Segmentation d'image avec Union-Find

La *segmentation d'image* consiste à partitionner une image en zones de pixels similaires appelées segments d'images. La segmentation a de nombreuses applications dans le domaine du traitement d'image : détection de zones et de contours, simplification d'une image, etc.

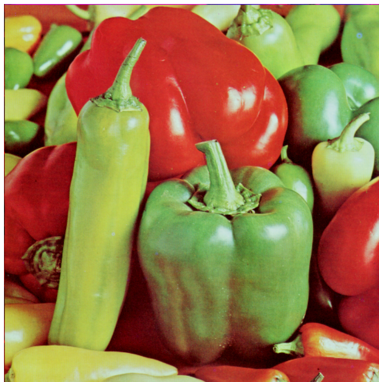

Figure 1 Une photo de poivrons

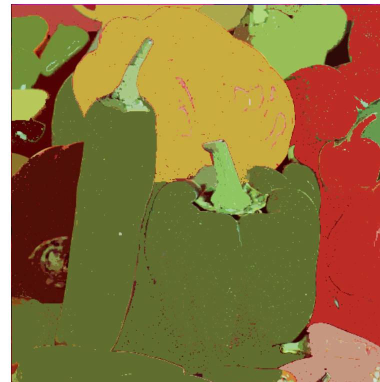

Figure 2 La photo segmentée

Dans ce TP, on propose d'implémenter une méthode de segmentation ascendante utilisant la structure de donnée Union-Find ([Fioro *et al.*, 2000]). Le principe est simple : initialement chaque pixel est un singleton, puis on procède à l'union des pixels voisins dont les couleurs se ressemblent.

1 Le projet C

On vous donne un squelette du projet sous forme d'une archive `imgseg_eleve.zip`. Dans le dossier du projet on trouve :

- des images de tests au format `.ppm`
- des fichiers d'implémentation `.c`
- des fichiers d'en-tête `.h`
- un fichier `Makefile` contenant les instructions de compilation

Le fichier `Makefile` permet une compilation simplifiée à l'aide de l'outil `make`. La commande `make` permet de compiler l'intégralité du projet en une seule commande. En cas de modification d'un ou plusieurs fichiers sources, un nouvel appel à la commande `make` recompilera automatiquement les parties de projet modifiées. La commande `make clean` nettoie le répertoire de travail en supprimant les fichiers compilés `.o`. Je vous invite à observer le contenu du fichier `Makefile`, la maîtrise de cet outil de compilation n'est pas un attendu du programme.

2 Le format d'image PPM

Notre programme travaillera sur des images enregistrées sous le format d'image *portable pixmap format* (ppm). L'avantage de ce format est qu'il est plutôt simple comparé aux autres, un fichier image ppm contient :

- un *nombre magique* sur deux octets : `P6`
- des caractères d'espacement (espaces, tabulations, retour à la ligne, etc)
- la largeur `width` de l'image en nombre de pixels écrits en texte ASCII
- des caractères d'espacement
- la hauteur `height` de l'image en pixels
- des caractères d'espacement
- la valeur maximale possibe pour une donnée pixel `maxval`
- un caractère d'espacement
- les données de l'image (pixels) encodées sous forme d'une liste d'entiers de 8 bits ou 16 bits.

On utilisera le type suivant, défini dans `image.h`, pour représenter une image ppm :

```
struct image_s {
    int width; // image width in pixels
    int height; // image height in pixels
    int maxval; // color maximum value
    uint16_t* data; // image data
};

typedef struct image_s image;
```

Le champ `data` contient les données des pixels sous forme d'une liste de pixels lus de gauche à droite et de haut en bas dans l'image. Chaque pixel est un triplet (r, g, b) de trois entiers compris entre 0 et `maxval`. Ainsi `data` contient la liste : `r0, g0, b0, r1, g1, b1, r2, g2, b2, ...`. La valeur `maxval` est inférieure strictement à 65536 ce qui garantie de pouvoir écrire les valeurs sur 16 bits.

