ING5SE – 2020
Projet : étiquetage en composantes connexes

Etienne Hamelin (ehamelin@inseec-edu.com), Alexandre Berne (aberne@inseec-edu.com)

1 Introduction

Ce projet est un peu un « grand TP », il vous demandera de fouiller un peu plus les aspects algorithmiques, et non pas seulement le langage de programmation.

Prenez donc d'abord le temps de comprendre, puis de résoudre, le problème algorithmique. Ensuite, la programmation sera une traduction « triviale » de la solution algorithmique.

2 Présentation du projet

Nous vous proposons de travailler sur l'analyse en composantes connexes, c'est un algorithme souvent utilisé en vision par ordinateur et en robotique.

Derrière ce terme barbare se cache une notion intuitivement très simple : distinguer et compter des objets dans une image. Un exemple très simple : dans l'image noir et blanc ci-dessous, il est évident pour un œil humain de distinguer chaque « blob » ou objet. Pour un ordinateur, distinguer ces objets nécessite un algorithme pas tout à fait trivial. Qu'il s'agisse de compter des objets sur une ligne de production en usine, ou distinguer des obstacles sur l'image caméra d'un robot mobile, on comprend le besoin de faire cette opération rapidement, donc en parallélisant.

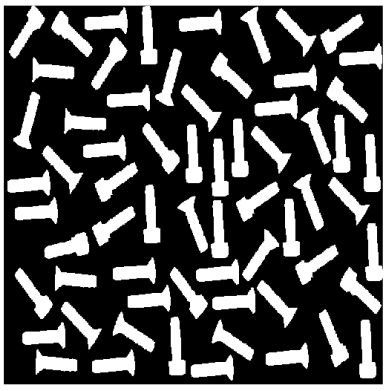

Figure 1 – Image binaire: boulons

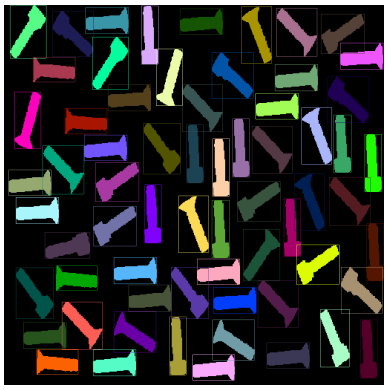

Figure 2 – Boulons avec les composantes connexes annotées

2.1 Généralités : manipulation d'images

Dans ce projet, nous allons donc manipuler des images. Une image de largeur (*width*) W et de hauteur (*height*) H est un tableau à 2 dimensions, de dimension $W \times H$, dont chaque élément $I_{x,y}$ est la couleur du pixel de coordonnées (x, y).

Par convention, en informatique on représente généralement le pixel de coordonnées $(0,0)$ en haut gauche de l'image; le coin en bas à droite est donc $(W-1, H-1)$.

2.1.1 La librairie fournie

Nous vous fournissons une petite librairie permettant de lire et manipuler des fichiers images au format NetBPM[1], que vous avez déjà utilisé dans le TP sur le gradient. Cette librairie permet de manipuler différents formats d'images :

- binaires noir et blanc (`type=IMAGE_BITMAP`),
- niveaux de gris (encodés sur 8 ou sur 16 bits, `IMAGE_GRAYSCALE_8` et `IMAGE_GRAYSCALE_16`),
- images couleurs, où les composantes rouge, vert, et bleu, sont encodées sur 8bits (`IMAGE_RGB_888`).

2.1.2 Principales APIs

Le code ci-dessous résume les principales API que vous pourrez être amenés à utiliser dans ce projet.

```
/* Créer une image 320x200 pixels noir & blanc */
image_t *img = image_new(320, 200, IMAGE_BITMAP);

/* lire, écrire un pixel d'une image noir & blanc */
bool b = image_bmp_getpixel(img, x, y).bit;
image_bmp_setpixel(img, x, y, (color_t){.bit = 1});

/* Créer une image couleur */
image_t *img = image_new(320, 200, IMAGE_RGB_888);

/* lire, écrire un pixel d'une image couleur */
uint8_t r = image_rgb_getpixel(img, x, y).rgb.r;
image_rgb_setpixel(img, x, y, (color_t){.rgb = {.r = 255, .g = 128, .b =
0}});

/* Paramètres utiles */
img->width, img->height

/* Lire, écrire un fichier NetBPM */
image_t *img = image_new_open("file.ppm");
image_save_ascii(img, "file.ppm")
image_save_binary(img, "file.ppm")
```

[1] Voir https://en.wikipedia.org/wiki/Netpbm

3 Etiquetage en composantes connexes

L'étiquetage en composantes connexes consiste à attribuer un même numéro (une étiquette) à chaque *groupe de pixels voisins de la même couleur*. Ici nous nous restreindrons à des images d'entrée en noir et blanc : on peut supposer par exemple qu'un algorithme a préalablement filtré le fond (*background*).

