Devoir 4 – Algorithme de segmentation avancé : GrabCut

Note : 10% des points de ce laboratoire seront attribués en fonction de la qualité du code. Du code qui n'exploite pas correctement la syntaxe compacte de MATLAB, similaire à du code C, sera pénalisé. Pour plus d'information, consulter les trois documents d'introduction à MATLAB fournis avec le premier laboratoire.

Introduction au GrabCut

Le but de ce laboratoire est de comprendre et implémenter un algorithme avancé de segmentation d'images. C'est l'algorithme GrabCut basé sur les travaux séminaux suivants. Chacun de ces travaux est cité plus que 4000 fois !

[1] Yuri Boykov and Marie-Pierre Jolly (2001): Interactive Graph Cuts for Optimal Boundary and Region Segmentation of Objects in N-D Images. *IEEE International Conference on Computer Vision*, p. 105-112

[2] Carsten Rother, Vladimir Kolmogorov, Andrew Blake (2004): "GrabCut": interactive foreground extraction using iterated graph cuts. *ACM Transactions on Graphics*, vol. 23, no. 3, p. 309-314

[3] Yuri Boykov, Vladimir Kolmogorov (2004), An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 9, p. 1124-1137.

L'algorithme GrabCut est utilisé aujourd'hui dans plusieurs produits commerciaux incluant Powerpoint! Il s'agit également d'un outil très puissant pour l'annotation d'images qui devient de plus en plus populaire dans l'industrie et dans le monde académique depuis l'arrivée de l'apprentissage profond (*Deep learning*).

Nous avons vu en classe que l'algorithme GrabCut est basé sur l'optimisation de la fonction suivante:

$$min_S \lambda |\partial S| + \sum_{p \in S} f(p)$$

La fonction *f* représente un ratio de probabilité entre deux termes : la probabilité qu'un pixel p ayant l'intensité ou la couleur I_p appartienne à l'avant plan S, divisé par la probabilité qu'il appartienne à l'arrière-plan (tout ce qui n'est pas dans S).

$$f(p) = -\log \frac{\Pr(I_p|S)}{\Pr(I_p|\Omega - S)}$$

Où p représente un pixel de l'image
S représente les pixels de la région à segmenter (voir fig. 1)
Ω représente tous les pixels de l'image
I_p représente l'intensité (ou la couleur si c'est un vecteur) d'un pixel p
La fonction logarithmique est utilisée pour permettre l'addition des probabilités.

Le deuxième terme mesure la longueur de frontière de la segmentation *S* recherchée. Son objectif est de favoriser les frontières lisses et d'éliminer les petites régions isolées dans la solution. Nous avons vu en classe que ce terme est donné par :

$$\lambda|\partial S| = \lambda \sum_{p,q\in\mathcal{N}} \delta(s_p, s_q)$$

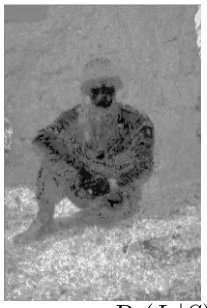
Qui représente la longueur de la frontière, multipliée par une constante lambda.

$$\delta(s_p, s_q) = \begin{cases} 1 & \text{si } s_p \neq s_q; \\ 0 & \text{sinon.} \end{cases}$$

La fonction du delta de Dirac ici permet de représenter les frontières : elle s'active uniquement le long des frontières.

Aussi, notez que cette fonction de segmentation est équivalente à:

$$min_S \lambda|\partial S| + \sum_{p\in S}(-logPr(I_p|S)) + \sum_{p\in\Omega-S}(-logPr(I_p|\Omega - S))$$



$$f(p) = -\log \frac{\Pr(I_p|S)}{\Pr(I_p|\Omega - S)}$$

En ajoutant le terme de frontière

figure 1: Illustration de l'algorithme GrabCut.

Librairies, codes et images partagées sur Moodle

Disponible sur Moodle, vous trouverez une implémentation incomplète se limitant à la première itération de l'algorithme GrabCut, aussi bien qu'une base d'images. Il s'agit de la base de données GrabCut de Microsoft que les chercheurs et ingénieurs utilisent pour valider leurs algorithmes de segmentation.

