Gonçalves
Taïga
Groupe D1b
Encadrant: Vuillemot Romain

TD #3 : Compression d'image

1. Introduction

Dans ce TD nous allons implémenter une méthode de compression d'image par segmentation. Tout au long du TD, nous testerons cette méthode à l'aide de l'image suivante: 

Cette image est nommée *lyon.bmp* dans le dossier zip. On s'attend à ce que la compression uniformise la couleur du ciel, du bâtiment, du sol et du lion. Cette image est de taille 909 x 1092 pixels.

2. Chargement d'une image et fonctions utilitaires

Les premières fonctions du programme sont simples. Nous n'expliquerons donc que la fonction `Create` :

```
### Exercice 1.5 ###
def Create(x,y,w,h,seuil):
    """Divise le rectangle d'entrée en quatre rectangles plus petits"""
    if w < 1 or h < 1:
        # Le paramètre d'entrée n'est pas un rectangle de dimension convenable, on ne renvoie donc rien
        return None
    else:
        hg = (x, y, w//2, h//2, seuil) # rectangle en haut à gauche
        bg = (x+h//2, y, w//2, h-h//2, seuil) # rectangle en bas à gauche, de hauteur potentiellement impair (h-h//2)
        hd = (x, y+w//2, w-w//2, h//2, seuil) # rectangle en haut à droite, de largeur potentiellement impair (w-w//2)
        bd = (x+h//2, y+w//2, w-w//2, h-h//2,seuil) # rectangle en bas à droite, de dimensions potentiellement impaires
        return [hg,bg,hd,bd]
```

Dans l'exercice 1.5, nous avons pris en compte les différentes tailles que pourraient prendre la rectangle d'entrée. D'abord, pour créer 4 sous rectangles à partir d'un grand rectangle, il nous faut w et h strictement supérieur à 1. Ensuite, les rectangles du bas (bg et bd) sont codés pour avoir une hauteur potentiellement impaire, et les rectangles de droite (hd et bd) ont été définis de sorte d'avoir une largeur potentiellement impaire.

3. Création d'arbre explicite et parcours

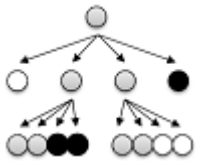
Dans cette partie, nous allons créer la classe `Node` permettant de convertir notre image en un arbre quaternaire:

```
class Node:
    """Classe permettant de créer un noeud contenant une région rectangulaire
(x,y,w,h) de couleur color, et ayant pour enfants [hg,hd,bg,bd]"""
    # Exercice 2.1
    def __init__(self,x,y,w,h,color,hg,hd,bg,bd):
        self.x = x
        self.y = y
        self.w = w
        self.h = h
        self.color = color
        self.enfants = [hg,bg,hd,bd]
        self.hg = hg
        if hg is not None:
            g.edge("-".join(map(str, (x,y,w,h))), "-".join(map(str, (hg.x, hg.y, hg.h,
hg.w))))
        self.hd = hd
        if hd is not None:
            g.edge("-".join(map(str, (x,y,w,h))), "-".join(map(str, (hd.x, hd.y, hd.h,
hd.w))))
        self.bg= bg
        if bg is not None:
            g.edge("-".join(map(str, (x,y,w,h))), "-".join(map(str, (bg.x, bg.y, bg.h,
bg.w))))
        self.bd = bd
        if bd is not None:
            g.edge("-".join(map(str, (x,y,w,h))), "-".join(map(str, (bd.x, bd.y, bd.h,
bd.w))))

    def __str__(self):
        return str(self.nb_noeuds())

    # Exercice 3.6
    def schema(self):
        """Affiche le schéma du noeud et de ses enfants"""
        g.render(filename='img')
        g.view()
```

La fonction schema (codée dans l'exercice 3.6) permet de mieux visualier l'arbre. A l'aide de cette fonction, nous avons recréé l'arbre suivant:



