



# ÉCOLE CENTRALE LYON

UE INF TC1  
RAPPORT

## Compression d'image

*Élèves :*

Matteo KOZLOWSKI  
Yohan PELLERIN

*Enseignant :*

Stéphane DERRODE

26 novembre 2021

# 1 Arbre implicite de quadripartition

## 1.1 Arbre quaternaire implicite

Dans un premier temps, il s'agit de stocker l'image quadripartitionnée sous la forme d'une liste de Noeud dont la place de l'élément dans la liste nous renseigne sur sa position dans l'arbre. Ici, l'arbre n'est plus binaire comme au TD2 mais quaternaire, chaque Noeud a 4 fils. Ainsi la position des noeuds dans la liste pourra s'établir comme suit, soit un Noeud indicé  $i$  :

- La racine a la position  $i = 0$
- Le parent a la position  $\lfloor (i - 1)/4 \rfloor$
- Le fils haut-gauche a la position  $4 \times i + 1$
- Le fils haut-droit a la position  $4 \times i + 2$
- Le fils bas-gauche a la position  $4 \times i + 3$
- Le fils bas-droit a la position  $4 \times i + 4$

Pour stocker les régions sous forme de liste, nous avons pensé à remplir celle-ci selon le principe suivant :

- On test si la région est homogène ou s'il sagit d'un None
- Si non, ou si c'est un None, on ajoute 4 None à la liste et on poursuit avec l'élément suivant de la liste
- Autrement, on la divise en 4 et on ajoute ses 4 sous parties à la liste

5	6	9	10
7	8	11	12
13	14	17	18
15	16	19	20

FIGURE 1 – Ordre de remplissage de la liste

La région orange est indicée 1, la verte 2, la bleue 3 et la noire 4. Ensuite, on indice les sous région dans l'ordre indiqué sur la figure.

## 1.2 Problème à l'implémentation

Nous avons créé un programme qui permettait de générer cette liste. Il s'agissait d'une boucle while qui terminait lorsque l'on atteignait le bas de l'abre, c'est à dire lorsque les enfants de tous les noeuds étaient 4 None. Cependant, l'algorithme ne terminait pas en temps raisonnable et de fait, ses performances temporelles étaient trop éloignées du programme récursif. Ainsi, notre idée a été de retirer les None de la liste quitte à modifier

l'indexation des noeuds. La liste ne contient alors que les noeuds correspondant à des régions homogènes.

### 1.3 Compression de l'image et accès aux régions homogènes

Notre but était d'obtenir une image compressée en coloriant les régions homogènes dès leur ajout dans la liste. Ainsi, la fonction `peindre_arbre` est directement créée dans la fonction `arbre` afin qu'à mesure qu'on ajoute les régions homogènes, elles sont peintes et créent l'image finale. Pour ce faire, on peint récursivement les régions obtenues après quadripartitionnement de la région non homogène choisie. Contrairement à l'algorithme implémenté en TD, on ne peint pas l'arbre après qu'il ait été construit mais en même temps qu'une liste contenant les régions homogènes est créée.

### 1.4 Comparaisons

Nous voulons maintenant comparer en terme de vitesse d'exécution et de consommation en mémoire les deux algorithmes qui affichent l'image compressée.

#### 1.4.1 Comparaison temporelle

Un premier élément de comparaison est le temps d'exécution du programme *ie* le temps que met l'image à être générée. Pour cela, nous avons importé le module `time` et la fonction `process_time` afin d'acquérir les instants de lancement et d'arrêt du programme. La performance de la méthode que nous avons voulu quantifier est la méthode `peindre_arbre`. En effet, nous voulions savoir si le stockage de l'arbre sous forme de liste mettait plus de temps à se générer.

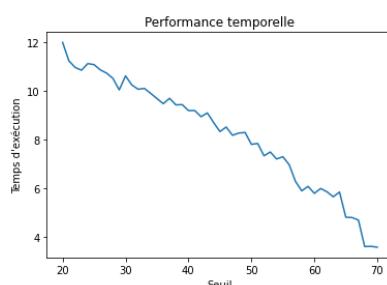


FIGURE 2 – `peindre_arbre` dans le cas du stockage de l'image sous forme d'arbre

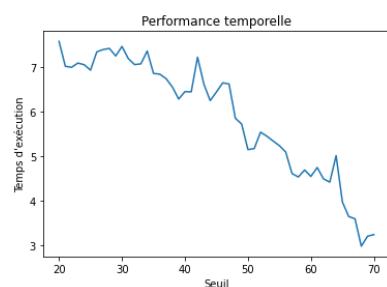


FIGURE 3 – `peindre_arbre` dans le cas du stockage de l'image sous forme de liste

On observe que notre structure de liste est plus rapide à l'exécution que la structure d'arbre. En effet, l'accès aux éléments de la liste est plus immédiat que l'accès aux noeuds de l'arbre et donc la recherche d'éléments dans un arbre est moins rapide que dans la liste. Cependant, il est probable que le temps de construction de la liste rallonge l'exécution du programme . Pour vérifier cela, nous avons re calculé le temps d'exécution de notre programme s'il ne génère plus la liste mais se contente de peindre les régions sans les stocker. Cela ne nous permettra en revanche pas d'accéder aux éléments de la liste par la suite.

