Projet : Compression basée super-pixels
Codage et compression multimédia [HAI809I]
Analyse et traitement des images [HAI804I]

Université de Montpellier - FDS
1ère année Master IMAGINE
Jean-Baptiste BES - Thomas CARO - Valentin NOYÉ

10 mars 2024

1 Introduction

Durant cette semaine nous avons implémenté l'algorithme SLIC et l'agorithme de Felzenszwalb-Hutterlocher dans le but de décomposer l'image en superpixels.

2 Algorithme SLIC

En premier lieu nous rappelons l'algorithme SLIC implémenté :

Algorithm 1 Algorithme SLIC

Require: Image I, Nombre de clusters K, Seuil s
Initialiser les centres de clusters sur une grille
Assigner chaque pixel au centre de cluster le plus proche
while Non convergé **do**
  for chaque centre de cluster C_i **do**
    Calculer la couleur moyenne et la position des pixels assignés à C_i
    Réassigner les pixels aux différents C_i selon la distance
  end for
end while
return Image segmentée

Tout d'abord après les recherches que nous avons fait sur internet il semble que la méthode SLIC fonctionne mieux lorsque l'on change d'espace couleur vers l'espace Lab. Nous avons donc implémenté une fonction permettant de passer de l'espace RGB à l'espace XYZ à l'espace Lab. Ensuite nous avons implémenté une structure point contenant les caractéristiques spatiales et chromatiques de chaque pixel :

```
struct Point{
    int x;
    int y;
    int L;
    int a;
    int b;
};
```

Figure 1 – Structure Point

Pour les mesures de similarité entre centroide de superpixels et les autres pixels nous utiliserons la distance euclidienne.

```
float distanceEuclidienne(Point p1, Point p2) {//Potentiellement mettre des poids
    float distanceXY=sqrt(pow(p2.x-p1.x,2)+pow(p2.y-p1.y,2));
    float distanceLab=sqrt(pow(p2.L-p1.L,2)+pow(p2.a-p1.a,2)+pow(p2.b-p1.b,2));
    return distanceXY+distanceLab;
```

Figure 2 – Calcul de similarité

À noter que nous pouvons modifier le poids des caractéristiques spatiales et chromatiques pour mettre plus d'importance sur l'une ou l'autre.

Nous pouvons choisir le nombre de centroides que l'on souhaite en entrée de programme. Cependant il faut placer ces centroides dans l'image. Au départ nous avions implémenté une génération aléatoire de centroides dans l'image, mais cela faisait grandement varié les résultats et généralement rendait la convergence plus difficile. Nous avons donc opté pour une génération de centroides dans des cases de grille régulière générées au sein de l'image, ce qui est en général l'approche utilisée pour l'initialisation de centroide dans ce genre de méthode de segmentation d'image.

Au sujet du seuil de convergence utilisé nous avons choisi d'utiliser le nombre de changement de classe de pixel à chaque itération. Nous prenons ce paramètre en entrée et en général les valeurs permettant d'avoir des résultats intéressants se situent entre un dixième de la taille en pixel de l'image et la moitié de l'image.

Dernier détail, pour afficher les bordures de superpixel nous utilisons une fonction colorant en noir les pixels de transitions horizontale et verticale de classe.

Voici un exemple avec comme image de départ :

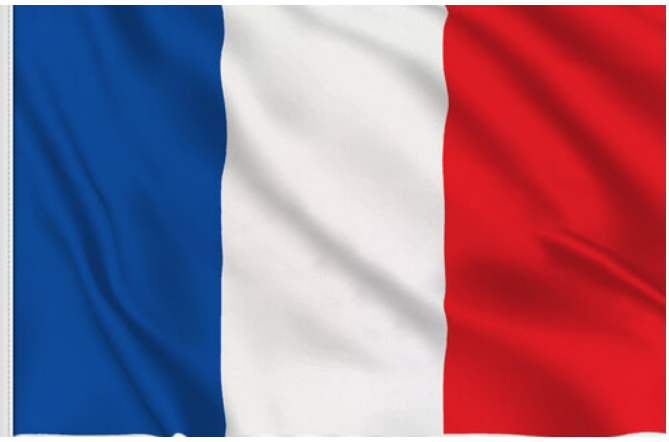


Figure 3 – Drapeau France

En appliquant l'algorithme avec 3 clusters et un seuil de 5000 changement de classe nous obtenons au bout d'environ 10 itération les superpixels suivants :