Le code obtenu est le suivant (exercice 2.2):

```
    racine = Node(0, 0, 4, 4, 'grey',
                  Node(0, 0, 2, 2, 'white', None, None, None, None),
                  Node(2, 0, 2, 2, 'grey',
                        Node(2, 0, 1, 1, 'grey', None, None, None, None),
                        Node(3, 0, 1, 1, 'black', None, None, None, None),
                        Node(2, 1, 1, 1, 'black', None, None, None, None),
                        Node(3, 1, 1, 1, 'grey', None, None, None, None)),
                  Node(0, 2, 2, 2, 'black', None, None, None, None),
                  Node(2, 2, 2, 2, 'grey',
                        Node(2, 2, 1, 1, 'white', None, None, None, None),
                        Node(3, 2, 1, 1, 'white', None, None, None, None),
                        Node(2, 3, 1, 1, 'grey', None, None, None, None),
                        Node(3, 3, 1, 1, 'grey', None, None, None, None)))
```

Nous créons ensuite la fonction `Terminal` à l'extérieur de la classe, et permettant de créer un arbre à partir d'une image globale délimitée par une région rectangulaire (paramètre d'entrée). La création de l'arbre s'appuie sur un critère d'homogénéité, avec donc un seuil à choisir en entrée.

```
def terminal(x,y,w,h,seuil):
    """Créee des noeuds terminaux de manière récursive de sorte à ce qu'elles
vérifient le critère d'homogénéité"""
    if homogeneite(x, y, w, h, seuil):
        color = moy(x,y,w,h)
        return Node(x,y,w,h,color,None,None,None,None)
    else:
        hg = terminal(x, y, w//2 , h//2)
        hd = terminal(x, y+w//2 , w-w//2 , h//2)
        bg = terminal(x+h//2, y, w//2 , h-h//2)
        bd = terminal(x+h//2, y+w//2, w-w//2 , h-h//2)
        return Node(x ,y, w, h, None, hg, hd, bg, bd)
```

Maintenant, implémentons d'autres fonctions dans la classe `Node` :

```
    # Exercice 2.4
    def peindre(self):
        """Peint les rectangles de chaque noeud terminal selon la couleur moyenne de
tous ses pixels dans la région"""
        if self.enfants == [None,None,None,None]: #Vérifie si le noeud est terminal
            SetColorRegion(self.x, self.y, self.w, self.h, self.color) #On réutilise
la fonction de l'exercice 1.1
        # Utilisation d'une méthode récursive pour atteindre chaque noeud terminal
        if self.hg != None:
            self.hg.peindre()
        if self.hd != None:
```

```
            self.hd.peindre()
        if self.bg != None:
            self.bg.peindre()
        if self.bd != None:
            self.bd.peindre()

    # Exercice 2.5
    def peindre_profondeur(self,profondeur=0):
        """Peint les noeuds terminaux d'un arbre d'une couleur proportionnelle à la
profondeur dans l'arbre"""
        self.__profondeur = 0 #profondeur du noeud maximal
        def profondeur_max(noeud,profondeur):
                """Calcule la profondeur maximale dans l'arbre, en prenant comme
argument la profondeur initiale"""
                if self.hg == None and self.hd == None and self.bg == None and self.bd
== None:
                    self.__profondeur = max(self.__profondeur,profondeur)
                # méthode récursive pour aller trouver la plus grande profondeur
                # à chaque nouvel appel de cette fonction, la profondeur augmente de 1
                if self.hg != None:
                    profondeur_max(self.hg,profondeur+1)
                if self.hd != None:
                    profondeur_max(self.hd,profondeur+1)
                if self.bg != None:
                    profondeur_max(self.bg,profondeur+1)
                if self.bd != None:
                    profondeur_max(self.bd,profondeur+1)
        if profondeur==0:
            # la fonction peindre_profondeur est appelée de manière récursive
            # cette condition permet donc de ne calculer la profondeur maximale qu'au
premier appel
            profondeur_max(self,0)
        if self.hg == None and self.hd == None and self.bg == None and self.bd ==
None:
            r = 255*profondeur/self.__profondeur # Crée un niveau de gris
proportionnel à la profondeur du noeud dans l'arbre
            SetColorRegion(self.x, self.y, self.w, self.h, [r,r,r])
        if self.hg != None:
            self.hg.peindre_profondeur(profondeur+1)
        if self.hd != None:
            self.hd.peindre_profondeur(profondeur+1)
        if self.bg != None:
            self.bg.peindre_profondeur(profondeur+1)
        if self.bd != None:
            self.bd.peindre_profondeur(profondeur+1)
```

La fonction `profondeur` peint chaque noeud terminal avec la couleur moyenne dans la région, alors que la fonction `peindre_profondeur` peint chaque noeud terminal avec un niveau de gris proportionnel à la profondeur du noeud dans l'arbre. Pour déterminer ce niveau de gris, `peindre_profondeur` se sert d'une fonction auxiliaire `profondeur_max` qui renvoie la valeur du noeud terminal le plus profond dans l'arbre. Ensuite, pour

chaque noeud terminal, la couleur à lui attribuer correspond à un niveau de gris égal à la profondeur du noeud divisée par la profondeur maximale.

