EL YAKABI Mehdi

GOT Emeline

A1a

ARDABILIAN Mohsen

BE 2 : L'algorithme quadripartition appliqué pour la segmentation de l'image

a) Ecrire une fonction permettant de lire la valeur d'un pixel.

La fonction *lecture* vérifie que le point demandé est bien dans l'image. Si oui, elle renvoie la valeur du pixel. Sinon, elle indique que l'on demande la valeur d'un pixel situé hors de l'image

```
1.    def lecture(x,y):
2.        if x<=w and y<=h :
3.            return(px[x,y])
4.        else:
5.            print('Hors de l image')
```

b) Ecrire une fonction permettant d'affecter une couleur à un pixel.

La fonction *couleur* vérifie qu'un point est bien dans l'image. Si oui, elle affecte la couleur demandée au pixel. Sinon, elle indique que l'on demande à colorer un pixel situé hors de l'image.

```
1.    def couleur(r,g,b,x,y):
2.        if x<=w and y<=h:
3.            px[x,y]=r,g,b
4.        else:
5.            print('Hors de l image')
```

c) Ecrire une fonction permettant d'affecter une couleur à une région rectangulaire de l'image.

La fonction *rect* permet de colorer un rectangle basé en (x,y) et de taille lx x ly. Avec une double boucle, on parcourt le triangle en longueur et en largeur et on colorie chaque pixel à l'aide de la fonction *couleur*.

```
1.    def rect(x,y,lx,ly,r,g,b): #Rectangle basé en (x,y) de largeur
      lx,ly
2.        for i in range(x,x+lx):
3.            for j in range(y,y+ly):
4.                couleur(r,g,b,i,j)
```

d) Proposez une fonction qui estime l'homogénéité des pixels d'une région rectangulaire de l'image.

On introduit d'abord une fonction *moyenne* qui calcule la couleur moyenne d'une région considérée en faisant la somme des couleurs puis en divisant par le nombre de points.

```
1.    def moyenne(x,y,lx,ly): #Calcul de la couleur moyenne sur le re
```

```
    ctangle
        r=0
        g=0
        b=0
        aire=lx*ly
        for i in range(x,x+lx):
            for j in range(y,y+ly):
                p=px[i,j]
                r+=p[0]
                g+=p[1]
                b+=p[2]
        r=r/aire
        g=g/aire
        b=b/aire
        return(int(r),int(g),int(b))
```

On utilise cette fonction *moyenne* pour déterminer ensuite les écarts-types des couleurs des points du rectangle par rapport à la couleur moyenne du rectangle.
Ensuite, on regarde si l'écart-type total est inférieur au seuil, auquel on renvoie un booléen True pour indiquer que la région est homogène. Sinon, on renvoie False.

```
    def homogeneite(x,y,lx,ly,s):
        moy=moyenne(x,y,lx,ly)
        aire=lx*ly
        ecr=0
        ecg=0
        ecb=0       #Calcul des écarts-types
        for i in range(x,x+lx):
            for j in range(y,y+ly):
                p=px[i,j]
                ecr+=(p[0]-moy[0])**2
                ecg+=(p[1]-moy[1])**2
                ecb+=(p[2]-moy[2])**2
        ecr/=aire
        ecg/=aire
        ecb/=aire
        if (sqrt(ecr)+sqrt(ecg)+sqrt(ecb)) <s:
            return(True)
        else:
            return(False)
```

e) Proposez un algorithme récursif permettant de reproduire la stratégie quadripartition.

L'algorithme de split consiste à traiter des régions homogènes pour les colorer par une même couleur (ici la valeur moyenne).

