.time()
    dictio=arbre(matrice_px,seuil,255)
    fin=time.time()
    Duree.append(fin-deb)
    Seuil.append(seuil)
plt.plot(Seuil,Duree,'bo')
plt.title('Durée en fonction de la précision')
plt.show()

```



On remarque que la version implicite permet de gagner 1,5 s sur la version explicite. Le passage à la version implicite permet donc de **gagner du temps et de l'espace mémoire**.

### 3.4 Nouveau critère d'homogénéité

On se propose dans cette partie d'améliorer notre critère d'homogénéité par l'ajout d'une nouvelle condition sur la valeur de la luminosité (critère que la vision humaine détecte avec précision).

La valeur du niveau de luminosité X sera calculée par:

$$X = 0,2126 * R + 0,7152 * G + 0,0722 * B$$

Avec  $R$ ,  $G$  et  $B$  respectivement les composantes rouge, verte et bleu.

```

def est_homogene2(x,y,w,h,seuil_couleur,seuil_lum):
    ecr, ecg, ecb = ecart_type(x,y,w,h)
    X= 0,2126*ecr + 0,7152*ecg + 0,0722*ecb
    return ecr <= seuil_couleur and ecg <= seuil_couleur and ecb <= seuil_couleur and X<=seuil_lum

```

[cf Epreuve d'informatique et Modélisation de Systèmes Physiques, Concours Banque PT 2021 \(lien\)](#)

### 3.5 Effets du nouveau critère d'homogénéité

On effectue différents tests pour prouver l'efficacité du nouveau critère d'homogénéité.

**On fait varier seulement le `seuil_lum` en maintenant `seuil_couleur=50`**

`seuil_couleur=50; seuil_lum=255`

Le seuil de luminosité n'a pas d'impact sur la compression pour cette image



`seuil_couleur=50; seuil_lum=50`



`seuil_couleur=50; seuil_lum=25`



`seuil_couleur=50; seuil_lum=10`



## Tailles des images

Aa seuil_lum	# Taille de l'image (en Ko)
255	307
50	307.2
25	587
10	781

Les 2 premières images sont quasiment identiques, le `seuil_lum` a peu d'impact entre 50 et 255. Puis lorsqu'il vaut 25 et 10, il permet de compléter le `seuil_couleur`. Il faut trouver le bon compromis pour faire fonctionner les 2 seuils correctement.

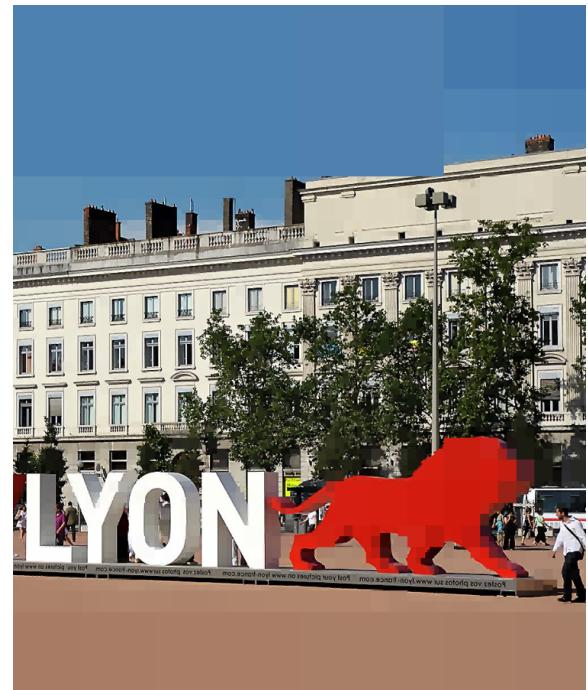
**On fait varier seulement le `seuil_lum` en maintenant `seuil_couleur=255`**

C'est à dire que le seuil de couleur n'a aucun impact sur la compression des prochaines images

`seuil_couleur=255; seuil_lum=50`



`seuil_couleur=255; seuil_lum=25`



`seuil_couleur=255; seuil_lum=10`

`seuil_couleur=255; seuil_lum=5`



### Tailles des images

Aa seuil_lum	# Taille de l'image (en Ko)
<u>50</u>	271.7
<u>25</u>	586.3
<u>10</u>	781
<u>5</u>	847.5

A lui tout seul, le `seuil_lum` permet de faire un beau contour des fenêtres dès la première image `seuil_lum=50` car ce sont des zones de hauts contrastes. Les tons clairs du bas de la photo sont plus dur à disocier à cause du contraste faible. Mais à partir de `seuil_lum=10`, on observe plus beaucoup de différences.