Nous allons utiliser la fonction principale *GrabCut.m*. Cette fonction de segmentation utilise une librairie C++ précompilée (fichier matlab compilé *mex*) implémentant

l'algorithme séminal de Boykov et Kolmogorov (*BK algorithm*). Cet algorithme d'optimisation est basé sur le principe de coupure de graphes (voir figure 2 ci-dessous) qui consiste à construire un graphe dont la division en deux sous-graphes (avant et arrière-plan) correspond au minimum de notre fonction d'optimisation, c'est-à-dire la meilleure segmentation possible selon notre critère. Il s'agit donc de définir les nœuds et arêtes du graph, aussi bien que les poids de ces arêtes et le tour est joué. Un nœud est un pixel et une arête représente un lien de voisinage. La librairie « graph cut » trouve la coupure optimale du graphe pour vous, et de façon très efficace !

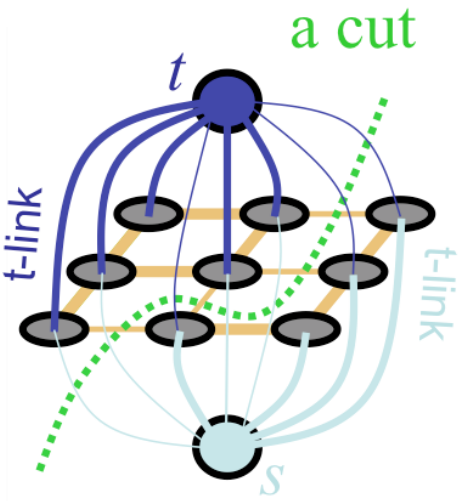


Figure 2: Illustration d'une coupure de graphe à partir des acétates du Prof. Boykov

Pour optimiser notre fonction, on peut construire un graphe qui consiste en

1. Un nœud par pixel dans l'image (les nœuds gris dans la figure)
2. Un nœud virtuel représentant le l'arrière-plan (nœud t)
3. Un autre représentant le l'avant-plan (nœud s)

Dans le code partagé avec vous, à l'intérieur de la fonction calculerProbabilitesParPixel, la connexion de chaque pixel à *t* est définie par le poids:

$$-\log \Pr(I_p|\Omega - S)$$

La connexion de chaque pixel à *s* est définie par le poids:

$$-\log \Pr(I_p|S)$$

La coupure de graphe trouve de façon très efficace une chaine d'arêtes dont la somme des poids est minimale, et qui coupe le graphe en deux sous-graphes déconnectés. Cette somme de poids porte également le nom « d'énergie ». On peut démontrer facilement que minimiser les termes logarithmiques pixel-avant-plan et pixel-arrière-plan ci-haut correspond exactement à la minimisation de la fonction *f* définit plus tôt. Ces poids sont souvent appelés *unary potential* dans la littérature.

Aussi, les pixels de l'image sont connectés entre eux (voir figure 2). Les poids de ces connexions entre ces pixels voisins (appelés *pairwise potentials* dans la littérature)

représentent le terme de régularisation de frontière. L'implémentation qui est partagée avec vous contient tout ce qu'il faut pour cela. Vous n'avez qu'à donner une valeur au paramètre lambda ! La fonction getNeighborhoodWeights_radius se chargera de générer les poids pour les arêtes du graphe.

À faire:

1) Lisez le code attentivement. Le code contient des commentaires qui vous aideront à comprendre l'implémentation.

2) Essayez le code et observez les figures. Prenez le temps de bien comprendre chacune des trois figures.

3) Répétez l'expérience avec différentes valeurs de lambda. Prenez le temps de bien comprendre l'effet du lambda.

4) Dans l'implémentation fournie, la fonction de probabilité est approximée à partir d'un histogramme de l'image. L'histogramme est généré par la fonction *calculerProbabiliteParPixel()* qui assigne deux probabilités à chaque pixel de l'image. L'approche est présentement très limitée, puisqu'elle ne tient pas compte des couleurs : elle convertit l'image en niveau de gris et calcule un histogramme 1d. Essayez l'algorithme avec une autre image qui contient de la couleur pour vous en convaincre. Modifiez cette fonction afin qu'elle ait les propriétés suivantes :

   a. Aucune conversion d'image en niveau de gris.
   b. Les histogrammes 3d doivent avoir une taille Nbins x Nbins x Nbins.
   c. Les poids logarithmiques sont correctement calculés à partir des histogrammes en 3d.

5) Maintenant, on va compléter le code de coupe de graph pour obtenir l'algorithme GrabCut qui est un algorithme itératif. Il consiste à répéter ce qu'on a fait dans *GrabCut.m* pour un certain nombre d'itérations jusqu'à convergence. Dans chaque itération, nous avons deux étapes :

   a. Mise à jour des histogrammes des régions à partir du dernier masque de segmentation généré à l'itération précédente.
   b. Appliquer la coupure des graphes pour obtenir une nouvelle segmentation.