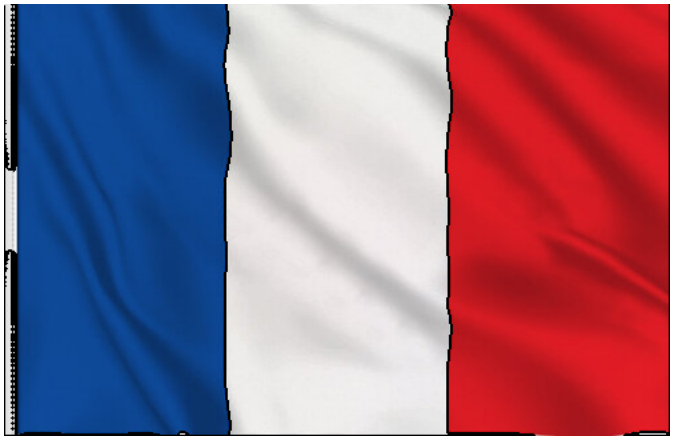


Figure 4 – Drapeau France segmenté

Comme nous pouvons le voir les 3 zones du drapeau sont clairement bien identifiées mais la petite zone blanche sur la gauche pose problème à l'algorithme. Cela vient des distances de similarité étant donné que la couleur de cette bordure à gauche se rapproche bien évidemment du blanc du drapeau mais que tout ces pixels sont spatialement plus proche du superpixel bleu.

Un autre exemple avec avec le drapeau de la Russie prenant pleinement l'image :

Figure 5 – Drapeau Russie

L'image étant de base bien segmentée nous obtenons le résultat suivant avec 3 cluster et un seuil à 5000 :

Figure 6 – Drapeau Russie segmenté

Comme prévu, aucun problème au niveau de la segmentation.
Et maintenant un exemple un peu plus complexe avec l'image suivante :

Figure 7 – Image Cote

En appliquant l'algorithme avec 30 clusters et 5000 de seuil nous avons le résultat suivant :

Figure 8 – Image Cote segmentée

L'exécution a pris une dizaine de seconde et le résultat semble convenable pour une première implémentation, réussissant à isoler parfaitement la plage par exemple.

3 Algorithme de Felzenszwalb-Hutterlocher

3.1 Description de l'algorithme

L'algorithme de Felzenszwalb-Hutterlocher est un algorithme de segmentation de l'image, convertie depuis l'espace RGB vers l'espace Lab, permettant de traiter les pixels avec plus de précision en ce qui concerne la capacité de l'oeil à percevoir la luminance et les informations colorimétriques de l'image. Cet algorithme a été implementé selon l'algorithme proposé par Felzenszwalb et Hutterlocher dans leur article de recherche à ce sujet, nous permettant de détailler son implémentation dans ce compte-rendu. Cet algorithme se base sur un graphe d'adjacence des pixels de l'image, où chaque sommet représente un pixel et sa couleur, et une arête du graphe relie un sommet v_i à un autre sommet v_j de son 4-voisinage. Cette arête, dénotée $o_{i,j} = (v_i, v_j)$ est pondérée par la dissimilarité de ces deux pixels, soit leur distance colorimétrique dans l'espace Lab, notée $w(o_{i,j})$. La composante C_i de notre graphe est un sous-graphe dont la différence interne est définie par le poids de l'arête maximale de l'arbre couvrant de poids minimal (MST) de ce sous-graphe.