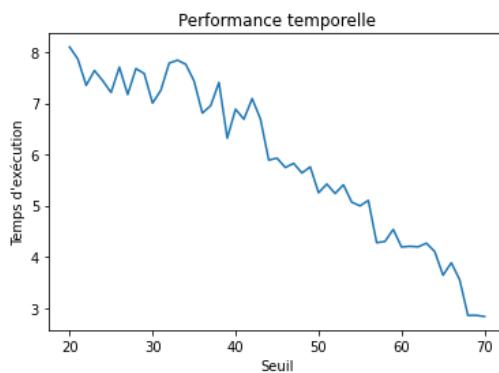


FIGURE 4 – peindre\_arbre sans ajouter les régions colorierées à une liste quelconque

On observe que si on enlève l'ajout des noeuds dans la liste à chaque itération, le programme gagne légèrement en rapidité et reste plus rapide que l'algorithme récursif. Cela confirme que la construction de la liste est bien un facteur qui ralentit le programme.

#### 1.4.2 Comparaison en mémoire

Nous ne sommes pas parvenu à quantifier et à tracer une évolution de la consommation en mémoire en fonction du seuil pour les deux algorithmes. Cependant étant donné que la complexité spatiale peut s'évaluer en  $O(\text{nb de région homogène})$  et que l'image fournie est la même à un seuil donné pour les deux programmes, ils génèrent probablement tous deux la même occupation en mémoire.

## 2 Nouveau critère d'homogénéité

### 2.1 Vision humaine

En se renseignant sur les caractéristiques de la vision humaine, nous avons vu que l'oeil est moins sensible aux contrastes de bleu que de vert. De plus, il est plus attentif aux détails du centre de l'image. Notre but a donc été de modifier la fonction est\_homogène en conséquence afin de prendre en compte ces spécificités et d'obtenir une image compressée plus adaptée à l'oeil humain. Le résultat attendu est donc une image moins compressée que dans les cas précédents mais plus 'jolie' sans pour autant prendre la même place en mémoire que l'image d'origine.

## 2.2 Implémentation

L'idée est donc de réutiliser l'algorithme de création de l'image en modifiant les tests d'homogénéité. En prenant en compte que c'est le seuil fixé qui va déterminer la finesse du découpage, nous avons décidé de pondérer par un poids plus fort dans le calcul de l'écart-type la valeur de la composante verte du pixel. De plus, l'attention étant plus portée sur le centre de l'image, nous avons décidé dans notre fonction de fixer un seuil plus bas pour une zone définissant le centre de l'image et un seuil plus haut pour les bordures.

```
def homogeneite(x,y,w,h,x0,y0,seuil):
    r=min((x-x0)**2,(x+w-x0)**2)+min((y-y0)**2,(y+h-y0)**2)
    nouveau_seuil = seuil*(r/4000000+1)
    if sum(ecart_type(x, y, w, h))/3< nouveau_seuil:
        return(True)
    else:
        False
```

Techniquement, la fonction prend en argument les caractéristiques de l'image et un seuil mais également les coordonnées du centre de l'image. Ensuite, on définit  $r$  la distance cartésienne entre le centre et le sommet de la région le plus proche. On définit un nouveau seuil de manière affine et non proportionnelle pour éviter d'avoir un seuil nul. Ainsi, plus la région est loin du centre plus celle-ci a un seuil haut. La pondération des composantes du pixel est prise en compte dans la fonction `ecart_type`.

## 2.3 Résultats et comparaisons

Pour tester l'effet de cette fonction, on a utilisé une image avec des teintes vertes et bleues afin de mieux percevoir les nuances.



FIGURE 5 – Image d'origine



FIGURE 6 – Image compressée au seuil de 10 avec la fonction homogène initiale



FIGURE 7 – Image compressée avec la fonction d’homogénéité modifiée

La différence entre les deux images ne saute pas aux yeux et pour cause, notre but était de dégrader le moins possible tout en la compressant plus en insistant sur les zones importantes. En effet, les bordures sont moins détaillées sur l’image de droite alors que son centre l’est plus. On remarque aussi que les régions vertes de la feuille qui ont été traitées avec un seuil plus bas par pondération ont plus de détails. Pour évaluer la différence de précision, on procède au calcul des PSNR :

	Fonction initiale	Fonction modifiée
PSNR	30.7	29.3

TABLE 1 – Comparaison des PSNR

On voit donc que notre nouveau critère d’homogénéité, en se focalisant sur les zones d’attention de l’œil, fournit une image moins fidèle à l’image d’origine et donc moins lourde à stocker que dans le premier cas. Pour autant, on ne perçoit pas aisément que l’image a plus été dégradée que lorsqu’elle est traitée avec la fonction initiale au seuil de 10.