Cette classe est maintenant fonctionnelle : nous pouvons l'utiliser en entrant `A=terminal(0,0,W,H,74)`. Ici, le seuil qui a été choisi est 74. Ce choix a été fait de sorte à noeud pas créer plus de 10 000 noeuds dans l'arbre. Pour calculer le nombre de noeuds, nous avons ajouté dans la classe `Node` la fonction suivante:

```
    def nb_noeuds(self,n=0):
        nb_noeuds = n+1 #stock le nombre de noeuds. La valeur par défaut est 0
        if self.enfants == [None,None,None,None]:
            return nb_noeuds # Il n'y a plus de noeuds à compter
        else :
            # On compte par récurrence
            noeuds_hg = self.hg.nb_noeuds(n)
            noeuds_hd = self.hd.nb_noeuds(n)
            noeuds_bg = self.bg.nb_noeuds(n)
            noeuds_bd = self.bd.nb_noeuds(n)
            return noeuds_hg+noeuds_hd+noeuds_bg+noeuds_bd
```

4. Optimisation de la compression

Nous allons maintenant créer la classe `Node2` qui reprend les fonctions de la classe `Node` mais en y ajoutant la possibilité de choisir le type des noeuds terminaux. Nous allons donc d'abord modifier l'initialisation :

```
class Node2:
    """Classe permettant de créer un noeud contenant une région rectangulaire
(x,y,w,h) de couleur color, et ayant pour enfants [hg,hd,bg,bd]"""
    # Exercice 3.1
    def __init__(self,x,y,w,h,hg,hd,bg,bd, type_partition, colors):
        self.x = x
        self.y = y
        self.w = w
        self.h = h
        self.enfants = [hg,bg,hd,bd]
        self.type = type_partition
        self.colors = colors
        self.hg = hg
        if hg is not None:
            g.edge("-".join(map(str, (x,y,w,h))), "-".join(map(str, (hg.x, hg.y, hg.h,
hg.w))))
        self.hd = hd
        if hd is not None:
            g.edge("-".join(map(str, (x,y,w,h))), "-".join(map(str, (hd.x, hd.y, hd.h,
hd.w))))
        self.bg= bg
        if bg is not None:
            g.edge("-".join(map(str, (x,y,w,h))), "-".join(map(str, (bg.x, bg.y, bg.h,
bg.w))))
        self.bd = bd
        if bd is not None:
```

```
            g.edge("-".join(map(str, (x,y,w,h))), "-".join(map(str, (bd.x, bd.y, bd.h,
bd.w))))

    # Exercice 3.6
    def schema(self):
        """Affiche le schéma du noeud et de ses enfants"""
        g.render(filename='img')
        g.view()
```

Nous allons également créer une fonction permettant d'avoir des informations sur le type de noeud utilisé :

```
    # Exercice 3.2
    def type_partition(self):
        """Renvoie le type de partition, les coordonnées du points d'origine de la
région, ses dimensions, et les
        couleurs de chaque sous régions de haut en bas, de gauche à droite..."""
        if self.colors == None:
            self.hg.type_partition()
            self.hd.type_partition()
            self.bg.type_partition()
            self.bd.type_partition()
        else:
            s = str(self.type)+" "+str(self.x)+" "+str(self.y)+" "+str(self.w)+"
"+str(self.h)
            for c in self.colors:
                s+=" "+str(c)
            print(s)
```