3.1 Définitions

Si besoin, nous donnons ici les définitions rigoureuses nécessaires pour définir les composantes connexes.

3.1.1 Notion de connexité

Deux pixels $p_A = (x_A, y_A)$ et $p_B = (x_B, y_B)$ sont connexes (on dit également voisins, ou adjacents), ssi $\begin{cases} ou\ bien\ x_A = x_B et\ y_A = y_B \pm 1 \\ ou\ bien\ x_A = x_B \pm 1\ et\ y_A = y_B \end{cases}$. On appelle souvent nord (N), sud (S), est (E) et ouest (W) les 4 voisins d'un pixel.

Figure 3 - En jaune, les pixels adjacents, ou connexes, au pixel rouge

3.1.2 Composante connexe

Une composante connexe $\mathcal{A}$ de l'image I est un ensemble de pixels connexes de même couleur, c'est-à-dire que pour toute paire de pixels $p_A, p_B \in \mathcal{A}$, il existe un chemin de pixels connexes de p_A à p_B, et ayant la même couleur dans l'image I.

4 L'algorithme de Rosenfeld & Pfalz

L'algorithme de Rosenfeld & Pfalz (proposé en 1966) est l'un des plus simples pour étiqueter des composantes connexes d'une image. Il se déroule en 3 temps, ébauchés puis détaillées ci-dessous.

1. En une passe sur l'image, on marque les pixels avec des étiquettes temporaires.

   A ce stade, plusieurs étiquettes peuvent désigner la même composante connexe : on note dans une table d'équivalence les étiquettes qui désignent la même composante connexe (classe d'équivalence),

2. Par analyse de la table d'équivalence, on désigne une étiquette unique et définitive pour chaque classe d'étiquettes.
3. On remplace les étiquettes temporaires par l'étiquette finale de la composante connexe.

4.1 Définition algorithmique

4.1.1 Algorithme: marquage initial.

Algorithme	**Commentaire**
Entrées : image $\boldsymbol{I}$ de taille $\boldsymbol{W \times H}$ **Sorties** : - étiquettes temporaires E (de taille $W \times H$), - nombre d'étiquettes temporaires utilisées n_E, - table d'équivalence T **Initialisation** : - $\boldsymbol{n_E \leftarrow 0}$	

- $\boldsymbol{T} \leftarrow [\mathbf{0}, \dots, \mathbf{0}]$ tableau de taille n_E^{max} - $E \leftarrow \left[[0, \dots 0], \dots, [0, \dots, 0]\right]$ de taille $W \times H$ **Procédure** : - pour $\boldsymbol{y}$ de $\mathbf{0}$ à $\boldsymbol{H} - \mathbf{1}$: o pour $\boldsymbol{x}$ de $\mathbf{0}$ à $\boldsymbol{W} - \mathbf{1}$: ▪ si $\boldsymbol{I_{x,y}} = \mathbf{0}$: • $\boldsymbol{E_{x,y}} \leftarrow \mathbf{0}$ ▪ sinon : • $\boldsymbol{e_N} = \boldsymbol{E_{x,y-1}}$ • $\boldsymbol{e_W} = \boldsymbol{E_{x-1,y}}$ • si $\boldsymbol{e_N} = \mathbf{0}$ et $\boldsymbol{e_W} = \mathbf{0}$: o $\boldsymbol{n_E} \leftarrow \boldsymbol{n_E} + \mathbf{1}$ o $\boldsymbol{T}[\boldsymbol{n_E}] \leftarrow \boldsymbol{n_E}$ o $\boldsymbol{E_{x,y}} \leftarrow \boldsymbol{n_E}$ • sinon: o $\boldsymbol{E_{x,y}} = \boldsymbol{minNonZero}(\boldsymbol{e_N}, \boldsymbol{e_W})$ • si $e_N > 0$ et $e_W > 0$ et $e_N \neq e_W$: o $\boldsymbol{Union}(\boldsymbol{T}, \boldsymbol{e_N}, \boldsymbol{e_W})$	*Balayage des pixels :* *Background :* *étiquette = 0* *Foreground :* *Etiquettes des voisins déjà traités* *Pas de voisin étiqueté :* *On crée une nouvelle étiquette* *Qui n'a pas d'équivalence* *Si au moins 1 voisin étiqueté :* *On met la plus petite des étiquettes non-nulle* *Si les deux voisins ont des étiquettes différentes : on réunit leurs classes*