### 3 Filtre de flou

#### 3.1 Implémentation

Pour implémenter le filtre flou, nous avons d’abord créé une fonction qui récupère la liste des voisins d’un pixel. Pour cela, on a réutilisé la trame de la fonction qui effectue la même tâche dans le TD4. Suite à cela, on implémente une fonction moyenne\_around qui renvoie un triplet moyen d’une liste de triplet, que l’on appliquera à la liste des voisins de chaque pixel. Enfin, la fonction flou consiste à parcourir tous les pixels de l’image et d’appliquer l’algorithme suivant :

- On génère la liste des voisins
- On récupère les couleurs correspondantes et on calcule le triplet moyen
- On affecte ce triplet au pixel en cours d’étude en prenant soin d’utiliser la partie entière

```

def moyenne_around(px,L):
    n = len(L)
    s = [0,0,0]
    for el in L:
        s[0]+=el[0]
        s[1]+=el[1]
        s[2]+=el[2]
    return(s[0]/n,s[1]/n,s[2]/n)

def flou(px,w,h):

    for i in range (w):
        for j in range (h):
            v = voisins(px, i, j)
            C = []
            for k in range (len(v)):
                C.append(get_couleur(px,v[k][0],v[k][1]))
            m = moyenne_around(px,C)
            m_entier = (ceil(m[0]),ceil(m[1]),ceil(m[2]))
            px[i,j] = m_entier

    def voisins(px, x, y):
        w,h = im.size
        L = []
        #for dx, dy in ((1,1), (1,0), (0,1), (1,-1), (0,-1), (-1,-1), (-1,0), (-1,1))
        for dx, dy in ((-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)):
            if 0 <= x+dx < w and 0 <= y+dy < h:
                L.append([x+dx,y+dy])
        return L

```

FIGURE 8 – Code pour générer l’image floutée

### 3.2 Résultats et comparaisons

Les résultats pour l’image de Lyon sont affichés ci dessous :



FIGURE 9 – Image avant application du filtre flou



FIGURE 10 – Image après application du filtre flou

On décide de focaliser l'affichage sur un détail de l'image afin de mieux percevoir l'effet de flou qu'instaure la fonction.



FIGURE 11 – Image avant application du filtre flou



FIGURE 12 – Image après application du filtre flou

Nous comparons à présent les PSNR des images générées pour évaluer comment le filtre dégrade l'image. Le calcul du PSNR pour les deux images fournit :

	Image nette	Image floue
PSNR	$\infty$	19,6

TABLE 2 – Comparaison des PSNR

On constate donc que le PSNR de l'image floue traduit une image peu dégradée par rapport aux résultats obtenus avec la fonction d'homogénéité. Cela est cohérent puisque les modifications sont moindres. Le PSNR de l'image d'origine est quant à lui infini puisque le calcul prend en compte cette image comme référence. Le filtre permet donc de compresser l'image sans pour autant dégrader significativement la perception qu'on en a. Il rend de plus les contours hachés par la compression plus lisses. Il faut donc à présent essayer d'appliquer le filtre sur une image que l'on a au préalable compressé via notre fonction homogène modifiée. Cela permettra de compresser l'image et d'atténuer l'effet 'pixelisé' grâce au filtre.

## 4 Test final

Pour finir, nous avons voulu comparer l'efficacité de l'utilisation en parallèle de notre fonction homogénéité et du lissage par rapport à la fonction d'homogénéité initiale au seuil de 10. Les résultats sont les suivants :



FIGURE 13 – Image compressée avec la méthode initiale



FIGURE 14 – Image compressée avec la combinaison des traitements

Encore une fois, il faut se concentrer sur un détail pour percevoir la différence :



FIGURE 15 – Détail de l'image de gauche



FIGURE 16 – Détail de l'image de droite

	Image de gauche	Image de droite
PSNR	30,7	28,3

TABLE 3 – Comparaison des PSNR

Finalement, la comparaison des PSNR nous indique que notre image finale est plus dégradée que l'image générée avec un seuil de 10. Cependant, le faible écart entre les valeurs montre correspond aussi à la ressemblance entre les deux images. Le fait d'appliquer le filtre sur l'image diminue le PSNR d'un point point ce qui est cohérent avec le sens de dégradation de l'image. Notre stratégie de combiner une compression via une fonction qui prend en compte les caractéristiques de la vison humaine et un filtre de lissage nous a permis de générer une image moins coûteuse en mémoire et assez proche visuellement de celle qu'un algorithme de quadripartition classique basé sur un seuil fixe aurait créé.