Nous allons maintenant créer la fonction `quadripartition_type` qui garde le principe de la fonction `terminal` de la partie précédente, mais qui est adapté pour la classe `Node2`. Cette nouvelle fonction inclus donc le choix du type du noeud :

```
# Exercice 3.3 (fonction terminal de l'exercice 2.3 adaptée pour inclure les types)
def quadripartition_type(x,y,w,h,seuil):
    h_carre_hg = homogeneite(x, y, w//2, h//2, seuil) # carré en haut à gauche
    h_carre_hd = homogeneite(x, y+w//2 , w-w//2 , h//2 , seuil) # carré en haut à
droite
    h_carre_bg = homogeneite(x+h//2, y, w//2 , h-h//2 , seuil) # carré en bas à gauche
    h_carre_bd = homogeneite(x+h//2, y+w//2, w-w//2 , h-h//2 , seuil) # carré en bas à
droite
    h_rect_gd_g = homogeneite(x, y, w//2, h, seuil) # grand rectangle à gauche
    h_rect_gd_d = homogeneite(x, y+w//2, w-w//2, h, seuil) # grand rectangle à droite
    h_rect_pt_gg = homogeneite(x, y, w//4, h, seuil) # petit rectangle tout à gauche
    h_rect_pt_g = homogeneite(x, y+w//4 , w//4, h, seuil) # petit rectangle centre
gauche
    h_rect_pt_d = homogeneite(x, y+2*w//4, w//4, h, seuil) # petit rectangle centre
droite
    h_rect_pt_dd = homogeneite(x, y+3*w//4, w-3*w//4, h, seuil) # petit recntre tout à
droite
    h_rect_gd_h = homogeneite(x, y, w, h//2, seuil) # grand rectangle en haut
    h_rect_gd_b = homogeneite(x+h//2, y, w, h-h//2, seuil) # grand rectangle en bas
    h_rect_pt_hh = homogeneite(x, y, w, h//4, seuil) # petit rectangle tout en haut
```

```
    h_rect_pt_h = homogeneite(x+h//4, y, w, h//4, seuil) # petit rectangle centre haut
    h_rect_pt_b = homogeneite(x+2*h//4, y, w, h//4, seuil) # petit rectangle centre
bas
    h_rect_pt_bb = homogeneite(x+3*h//4, y, w, h-3*h//4, seuil) # petit rectangle tout
en bas
    ho = homogeneite(x,y,w,h,seuil) # tout
    if ho or w<=1 or h<=1:
        #type 3
        r,g,b = moy(x, y, w, h)
        return Node2(x, y, w, h, None, None, None, None, 3, [r,g,b])
    elif h_carre_hg and h_carre_hd and h_carre_bg and h_carre_bd:
        # type 0
        rhg,ghg,bhg = moy(x, y, w//2, h//2)
        rhd,ghd,bhd = moy(x, y+w//2 , w-w//2 , h//2)
        rbg,gbg,bbg = moy(x+h//2, y, w//2 , h-h//2)
        rbd,gbd,bbd = moy(x+h//2, y+w//2, w-w//2 , h-h//2)
        return Node2(x,y,w,h, None, None, None, None, 0,
[rhg,ghg,bhg,rhd,ghd,bhd,rbg,gbg,bbg,rbd,gbd,bbd])
    elif h_rect_gd_g and h_rect_gd_d:
        # type 1
        rg,gg,bg = moy(x, y,w//2, h)
        rd,gd,bd = moy(x, y+w//2, w-w//2, h)
        return Node2(x, y, w, h, None,None, None, None, 1, [rg,gg,bg,rd,gd,bd])
    elif h_rect_gd_h and h_rect_gd_b:
        #type 2