On définit une fonction auxiliaire split qui traite une région basée en (x,y) et de taille lx x ly. L'algorithme vérifie si la région en cours de traitement est réduite à un seul pixel. Si c'est le cas, alors il n'y a pas de traitement à effectuer, la couleur est déjà la couleur moyenne de la région et on l'ajoute à la liste L des régions déjà traitées. Si ce n'est pas le cas, on vérifie si la région est considérée comme homogène. Si oui, alors on applique la couleur moyenne de la région à chacun des pixels de la région considérée. On ajoute ensuite cette région à la liste L des régions traitées. Si la région n'est pas homogène, on la sépare en quatre régions (haut gauche, haut droite, bas gauche, bas droite) et on applique l'algorithme (qui est donc récursif) à chacune des régions divisées.
On applique l'algorithme récursif à l'intégralité de l'image.

```
1.    def algosplit(s):
2.        L=[]
3.        def split(x,y,lx,ly):
4.            if lx>1 and ly>1:
5.                if homogeneite(x,y,lx,ly,s):
6.                    r,g,b=moyenne(x,y,lx,ly)
7.                    rect(x,y,lx,ly,r,g,b)
8.                    L.append([x,y,lx,ly,r,g,b])
9.                else:
10.                   split(x,y,lx//2,ly//2)  #coin haut gauche
11.                   split(x+lx//2,y,lx//2,ly//2)  #coin bas gauche

12.                   split(x,y+ly//2,lx//2,ly//2)  #coin haut droite

13.                   split(x+lx//2,y+ly//2,lx//2,ly//2)  #coin bas d
      roite
14.           else:
15.               r,g,b=moyenne(x,y,lx,ly)
16.               couleur(r,g,b,x,y)
17.               L.append([x,y,lx,ly,r,g,b])
18.       split(0,0,w,h)
```

```
>>> algosplit(60)
>>> im.show()
```

```
>>> algosplit(100)
>>> im.show()
```

Plus le seuil est élevé, moins les zones sont nombreuses et moins il y a de plages de couleurs différentes.

f) Selon vous, la réalisation récursive d'un arbre quaternaire, est-elle indispensable pour l'étape de split ? Qu'en est-il pour les deux étapes split et merge ?

On peut traiter toutes les zones de manière itérative en fixant des zones de taille fixe. Mais une réalisation récursive permet de traiter les zones plus rapidement et plus efficacement. De même pour le split et merge. La complexité de l'algorithme est déjà grande, la réalisation récursive est indispensable pour que le calcul soit réalisable.

g) Ecrire l'algorithme de split en langage python.

Tout d'abord on va chercher à construire le tableau des adjacents des différentes régions. Pour cela, on écrit la fonction *adjacente1* qui pour deux régions i et j de la liste des régions L, teste si elles sont adjacentes en testant les écarts par rapport à leur centre respectif. Si les écarts sont suffisamment faibles, cela signifie que les deux régions sont voisines. La fonction renvoie alors True. Sinon, elle renvoie False.

```
1.    def adjacente1(i,j,L):
2.        if i==j:
3.            return(False)
4.        else:
5.            [x1,y1,lx1,ly1,r1,g1,b1]=L[i]
6.            [x2,y2,lx2,ly2,r2,g2,b2]=L[j]
7.            if (lx1/2+lx2/2)>=abs(x1+lx1/2-x2-
      lx2/2) and(ly1/2+ly2/2)>=abs(y1+ly1/2-y2-
      ly2/2): #On teste si les régions sont adjacentes en testant les
       écarts par rapport au centre
8.                return(True)
```

```
9.            else:
10.               return(False)
```

À partir de la fonction précédent, on va pouvoir créer la liste A des adjacents. Pour chaque région de L, on va lui affecter un numéro correspondant à sa place dans la liste et on va former sa liste d'adjacents adj. Pour cela, on vérifie pour chaque région j de L différente de i le résultat de la fonction adjacente1. SI elle renvoie True, on ajoute la région j à la liste adj des adjacents de i. Pour chaque région i de L, on va ainsi former sa liste d'adjacents qui sera de la forme [i, suivi des régions adjacentes à j]. Ces listes d'adjacents (il y en a autant que de régions de L) sont regroupées dans la liste A (c'est donc une liste de liste). A contient la liste des adjacents pour chaque sommet i de L.

```
1.    def adjacente(L):
2.        A=[]
3.        for i in range(len(L)):
4.            adj=[i]
5.            for j in range(len(L)):
6.                if adjacente1(i,j,L):
7.                    adj.append(j)  #adj est la liste qui contient
      les adjacents de i
8.            A.append(adj)  #A est la liste de listes qui contient
      les adjacents de chaque rectangle
9.        return(A)
```

