Projet Fondements de l'informatique

Luc Brun & Eric Ziad-Forest

1 Informations générales

Début du projet : Semaine du 15 Novembre

Remise du projet : Dimanche 7 Janvier

Nombre de personnes : le projet est obligatoirement effectué en binôme dans le même groupe de TP.

Documents à remettre : Un rapport écrit de 10 pages (max) avec une description des méthodes algorithmiques et des structures de données utilisées ainsi qu'un jeu d'essais. Ce document doit être remis à votre encadrant de TP de C. Une présentation au format pdf de 4 à 8 diapositives à remettre sur la plateforme moodle.

Note d'information : une fonction (ou une variable) privée à un module est une fonction (resp. variable) déclarée en static dans le fichier .c de ce module. Elle est à ce moment là locale au module et n'interfère pas avec le reste du code qui ignore jusqu'à son existence.

2 Description du projet

La segmentation est la première étape d'abstraction du contenu d'une image. Cette étape permet en effet de passer d'une image brute (un tableau de pixels) à un ensemble de régions homogènes correspondant à des objets présents dans l'image ou a des parties de ceux-ci. Une grande famille de méthodes est basée sur la fusion itérative de régions. C'est cette approche que nous allons découvrir dans ce Projet via une approche simplifié.

On va pour cela commencer à créer une partition d'une image en un ensemble de $n \times m$ blocks rectangulaires. Ces blocks sont numérotés de haut en bas et de gauche à droite (Tab. 1).

1. Soit p le numéro d'un block. Donnez, en fonction de n et m les numéros du block de dessus, dessous, droite et gauche quand ils existent (on identifiera également les conditions permettant de tester l'existence de ces voisins).
2. On suppose que notre partition en blocks est superposée à une image de taille $L \times H$. De ce fait chaque block correspond à un rectangle de $\frac{L}{n} \times \frac{H}{m}$ pixels. Indiquez-les coordonnées dans l'image du pixel en haut à gauche d'un block d'indice i. (La réponse pour le block 0 est $(0, 0)$, celle pour le block 1 est égale à $(\frac{L}{n}, 0)$).

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29

TABLE 1 – Un exemple de numérotation de blocks avec une partition 6×5.

3 Calcul des moments dans un block

Cette partie (et les suivantes) utilisera le type `image` définit dans les fichiers image.c et image.h disponbile sous `/home/public/PROJET_FONDEMENTS/MODULE_IMAGE`

1. Définissez dans un fichier à part, la fonction :

   ```
   extern void give_moments(image,int,int,int,int*,double*,double*);
   ```

   Cette fonction calcule les moments d'ordre 0,1 et 2 des pixels situées dans un rectangle. Le second paramètre correspond au numéro du block tandis que les paramètres 3 et 4 correspondent aux valeurs de n et m indiquant le nombre de blocks par ligne et par colonne.

   Le moment d'ordre 0 (cinquième paramètre) correspond au nombre de pixels dans le block. Le moment d'ordre 1 est la somme des intensités (ou couleurs) des pixels dans le block(6^e paramètre). Le moment d'ordre 2 est la somme des carrés des intensités ou couleurs dans le block(7^e paramètre). On a donc en couleur et dans l'espace RGB :

   $$\begin{array}{rcl} M0 & = & \sum_{P\in\mathcal{B}} 1 \\ M1 & = & \sum_{P\in\mathcal{B}} (R(P), G(P), B(P)) \\ M2 & = & \sum_{P\in\mathcal{B}} (R^2(P), G^2(P), B^2(P)) \end{array}$$

   où $\mathcal{B}$ correspond au block spécifié par le paramètre deux, $R(P), G(P), B(P)$ correspond aux composantes rouges, vertes et bleus du pixel P. Les équations sont identiques en niveaux de gris. On a simplement dans ce cas une seule composante par pixel. Notez que $M0$ est toujours un scalaire alors que $M1$ et $M2$ sont des vecteurs en couleur et des scalaires en niveau de gris.