        rh,gh,bh = moy(x, y, w, h//2)
        rb,gb,bb = moy(x+h//2, y, w, h-h//2)
        return Node2(x, y, w, h, None,None, None, None, 2, [rh,gh,bh,rb,gb,bb])

    elif h_rect_gd_h and h_carre_bg and h_carre_bd:
        #type 4
        rh,gh,bh = moy(x, y, w, h//2)
        rbg,gbg,bbg = moy(x+h//2, y, w//2, h-h//2)
        rbd,gbd,bbd = moy(x+h//2, y+w//2, w-w//2, h-h//2)
        return Node2(x, y, w, h, None,None, None, None, 4,
[rh,gh,bh,rbg,gbg,bbg,rbd,gbd,bbd])
    elif h_carre_hg and h_carre_hd and h_rect_gd_b:
        #type 5
        rhg,ghg,bhg = moy(x, y, w//2, h//2)
        rhd,ghd,bhd = moy(x, y+w//2 , w-w//2 , h//2)
        rb,gb,bb = moy(x+h//2, y, w, h-h//2)
        return Node2(x,y,w,h, None, None,None, None, 5,
[rhg,ghg,bhg,rhd,ghd,bhd,rb,gb,bb])
    elif h_carre_hg and h_rect_gd_d and h_carre_bg:
        #type 6
        rhg,ghg,bhg = moy(x, y, w//2, h//2)
        rd,gd,bd = moy(x, y+w//2, w-w//2, h)
        rbg,gbg,bbg = moy(x+h//2, y, w//2 , h-h//2)
        return Node2(x,y,w,h, None, None,None, None, 6,
[rhg,ghg,bhg,rd,gd,bd,rbg,gbg,bbg])
    elif h_rect_gd_g and h_carre_hd and h_carre_bd:
        #type 7
```

```
        rg,gg,bg = moy(x, y,w//2, h)
        rhd,ghd,bhd = moy(x, y+w//2 , w-w//2 , h//2)
        rbd,gbd,bbd = moy(x+h//2, y+w//2, w-w//2 , h-h//2)
        return Node2(x, y, w, h, None, None,None, None, 7,
[rg,gg,bg,rhd,ghd,bhd,rbd,gbd,bbd])
    elif h_rect_pt_gg and h_rect_pt_g and h_rect_pt_d and h_rect_pt_dd:
        #type 8
        rgg,ggg,bgg = moy(x, y, w//4, h)
        rg,gg,bg = moy(x, y+w//4 , w//4, h)
        rd,gd,bd = moy(x, y+2*w//4, w//4, h)
        rdd,gdd,bdd = moy(x, y+3*w//4, w-3*w//4, h)
        return Node2(x, y, w, h, None, None, None, None, 8,
[rgg,ggg,bgg,rg,gg,bg,rd,gd,bg,rdd,gdd,bdd])
    elif h_rect_pt_hh and h_rect_pt_h and h_rect_pt_b and h_rect_pt_bb:
        #type 9
        rhh,ghh,bhh = moy(x, y, w, h//4)
        rh,gh,bh = moy(x+h//4, y, w, h//4)
        rb,gb,bb = moy(x+2*h//4, y, w, h//4)
        rbb,gbb,bbb = moy(x+3*h//4, y, w, h-3*h//4)
        return Node2(x, y, w, h, None, None, None, None, 9,
[rhh,ghh,bhh,rh,gh,bh,rb,gb,bb,rbb,gbb,bbb])
    else:
        # Si aucun de ces critères n'est vérifié, on découpe la région en 4 et on
appelle la même méthode sur ces 4 régions
        hg = quadripartition_type(x, y, w//2 , h//2 , seuil)
        hd = quadripartition_type(x, y+w//2 , w-w//2 , h//2 , seuil)
        bg = quadripartition_type(x+h//2, y, w//2 , h-h//2 , seuil)
        bd = quadripartition_type(x+h//2, y+w//2, w-w//2 , h-h//2 , seuil)
        return Node2(x ,y, w, h, hg, hd, bg, bd, 0, None)
```