On définit par la suite trois fonctions sur les listes qui vont permettre de simplifier l'algorithme final.
La fonction *fusion* consiste à fusionner deux éléments d'une liste si l'écart-type entre les couleurs est plus petit que t, le seuil de la fusion. Cela va permettre d'ajouter à une liste les régions adjacentes qui sont homogènes.

```
1.    def fusion(i,j,t): #t est le seuil de la fusion
2.        [x1,y1,lx1,ly1,r1,g1,b1]=L[i]
3.        [x2,y2,lx2,ly2,r2,g2,b2]=L[j]
4.        return ((abs(r1-r2)+abs(g1-g2)+abs(b1-b2))<t)
```

L'algorithme *merge* consiste en une amélioration de l'algorithme du split. Dans un premier temps, on effectue l'algorithme du split pour récupérer la liste des zones à la sortie de l'algorithme. À partir de cette liste, on calcule la liste A des adjacents de chacune des zones. Ensuite on traite chacune des listes d'adjacence. On cherche les adjacents compatibles (que l'on peut fusionner suivant le seuil) et on les ajoute à la liste. On ajoute les adjacents compatibles des adjacents également. À la fin on obtient des zones compatibles plus grandes qu'avec le seul algorithme de split.

```
1.    def merge(s,t):
2.        L=[]
```

```
3.
4.      def split(x,y,lx,ly):
5.          if lx>1 and ly>1:
6.              if homogeneite(x,y,lx,ly,s):
7.                  r,g,b=moyenne(x,y,lx,ly)  #Si la région est hom
    ogène, on attribue la couleur moyenne à  la région totale
8.                  rect(x,y,lx,ly,r,g,b)
9.                  L.append([x,y,lx,ly,r,g,b])
    #On ajoute la région traitée au tableau L
10.             else:  #Sinon, on sépare la région initiale en 4 ré
    gions
11.                 split(x,y,lx//2,ly//2)  #coin haut gauche
12.                 split(x+lx//2,y,lx//2,ly//2)  #coin bas gauche

13.                 split(x,y+ly//2,lx//2,ly//2)  #coin haut droite

14.                 split(x+lx//2,y+ly//2,lx//2,ly//2)  #coin bas d
    roite
15.         else:
16.             r,g,b=moyenne(x,y,lx,ly)  #Si la région est trop pe
    tite (<1 px^2), on attribue la couleur moyenne à  la région
17.             couleur(r,g,b,x,y)
18.             L.append([x,y,lx,ly,r,g,b])  #On ajoute la région
    traitée au tableau L car elle est maintenant homogène
19.
20.     split(0,0,w,h)  #On applique cette fonction à
    toute l'image
21.
22.     A=adjacente(L)  #A est le tableau des adjacents de chaque
    région homogène
23.
24. for i in range(len(A)):  #On traite toutes les listes
    d’adjacences de la liste A
25.     if A[i][0]!='T':  #On vérifie que la liste d’adjacence n’a
    pas déjà été traitée
26.         copie=A[i][1:]  #On copie la liste des adjacents de la
    zone i
27.         A[i]=[i]  #On efface les éléments de A[i] et on les
    remplace par i
28.         [x,y,lx,ly,r,g,b]=L[i]  #On stocke les infos de la zone
    i
29.         while copie!=[]:  #On vide la liste copie pour traiter
    chacune des zones
30.             r=copie[0]
31.             del(copie[0])  #r est la zone que l’on compare à i
    et on la supprime de la liste copie
32.             if A[r][0]=='T':  #Si les adjacents de la zone
    comparée ont déjà été traités
33.                 if fusion(r,i,t):  #Et si on peut fusionner les
```

```
    zones d'après
34.                         A[r][0]='T'  #Alors on marque que la zone
    est traitée
35.                         A[i].append(r)  #Et on ajoute la zone à la
    liste des adjacents « compatibles »
36.                     for j in range(1,len(A[r])):
37.                         if not (A[r][j] in copie):
38.                             copie.append(A[r][j])  #Ensuite on
    copie toutes les zones adjacentes à r dans la liste des zones
    compatibles à la zone i
39.                     [x1,y1,lx1,ly1,r1,g1,b1]=L[r]
40.                     rect(x1,y1,lx1,ly1,r,g,b)  #On applique la
    couleur de la zone i à la zone r
41.       im.show()
```

```
>>> merge(60,30)
>>> im.show()
```

Avec l'algorithme de *merge* et le même seuil pour le split, on obtient des zones plus grandes que le split seul pour le même seuil.