4 Création du graphe d'adjacence de régions

On désire coder les adjacences de blocks. Pour cela on va créer un tableau de liste. Chaque entrée du tableau correspond à un block et contient la liste des blocks adjacents (c'est a dire les blocks partageant au moins un coté avec le

block en question). On pourrait par exemple avoir (cf Tab. 1) :

0	→	1	→	6					
⋮									
21	→	20	→	15	→	22	→	27	
22	→	21	→	16	→	23	→	28	
⋮									

En fait ce codage est un peu redondant, dans la mesure où l'adjacence de deux blocks est codée 2 fois (21 est dans la liste de 22 et 22 se trouve également dans la liste de 21). Afin de réduire la complexité de la structure chaque entrée du tableau ne stocke que les indices des blocks adjacents de valeur supérieure à celle du block courant. Le stockage devient donc celui-ci :

0	→	1	→	6
⋮				
21	→	22	→	27
22	→	23	→	28
⋮				

En plus de stocker la liste des voisins, nous voulons stocker pour chaque block, ses moments (d'ordre 0, 1 et 2) ainsi qu'un pointeur sur un parent dont l'explication sera fournie ultérieurement. Un graphe d'adjacence peut donc se définir de la manière suivante :

```
typedef struct RAG * rag; /* a mettre dans RAG.h*/

/* a mettre dans RAG.c */
struct moments
{
    int M0;
    double M1[3];
    double M2[3];
};

typedef struct cellule* cellule;

struct cellule
{
    int block;
    cellule next;
};

typedef struct moments* moments;
```

```
struct RAG
{
     image  img;
     int nb_blocks;
     long double erreur_partition;
     moments m;
     int * father;
     cellule *neighbors;
};
```

où moments, father et neighbors sont des tableaux de taille `nb_blocks` de struct moments, entiers et cellule.

1. Définissez la fonction :

   ```
   extern rag create_RAG(image, int,int); /* dans RAG.h*/
   ```

   prenant en paramètre une image et les deux entiers n et m définissant le nombre de blocks par ligne et par colonne. On utilisera la fonction `give_moments` pour calculer les moments des différents blocks. Le père de chaque block sera initialisé à lui même. Les listes de voisins seront définies comme expliqué précédement. Ces opérations seront réalisées par les trois fonctions privées :
   — `static void init_moments_priv(rag,int,int);`
   — `static void init_father_priv(rag);`
   — `static void init_neighbors_priv(rag,int,int);`
   où la structure `rag` passée en paramètre aura déjà ses champs `image` et `nb_blocks` initialisés. Les `int`s passés en second et troisième arguments sont le nombre de blocks par ligne et par colonne.

L'erreur quadratique d'un block est définie comme la somme des écarts à la moyenne au carré des pixels du block :

$$EQ(B) = \sum_{P\in B} \|I(P)-\mu_B\|^2 = \sum_{P\in B} (R(P)-\mu_B[0])^2+(G(P)-\mu_B[1])^2+(B(P)-\mu_B[2])^2$$

où μ_B est la couleur moyenne du block définie par $\frac{M1}{M0}$. On peut montrer que l'erreur quadratique est également donnée par la formule ci-dessous :

$$\begin{aligned} EQ(B) &= \sum_{i=0}^{3} M2[i] - \left(\frac{M1[i]^2}{M0}\right) \\ &= M0 \sum_{i=0}^{3} \frac{M2[i]}{M0} - \left(\frac{M1[i]}{M0}\right)^2 \end{aligned}$$

On « voit » que l'erreur quadratique est égale à la somme des variances en R, G et B du block multiplié par le cardinal de celui-ci (son moment d'ordre 0). L'erreur quadratique mesure donc l'homognité d'un block.