Chaque condition `if` correspond à un ensemble de condition d'homogénéité qu'une zone rectangulaire doit remplir afin qu'un certain type de noeud puisse être utilisé. Grâce à cette fonction, nous pouvons maintenant transformer l'image en arbre en entrant `A=quadripartition_type(0, 0, W, H, 66)`. De même que dans la partie précédente, le seuil 66 a été choisi de sorte à ne pas dépasser un nombre de noeud de 10 000.

Nous allons maintenant modifier la méthode par PSNR utilisée dans la partie 3 pour évaluer la qualité visuelle de notre image en prenant en compte les caractéristiques de l'oeil humain. Pour cela, nous allons utiliser le mesure de la Structural Similarity (SIM). Cette mesure est donnée par la formule suivante:

$$SSIM(x,y) = l(x,y).c(x,y).s(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_x\sigma_y + c_2)(cov_{xy} + c_3)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)(\sigma_x\sigma_y + c_3)}$$

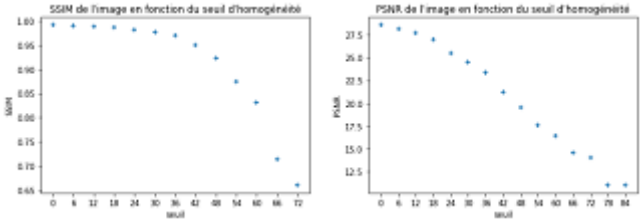
La SSIM varie de 0 à 1, et la qualité de l'image est meilleure quand la SSIM est proche de 1. En effet, la SSIM permet de mesurer la similitude de deux images. Ici, on compare l'image initiale et l'image compressée. Donc plus la SSIM est proche de 1, plus l'image compressée ressemble à l'image de base, donc la qualité de l'image est meilleure. Le code de la méthode est détaillé ci-dessous:

```
    # Exercice 3.4
    def SSIM(self):
```

```
        """Evalue la qualité de l'image en prenant en compte les caractéristiques de
l'oeil humain"""
        L = 255 # dynamique des valeurs des pixels, soit 255
        c1,c2 = (0.01*L)**2, (0.03*L)**2 # variables destinées à stabiliser la
division quand le dénominateur est très faible
        c3 = c2/2
        x,y,w,h = self.x,self.y,self.w,self.h
        rm,gm,bm = moy(x,y,w,h) # moyenne de l'image non compressée
        rv,gv,bv = ecart_type(x,y,w,h) # ecart-type de l'image non compresséex
        self.peindre()
        Rm,Gm,Bm = moy(x,y,w,h) # moyenne de l'image compressée
        Rv,Gv,Bv = ecart_type(x,y,w,h) # ecart-type de l'image compressée
        Rc,Gc,Bc = 0,0,0 #Covariance
        for i in range(self.y, self.y+self.w):
                for j in range(self.x, self.x+self.h):
                    r,g,b = px_ori[i,j] # couleurs de l'image initiale
                    R,G,B = px[i,j] # couleurs de l'image compressée
                    Rc+= (r-rm)*(R-Rm)
                    Gc+= (g-gm)*(G-Gm)
                    Bc+= (b-bm)*(B-Bm)
        n = h*w-1
        if n>0:
            Rc,Gc,Bc = Rc/n,Gc/n,Bc/n
        SSIM_R = ((2*Rm*rm+c1)*(2*Rv*rv+c2)*(Rc+c3))/((Rm**2+rm**2+c1)*
(Rv**2+rv**2+c2)*(Rv*rv+c3))
        SSIM_G = ((2*Gm*gm+c1)*(2*Gv*gv+c2)*(Gc+c3))/((Gm**2+gm**2+c1)*
(Gv**2+gv**2+c2)*(Gv*gv+c3))
        SSIM_B = ((2*Bm*bm+c1)*(2*Bv*bv+c2)*(Bc+c3))/((Bm**2+bm**2+c1)*
(Bv**2+bv**2+c2)*(Bv*bv+c3))
        return sqrt((SSIM_R**2+SSIM_G**2+SSIM_B**2)/3)

    def get_SSIM(self):
        print(self.SSIM())
```

Nous pouvons maintenant tracer le graphe des PSNR et SSIM en fonction de différents seuils (Exercice 3.5):



Nous pouvons remarquer que la qualité de l'image ne suit pas une décroissance linéaire, comme pourrait nous faire croire le graphe du PSNR (à droite). Le choix du seuil est donc important pour ne pas dégrader nettement la qualité de l'image.

Nous pouvons également visualiser l'image sous forme d'une chaîne au format DOT grâce à la fonction suivante :

```
    # Exercice 3.6
    def schema(self):
        """Affiche le schéma du noeud et de ses enfants"""
        g.render(filename='img')
        g.view()
```

5. Résumé

Dans ce TD, nous avons mis en place un programme permettant de compresser des images. Nous nous sommes d'abord intéressé à la réalisation d'une simple quadripartition (partie 3). Le résultat de la compression est le suivant:

Ensuite, nous avons mis en place des moyens pour améliorer cette compression tout en respectant la contrainte d'un nombre de noeud inférieur à 10 000. Le résultat est le suivant:

Dans les deux cas, la compression ne permet pas de garder une image suffisament nette la contrainte imposée. Toutefois, nous pouvons garder une qualité de l'image satisfaisante en allégeant cette contrainte (et donc en diminuant le seuil).