   Vous pouvez arrêter après un certain nombre prédéfini d'itérations ou en testant si la segmentation a changé d'une itération à une autre (Test de convergence). Maintenant, Réessayez ce que vous avez fait dans (3). Est ce que les résultats se sont améliorés?

6) Nous avons vu en cours qu'on peut ajouter des contraintes du type "*hard contraints*". Ces contraintes forcent certains pixels à appartenir à l'avant-plan ou à l'arrière-plan. Elles sont très faciles à imposer: Il s'agit d'ajouter des constantes larges à nos termes de connexion vers *s* et *t* (voir la figure 3 pour illustration). Dans l'implémentation *GrabCut.m*, faites les modifications suivantes :

   a. Déclarez deux tableaux de contraintes de taille MxN, un pour l'avant-plan, un pour l'arrière-plan, initialisés à 0.

b. Après première une série d'itérations (jusqu'à convergence), demandez à l'utilisateur s'il désir imposer une contrainte de type avant-plan (utiliser la fonction *input*).
c. Seulement si l'utilisateur répond extactement 'oui', utilisez la fonction *getrect*() pour permettre la sélection d'une région. Mettez à jour le tableau de contrainte.
d. Répétez b et c pour l'arrière-plan.
e. Pour mettre à jour le tableau de contraintes, utilisez la logique suivante :
    1. Pour forcer des pixels à l'avant-plan, on ajoute un très grand poids au mapping de probabilité de l'arrière-plan (bkgProbabilitees).
    2. Pour forcer des pixels à l'arrière-plan, on ajoute un très grand poids au mapping de probabilité de l'avant-plan (objProbabilitees).

    Note : si le poids pour une contrainte manuelle est trop petit, l'algorithme le traitera comme une "suggestion" plutôt qu'une contrainte. Ce poids devrait être plus grand que la somme de tous les poids obtenus de la fonction *calculerProbabiliteParPixel()*.

f. Lancez une nouvelle série d'optimisations par coupe de graphe jusqu'à convergence, mais cette fois-ci, modifiez la structure probabilitesParPixel dans pour qu'elle tienne compte des contraintes.

    Note : Les tableaux de poids pour les contraintes manuelles doivent être additionnés aux tableaux des poids obtenus de la fonction *calculerProbabiliteParPixel()* juste avant d'apeller la fonction d'optimisation par coupe de graphe. N'oubliez pas que ces probabilités changent à chaque itération!

g. Testez votre code sur quelques images.

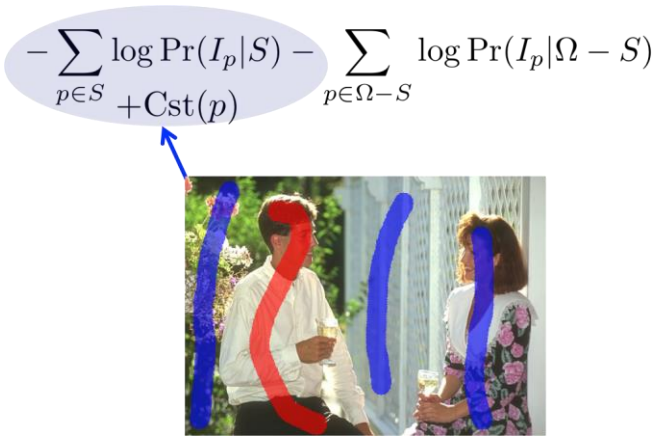


Figure 3: Contraintes du type "hard constraints": Forcer des pixels à appartenir au background. Ces contraintes imposées par l'utilisateur peuvent aussi être utilisées pour l'apprentissage des histogrammes des régions.

7) Enfin, nous allons ajouter une petite mesure permettant d'évaluer la qualité de la segmentation. Calculer et afficher dans la console l'indice de Sørensen-Dice (souvent appelé le *Dice index*). Vous pouvez utiliser la fonction *fprintf* ou *display* pour l'afficher. Faites une petite recherche sur internet pour en apprendre plus sur le calcul du *Dice index*. Utilisez la variable « groundTruth » comme référence dans votre calcul.

À remettre :

Un seul fichier zip contenant :

- La fonction GrabCut complétée
- La fonction calculerProbabilitesParPixel modifiée et complétée
- Toute autre fonction que vous avez créée nécessaire à l'exécution.
- Aucun autre fonction, fichier ou dossier.

Attention! Assurez-vous que votre fonction GrabCut s'exécute directement, sans erreur, telle que décrite ci-haut. Vous serez évaluée sur son bon fonctionnement.