On définit l'erreur d'une partition (ici en blocks) par la somme des erreurs quadratiques des blocks. Définissez la fonction privée :

```
static init_partition_error_priv(rag)
```

qui initialise l'erreur de partition. Cette fonction doit être appelée après `init_moments_priv`.

5 Calcul de la meilleure fusion

Nous allons mesurer le coût de fusion de deux blocks par l'augmentation de l'erreur quadratique induite par leur fusion. Cette augmentation peut être facilement mesurée par la formule suivante :

$$EQ(B \cup B') - EQ(B) - EQ(B') = \frac{M0(B)M0(B')}{M0(B) + M0(B')} \|\mu_B - \mu_{B'}\|^2 \quad (1)$$

1. Définissez la fonction :

   ```
   extern double RAG_give_closest_region(RAG,int* int*);
   ```

   qui renvoie les deux indices de blocks (dans les deux `int*` en paramètres) dont la fusion induit la plus petite augmentation d'erreur quadratique (equation 1). La valeur de cette augmentation sera renvoyée par la fonction. Seuls les blocks vérifiant (`father[i]==i`) seront pris en compte dans le calcul. L'explication de cette restriction est fournie dans la section suivante.

Si l'on suppose que chaque block a un nombre fixe de voisins qu'elle est la complexité de cette opération en fonction du nombre de blocks ?

6 Fusion de régions

La fusion de deux blocks nécessite de mettre à jour :

1. Les moments,
2. Le tableau `father`
3. Les listes de voisins,
4. L'erreur de partition.

Supposons que l'on veuille fusionner deux blocks d'indices i et j avec $i < j$. Le block i est invalidé en positionnant `father[i]` à j. Les moments du block j sont mis à jour simplement par sommation :

$$\begin{array}{lcll} moment[j].M0 & += & moment[i].M0 & \\ moment[j].M1[k] & += & moment[i].M1[k] & \forall k \in \{0,1,2\} \\ moment[j].M2[k] & += & moment[i].M2[k] & \forall k \in \{0,1,2\} \end{array}$$

La mise à jour des listes de voisins nécessite de :

1. Fusionner les listes de voisins des deux régions fusionnées,
2. Mettre à jour les listes de voisins des régions adjacentes aux régions fusionnées.

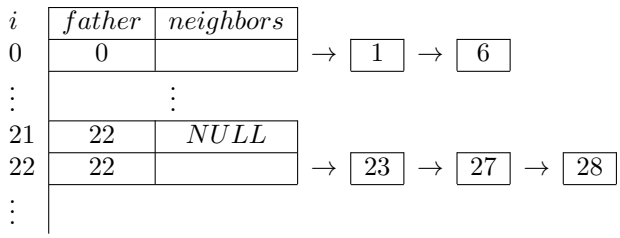
La fusion des listes de voisins de deux blocks i et j avec $i < j$ consite à intégrer à la liste des voisins de j les voisins de i, moins j ainsi que les blocks qui étaient déjà adjacents à j (chaque voisin apparaît une seule dois dans une

liste). En notation ensembliste si N_i et N_j désignent l'ensemble des voisins de i et j on obtient :

$$N_j = N_j \cup N_i - \{j\}$$

étant entendu que chaque élément de N_j apparait une seule fois.

Ainsi si l'on fusionne les régions 21 et 22 de l'exemple fourni en section 4 on obtient :



La mise à jour de l'erreur de partition s'effectue simplement en appliquant l'équation 1.

1. Définir la fonction :

   `void RAG_merge_region(RAG,int,int)`

   qui fusionne les deux régions passées en paramètres en effectuant :

   (a) La mise à jour du tableau `father`,

   (b) La mise à jour des moments,

   (c) La mise à jour des listes des deux régions fusionnées,

   (d) La mise à jour de l'erreur de partition.

   La mise à jour des moments et des listes de voisins se fera à l'aide de fonctions auxiliaires privées.

   On pourra utiliser des listes triées (attention de maintenir l'ordre) de voisins afin d'accélérer la fusion des listes de voisins.