4.1.2 Algorithme: Find

Algorithme	Commentaire
Entrées : table d'équivalence $\boldsymbol{T}$, étiquette $\boldsymbol{e}$ **Sorties** : $\boldsymbol{r}$, étiquette de la racine de la classe d'équivalence contenant $\boldsymbol{e}$ **Initialisation :** - $\boldsymbol{r} \leftarrow \boldsymbol{e}$ **Procédure** : - tant que $\boldsymbol{T}[\boldsymbol{r}] < \boldsymbol{r}$: o $\boldsymbol{r} \leftarrow \boldsymbol{T}[\boldsymbol{r}]$	Cette récursion termine forcément, puisque par construction $T[e]$ est toujours $\leq e$.

4.1.3 Algorithme: Union

Algorithme	Commentaire
Entrées : table d'équivalence $\boldsymbol{T}$, étiquettes e_1 et e_2 **Sorties** : table d'équivalence $\boldsymbol{T}$ modifiée **Initialisation :** - $r_1 \leftarrow Find(T, e_1)$ - $r_2 \leftarrow Find(T, e_2)$ **Procédure** : - Si $\boldsymbol{r_1} < \boldsymbol{r_2}$: ▪ $\boldsymbol{T}[\boldsymbol{r_2}] = \boldsymbol{r_1}$ - Sinon : ▪ $\boldsymbol{T}[\boldsymbol{r_1}] = \boldsymbol{r_2}$	On recherche les racines des classes de e_1 et e_2 On note l'équivalence $r_2 \sim r_1$

4.1.4 Algorithme: Renum

Algorithme	Commentaire
Entrées : table d'équivalence $\boldsymbol{T}$, étiquettes e_1 et e_2 **Sorties** : table N des renumérotation d'étiquettes ; n_C nombre des classes d'équivalences **Initialisation :**	

- $N \leftarrow [0, \dots, 0]$ de taille n_E^{max} - $n_C \leftarrow 0$ **Procédure** : - Pour e de 1 à n_E **:** o Si $T[e] = e$: ▪ $n_C \leftarrow n_C + 1$ ▪ $N[e] \leftarrow n_C$ o Sinon : ▪ $N[e] \leftarrow N[T[e]]$	$T[e] = e$: donc e est la racine d'un arbre Sinon : e pointe vers l'étiquette de son parent $T[e]$. Or $T[e] < e$, donc $N[T[e]]$ a déjà été calculé dans les itérations précédentes.

5 Le code fourni

L'implémentation fournie est séquentielle, et effectue correctement l'algorithme de Rosenfeld (en tous cas, s'il y a une erreur merci de nous l'indiquer rapidement !).

5.1 Commandes utiles pour débuter

Voici quelques commandes pour compiler & executer le code fourni.

Nettoyer: `make clean`

Compiler: `make`

Exécuter en mode debug: le code affiche des données intermédiaires, génère des fichiers additionnels:

```
./main img/test1.pbm debug
```

Si vous avez besoin d'enregistrer les sorties (stdout et stderr) du programme dans un fichier (la directive `2>&1` redirige stderr vers stdout, puis `tee main.log` duplique le flux stdout vers le fichier `main.log`):

```
./main img/test1.pbm debug 2>&1 |tee main.log
```

Mesurer les performances (sans le mode debug):

```
usr/bin/time ./main img/test1.pbm
```

Visualiser les fichiers résultats & fichiers temporaires:

```
display color.ppm
vim classes.pgm
vim tags.pgm
```