$$Int(C) = \max_{e \in MST(C)} w(e)$$

Nous notons qu'une composante comprise d'un unique pixel ne possède pas de MST, donc $Int(C) = 0$.
Par ailleurs, nous notons la différence entre deux composantes C_1 et C_2 comme le poids minimum entre deux pixels adjacents de ces composantes. Cette différence permettra de construire le prédicat permettant de déterminer la présence d'une limite entre deux composantes.

$$Dif(C_1, C_2) = \min_{v_i \in C_1, v_j \in C_2} w((v_i, v_j))$$

Or cette mesure est considérée problématique car elle ne reflète que le plus petit poids entre deux composantes. Même si le choix de la médiane ou d'un autre quantile s'avère produire de meilleurs résultats, elle peut être utilisée mais ne conduira pas à des résultats optimaux.
Enfin, l'ensemble des pixels de ce sous-graphe partagent donc une similarité définie à chaque itération par le prédicat suivant, déterminant si les pixels appartenant à une composante C_1 doivent être fusionnés à ceux de la composante C_2.

$$D(C_1, C_2) = \begin{cases} vrai & \text{si } Dif(C_1, C_2) > \min(Int(C_1) + \tau(C_1), Int(C_2) + \tau(C_2)) \\ faux & \text{sinon} \end{cases}$$

L'introduction d'un seuil $\tau(C)$ n'est pas fortuite, car elle permet d'ajuster la forte considération des petites composantes, telles que lorsque la différence interne est égale à 0. Elle est définie par $\tau(C) = \frac{k}{|C|}$ où $|C|$ est la cardinalité de la composante, soit le nombre de pixels appartenant à ce sous-graphe, et k est un paramètre que nous utiliserons dans notre programme afin de contrôler sa segmentation.

L'algorithme, proposé par Felzenszwalb et Hutterlocher est présenté comme suit, où S^n représente la $n^{\text{ème}}$ segmentation de l'image :

Algorithm 2 Algorithme de Felzenszwalb-Hutterlocher

Require: Image I, Paramètre k de contrôle de fusion
Trier l'ensemble des arêtes $(o_1, ..., o_m)$ dans l'ordre croissant
Initialiser la segmentation S^0 où chaque sommet est sa propre composante
for q allant de 1 à m **do**
  On sélectionne v_i de C_i^{q-1} et v_j de C_j^{q-1} tels que $o_q = (v_i, v_j)$
  if $C_i^{q-1} \neq C_j^{q-1}$ **and** $D(C_i^{q-1}, C_j^{q-1})$ est satisfait **then**
    Fusionner C_i^{q-1} et C_j^{q-1} pour former S^q
  else
    $S^q = S^{q-1}$
  end if
end for
return S^m

Le fait de trier les arêtes dans l'ordre croissant permet d'optimiser la section itérative de l'algorithme de différentes manières :

— Il n'est pas nécessaire de garder en mémoire le poids minimal entre chaque composante car pour $v_1 \in C_1$ et $v_2 \in C_2$ avec $C_1 \neq C_2$, $Dif(C_1, C_2) = w((v_1, v_2))$ car (v_1, v_2) sera toujours la plus petite arête reliant ces composantes et décisive sur leur fusion.

— À chaque fusion entre C_1 et C_2 en C avec l'arête (v_1, v_2), $Int(C) = w((v_1, v_2))$ car le MST construit ici passe nécessairement par cette nouvelle arête (v_1, v_2), où $w((v_1, v_2))$ est maximal, connectant les deux sous-graphes de C_1 et C_2.

3.2 Détails d'implémentation

Le programme écrit en C++ suit en grande partie l'algorithme proposé. Pour une image I de taille $w \times h$, l'initilisation démarre par le changement d'espace couleur de l'image et la création de la matrice $w \times h$ des composantes de chaque pixel ; ainsi que des matrices des arêtes horizontales de taille $(w-1) \times h$ et verticales de taille $w \times (h-1)$, triées dans l'ordre croissant de la différence entre le pixel et son voisin de droite, ou de son voisin du bas respectivement pour chacune de ces deux matrices.

Chaque composante C est une instance de la classe `Component` munie de l'ensemble de ses pixels, la couleur moyenne de ces derniers ainsi que du poids maximal de son MST, $Int(C)$.