   La mise à jour des listes de voisins des deux régions fusionnées n'est pas suffisante. En effet dans l'exemple précédent de fusion des blocks 21 et 22, le block 15 contient toujours 21 dans sa liste de voisins alors que 21 a été invalidé suite à sa fusion avec 22.

2. Modifiez le code de la fonction `RAG_give_closest_region` pour que lors du parcours de la liste des voisins de chaque block, chaque voisin soit remplacé par son père avant le calcul du coût de fusion. Ainsi le calcul du coût de la prochaine fusion met à jour les listes d'adjacences corrompues par la fusion précédente (cf point précédent).

7 Fusion itérative

1. En dehors du fichier RAG.c, définissez la fonction

   `void perform_merge(RAG,double)`

qui effectue ittérativement des fusions de régions jusqu'à ce que l'erreur de partition soit inférieure au seuil passé en second paramètre. On peut par exemple utiliser comme seuil un certain pourcentage de l'erreur quadratique de l'image initiale. Vous pouvez (éventuellement) proposer d'autres types de seuils si leur pertinence est justifiée dans le rapport.

2. Avec les même hypothèses que dans la section 5, quelle est la complexité de cette opération en supposant que l'on effectue K fusions ?

8 Mise sous forme normale des parents

Suite à l'application de la fonction `perform_merge` de nombreuses fusions ont pu être effectuées ce qui peut causer de nombreuses indirections. Par exemple, dans l'exemple de la section 4, si nous fusionnons 15 avec 21 puis 21 avec 22, nous aurons le tableau suivant en supposant que 22 n'est pas fusionné :

i	$father$
⋮	
15	21
⋮	
21	22
22	22
⋮	

1. Définissez la fonction :

   `extern void RAG_normalize_parents(RAG)`

   qui effectue un parcours retrograde (de la fin au début) du tableau `father` en remplaçant pour chaque indice `i`, `father[i]` par `father[father[i]]`. Expliquez pourquoi on est sûr, du fait du parcours rétrograde, que l'ensemble des blocks fusionnés en plusieurs étapes en un block d'indice maximum pointeront tous sur celui-ci.

9 Création de l'image de sortie

1. Définissez la fonction :

   `extern void RAG_give_mean_color(RAG, int, int*)`

   qui renvoie dans le dernier paramètre, la couleur moyenne du block parent du block dont l'indice est passé en second paramètre.

2. Définissez la fonction (dans le même fichier que `perform_merge`) :

   `image create_output_image(RAG)`

   qui crée une image où chaque block est affiché avec la couleur moyenne de son block parent (cf question précédente).

10 Intégration

1. Définissez un programme main. Qui récupère le nom d'une image, le nombre de blocks en ligne et en colonne ainsi qu'un seuil dans `argv[1], argv[2], arv[3] et argv[4]`. Ce programme :
   (a) Crée un RAG à partir de l'image (qui aura été chargé à partir de son nom) et des nombres de blocks par lignes et colonnes.
   (b) Appelle la fonction `perform_merge`, avec une valeur de seuil lue en paramètre
   (c) récupère l'image résultat avec `create_output_image` et l'enregistre dans un fichier (on pourra plus simplement l'envoyer sur le flot de sortie `stdout`).

11 Remarques

1. L'objet du rapport n'est pas de dupliquer les commentaires (qui doivent être présent dans le code) mais d'expliquer les différents choix possibles pour chaque problème et les avantages, inconvénients de la méthode retenue.
2. Utilisez également le rapport pour répondre aux questions (qui ne correspondent pas à du code) posées dans le sujet.
3. L'originalité et l'efficacité des solutions algorithmiques (montrées dans le rapport) ainsi que la qualité du code seront des éléments déterminant de la note. Chaque binôme devra faire une démonstration de son projet.
4. Des images seront disponibles sous /home/public/PROJET_FONDEMENTS/IMAGES/