Question 1

Dans le fichier `image.c`, compléter le corps de la fonction `check_magicnumber`. Cette fonction prend en entrée un argument `file` qui est un descripteur de fichier ouvert, et une chaîne de caractères `magic` qui est le nombre magique à vérifier. Elle doit avoir le comportement suivant :

- créer un tableau local de caractères `buffer` de longueur 3
- lire exactement 2 caractères du fichier à l'aide de `getc`
- si la lecture réussit (pas de retour `EOF`) on recopie les caractères lus dans `buffer`
- on ajoute le caractère `\0` en fin de `buffer`
- on compare le `buffer` et le `magic`
- si tout réussi on retourne `true` sinon `false`

Question 2

Dans le fichier `main.c`, écrire la fonction `main` réalisant les étapes suivantes :

- Vérifier que le nombre d'arguments en entrée du programme est 2 (`argc`).
- Charger l'image dont le nom est donné en entrée du programme (dans `argv[1]`) à l'aide des fonctions déclarées dans `image.h`
- Afficher à l'écran les meta données de l'image
- Afficher les 10 premiers pixels (10 triplets r, g, b)

Compiler avec `make` et tester cette première version de programme.

Question 3

Implémenter les fonctions `get_pixel` et `set_pixel` qui permettent de récupérer ou de fixer une des valeurs `RED`, `GREEN` ou `BLUE` du pixel de coordonnées (i, j).

Question 4

Modifier la fonction `main` ainsi :

- charger l'image passée en paramètre du programme dans une variable `img`
- réaliser une copie de l'image dans `img2` à l'aide des fonctions déclarées dans `image.h`
- modifier `img2` en mettant toutes les composantes `GREEN` et `BLUE` à 0 pour tout pixel
- enregistrer l'image dans 'all_red.ppm' et admirer le résultat

Pour savoir si deux pixels (r_1, g_1, b_1) et (r_2, g_2, b_2) ont des couleurs proches, on utilisera la distance suivante :

$$d = \frac{\sqrt{0.3(r_1 - r_2)^2 + 0.6(g_1 - g_2)^2 + 0.1(b_1 - b_2)^2}}{\texttt{maxval}}$$

qui est comprise entre 0 et 1.

Question 5

Implémenter la fonction `pixel_distance` calculant la distance colorimétrique entre deux pixels d'une image.

3 La structure Union-Find

On implémentera la structure Union-Find à l'aide d'une forêt, comme on l'a fait en cours. Les types sont définis dans `unionfind.h` et les fonctions implémentées dans `unionfind.c`.

Question 6

Compléter le fichier `Makefile` pour y inclure la compilation de `unionfind.o`. Il faudra aussi ajouter `unionfind.o` à la liste des fichiers objets à lier dans la commande de compilation principale.

Question 7

Implémenter les fonctions manquantes de `unionfind.c` :

- `uf_find` : cette fonction devra implémenter l'optimisation de compression des chemins
- `uf_union` : cette fonction devra implémenter l'optimisation sur la hauteur des arbres, si les éléments sont déjà dans la même partie on ne doit rien faire.

4 L'algorithme de segmentation

Dans le fichier `main.c` on codera la segmentation de l'image passée en entrée du programme. On procédera ainsi :

1. On vérifie que le programme a 2 entrées : un nom de fichier et une valeur de tolérance `tol` de type `double`[1]
2. On charge l'image et on affiche ses paramètres.
3. On créé une structure Union-Find où chaque numéro de pixel est initialement un singleton.
4. On affiche le nombre de composantes connexes.
5. Pour tous les pixels p de l'image, sauf ceux de la dernière ligne, on réalise une `union` avec le pixel q situé en dessous de lui, seulement lorsque $d(p,q) < \texttt{tol}$.
6. Pour tous les pixels p de l'image, sauf ceux de la dernière colonne, on réalise une `union` avec le pixel q situé à sa droite, seulement lorsque $d(p,q) < \texttt{tol}$.
7. On affiche le nombre de composantes connexes.
8. On créé une copie `img2` de `img`.
9. Dans `img2` on fixe la couleur de chaque pixel à celle de son représentant dans `img`
10. On sauvegarde `img2` dans `segmentation.ppm`

Question 8

Implémenter cet algorithme dans `main.c`.

[1] la conversion s'obtient avec l'appel `strtod(texte, NULL)`