Si vous rencontrez des problèmes mémoire (segfault, free, …):

```
valgrind ./main img/test1.pbm debug
```

Pour transformer un fichier NetBPM en un autre format (et réciproquement), pour exporter vos résultats ou analyser les composantes connexes d'autres images que celles fournies:

```
convert fichier.png fichier.ppm
```

(au besoin) `apt-get install imagemagick`

5.2 Structure du code

Le code fourni contient les fichiers suivants :

```
├── img                 quelques exemples d'images noir & blanc à analyser
│   ├── boulons.pbm     de « grosses » images pour tester les performances
│   ├── cadastre.pbm    idem
│   ├── ocr.pbm         idem
│   ├── test0.pbm       des « petits » exemples, pour comprendre/tester comment ça marche
│   └── test1.pbm
├── inc                   les fichiers d'en-têtes
│   ├── image_connected_components.h
│   ├── image_file_io.h
│   ├── image.h
│   ├── image_lib.h
│   ├── pixel.h
│   └── utils.h
├── Makefile
└── src                 les codes source
    ├── image.c         manipulation de fichiers images
    ├── image_connected_components.c
    ├── image_file_io.c       lecture & enregistrement de fichiers PBM/PGM/PPM
    ├── main.c          récupérer les arguments en ligne de commande, lancer l'étiquetage
    └── pixel.c         conversion de formats de couleurs et formats de pixels
```

Les modules (source+en-tête associé) *pixel*, *image*, et *image_file_io* correspondent à la librairie de manipulation d'images. Vous n'avez pas besoin de les modifier. Vous devriez pouvoir faire toutes vos modifications dans *image_connected_components.c*.

Les principales fonctions d'intérêt sont appelées dans l'ordre ci-dessous :

```
main() [main.c]
└── test_image_connected_components [main.c]
    └── image_connected_components [image_connected_components.c]
        ├── ccl_temp_tag 1° étape, balayage, attribution d'étiquettes temporaires
        ├── ccl_reduce_equivalences 2° étape : réduction des étiquettes équivalences
        ├── ccl_retag 3° étape, remplacement des étiquettes temporaires par définitives
        ├── ccl_analyze analyse (des dimensions) des composantes connexes
        └── ccl_draw_colors génération d'une image en couleurs
```

6 Questions

Q1 (0pts) : Combien votre machine a-t-elle de cœurs physiques ? de cœurs logiques ? de RAM ? Quel système utilisez-vous ? (e.g. Linux natif, WSL/Windows10, Mac, Machine virtuelle/Windows 8, etc.)

Q2 (3pts) : Identifiez les variables dans le code fourni correspondant aux matrices/tableaux/variables des algorithmes décrits ci-dessus, expliquez leur rôle.

Symbole dans l'algorithme	*Nom de la variable dans le code*	*Rôle (expliquer)*
I		
E		
T		
N		
n_E^{max}		
n_E		
n_C		

Q3 (4pts). Expliquez quelles fonctions (parmi les sous-fonctions appelées par image_connected_components) prennent le plus de temps de calcul et bénéficieraient d'une parallélisation.
Parallélisez (pour l'instant) les fonctions suivantes, dans le dossier Q4 : ccl_retag, ccl_analyze, ccl_draw_colors.

Q4 (3pts) : Décrivez sous forme de texte, illustrez avec des schémas au besoin, comment vous allez parallélisez la fonction ccl_temp_tag.

Q5 (7pts) : Dans le dossier Q5, copiez votre solution Q4, puis implémentez ccl_temp_tag de façon parallèle (si besoin, mettez à jour votre réponse à la question précédente pour que le design et l'implém soient cohérents…)
Nous noterons en fonction de : si votre code compile et produit des sorties correctes, si la solution est performante (nous testerons sur notre machine à 12 cœurs logiques, et sur d'autres exemples que ceux fournis), si votre code est bien écrit, commenté, agréable à lire et facile à comprendre.

Q6 (3pts) : Mesurez les temps d'exécution, en utilisant comme fichier d'entrée l'image cadastre.pbm, en faisant varier le nombre de threads de 1 à 10, mesurez les temps d'exécution. Tracez les courbes $S(n) = \frac{Real(1)}{Real(n)}$ *et* $\gamma(n) = \frac{S(n)}{n}$.

Expliquez ce que représentent $S(n)$ *et* $\gamma(n)$.