À l'itération, l'arête (v_1, v_2) au poids le plus faible appartenant à l'une des deux matrices des arêtes est sélectionnée, pour laquelle nous déterminons si les composantes C_1 et C_2, si différentes, sont en mesure de fusionner en fonction du prédicat $D(C_1, C_2)$. La fusion se fait d'abord par l'attribution de $w(v_1, v_2)$ à $Int(C_1)$ puis par le transfert des pixels de C_2 à C_1, et de leur couleur, où la nouvelle moyenne des couleurs est calculée. C_2 est ensuite supprimée.

Le programme se termine lorsque toutes les arêtes ont été parcourues, où lorsque nous avons atteint un nombre minimum de segmentations sur l'image, car malheureusement, cette approche ne permet pas efficacement de choisir au préalable un nombre précis de segmentations. On attribue à chaque pixel de l'image de sortie la couleur moyenne de leur composante, qui est convertie depuis Lab vers un format RGB.

Les libraries utilisées pour l'implémentation de cet algorithme sont stb ainsi que colorm disponibles sur GitHub.

3.3 Résultats

L'exécution de cet algorithme paraît générer des résultats corrects même s'il est difficile d'en déduire à la perfection son bon fonctionnement. Il délimite certaines régions de l'image tout en gardant ensemble les régions les plus homogènes. Or, cet algorithme génère ici 5184 segmentations, et un nombre si élevé est généralement attribué à un k très faible.

(a) Image côte

(b) Côte segmentée

FIGURE 9 – Segmentation de l'image de côte

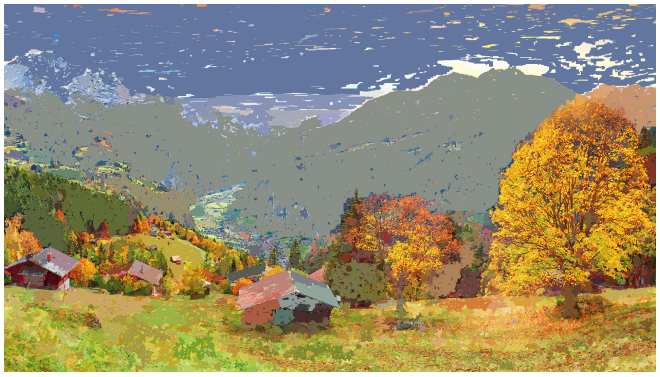
Notre second image de test est une photo de stock prise par Vadym Lavra. L'image étant large de base, il est naturel de constater la présence d'un grande nombre de segmentations de l'image, mais cela permet également de discerner les régions de pixels où les dissimilarités, donc les variations, sont faibles.

Par l'attribution des couleurs moyennes aux multiples composantes de l'image, nous pouvons déterminer en faisant varier k, la valeur de ce paramètre qui nous intéresse le plus, prioritisant à la fois la capacité de l'algorithme à réduire la quantité d'information de l'image sans pour autant la dégrader entièrement. On se base donc sur le PSNR afin de déterminer cette mesure.

(a) Image originale

(b) $k = 4$, PSNR $= 32,70$dB

(c) $k = 10$, PSNR $= 20,62$dB



(d) $k = 25$, PSNR $= 12,83$dB

FIGURE 10 – Segmentations dépendantes de k

Cet algorithme fait perdre un grand nombre d'information à l'image, mais nous apercevons que ces fusions sont cohérentes. Tandis qu'un k aussi faible que 4 reconstruit l'image à un PSNR de 32,70 dB, la grande majorité de l'image reste intacte. Il faut atteindre un k d'environ 2,48 afin de reconstruire l'image à un PSNR supérieur à 40 dB. Pour un k plus faible, le temps d'exécution du programme est plus rapide car moins de fusions prennent place. Par exemple, on estime ici un temps d'exécution de 4 à 5 secondes pour $k = 4$ mais d'environ 15 à 20 secondes pour $k = 25$.