ING5SE – 2022
Project : connected components labelling

Etienne Hamelin (ehamelin@omnesintervenant.com),

Alexandre Berne (aberne@ omnesintervenant.com)

1 Introduction

This project is like a bigger lab. It will require to dig deeper into algorithmic aspects ; not only programming language. Take the time to understand, then solve, the algorithmic problem. Then, programming the solution becomes trivial.

2 Presentation

You will work on connected components lablling, an algorithm often used in computer vision and robotics. The purpose is to distinguish and count, objects in a picture. In the following black/white image, it is obvious to a human eye to distinguish every single bolt, and counting them is (more-or-less) simple. To a computer however, distinguishing them requires some non-trivial work.

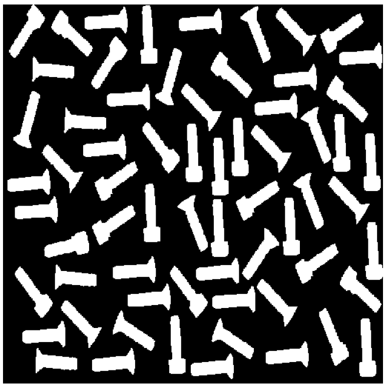

Figure 1 – binary image : bolts

Figure 2 – color output : every single bolt identified

2.1 Reminder: images & computer vision

An image of width W and height H is an array or dimension $W \times H$, where each element $I_{x,y}$ is the color of the pixel (x, y). For historical reasons, in computer systems the origin, i.e. coordinates $(0,0)$, is usually located on the top-left corner, with the y axis pointing down ; the bottom-right corner is at coordinates $(W - 1, H - 1)$.

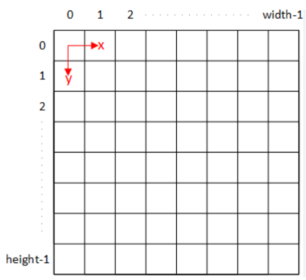

Figure 3 – image pixels representation

2.1.1 Proposed library

We provide you with a small library, that supports reading and writing NetBPM image files. You should use the `display` command to view images, and `convert` to transform to/from other image formats, both installed with the `imagemagick` package.

This library supports several image types :

- Black or white, or binary images (`type=IMAGE_BITMAP`),
- Grayscale images, with gray levels encoded either with 8 or 16 bits (`IMAGE_GRAYSCALE_8` and `IMAGE_GRAYSCALE_16`)
- Color images, where the red, green, blue, components of each pixel are encoded with 8 bits in that order (`IMAGE_RGB_888`).

2.1.2 Main library APIs

The example code below gives is a reference on how to use the library.

```
/* Create an object for a 320 (width) x 200 (height) binary image */
image_t *img = image_new(320, 200, IMAGE_BITMAP);

/* read, write, the binary color of a pixel at (x,y) */
bool b = image_bmp_getpixel(img, x, y).bit;
image_bmp_setpixel(img, x, y, (color_t){.bit = 1});

/* Create a color image */
image_t *img = image_new(320, 200, IMAGE_RGB_888);

/* Read, write the color of a RGB_888 pixel */
uint8_t r = image_rgb_getpixel(img, x, y).rgb.r;
image_rgb_setpixel(img, x, y,
    (color_t){.rgb = {.r = 255, .g = 255, .b = 0}});

/* Read the dimensions of an image object */
int w = img->width;
int h = img->height;

/* Check is coordinates (x,y) are within (0,0)…(w-1,h-1) range */
image_coord_check(img,x,y)

/* Read, write a NetBPM image file */
image_t *img = image_new_open("file.ppm");
image_save_binary(img, "file.ppm")
```

3 Connex components labelling

For simplification, we take as input a binary black/white image, where white (foreground) indicates the presence of an object, and black is the background. Connex components labelling consists in giving a number (a tag) to each white pixel, so that all adjacent foreground pixels share the same tag.

3.1 Definitions

3.1.1 Connectedness

Pixels $p_A = (x_A, y_A)$ and $p_B = (x_B, y_B)$ are said to be adjacent iff $\begin{cases} x_A = x_B \text{ and } y_A = y_B \pm 1 \\ \text{or} \\ x_A = x_B \pm 1 \text{ and } y_A = y_B \end{cases}$.

We usually denote with north, south, east, west (N/S/E/W) the 4 adjacent pixels of a given pixel.

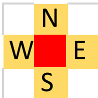

Figure 4 – Yellow pixels are adjacent to the red one

3.1.2 Connex component

A connex component $\mathcal{A}$ of image I is a set of pixels so that for each pair of pixels $p_A, p_B \in \mathcal{A}$, there exists a continuous path of adjacent pixels from p_A to p_B that share the same color in image I.

4 The Rosenfeld & Pfalz algorithm

The Rosenfeld & Pfalz algorithm (proposed en 1966) is one of the simplest algorithms for connex components labelling. It runs in 3 steps:

1. First pass, mark pixels with a temporary tag,

   At that stage, a single connex component might bear several distinct tags; but we note in a table when several tags are linked to the same connected component: an equivalence table

2. Analyze the equivalence table, in order to assign a definitive class tag to every temporary tag
3. Replace temporary tags with the definitive class tag.

4.1 Algorithm definition

4.1.1 Algorithm: initial marking

Algorithm	Comments
Inputs : image $\boldsymbol{I}$, of size $W \times H$ **Outputs** : - image $\boldsymbol{E}$ of temporary tags (size $W \times H$), - number of temporary tags n_E, - equivalence table $\boldsymbol{T}$ (of size n_E^{max}) **Initialization** : - $n_E \leftarrow 0$ - $\boldsymbol{T} \leftarrow [0, \dots, 0]$ - $\boldsymbol{E} \leftarrow \big[[0, \dots 0], \dots, [0, \dots, 0]\big]$ **Procedure** :	
- For y from 0 to $H-1$:	*For each pixel:*
○ For x from 0 to $W-1$:	
▪ if $I_{x,y} = 0$:	*Background ? tag = 0*

• $E_{x,y} \leftarrow 0$ ▪ else : • $e_N = E_{x,y-1}$ • $e_W = E_{x-1,y}$ • if $e_N = 0$ and $e_W = 0$: ○ $n_E \leftarrow n_E + 1$ ○ $T[n_E] \leftarrow n_E$ ○ $E_{x,y} \leftarrow n_E$ • else: ○ $E_{x,y} = minNonZero(e_N, e_W)$ • if $e_N > 0$ and $e_W > 0$ and $e_N \neq e_W$: ○ $Union(T, e_N, e_W)$	*Foreground ?* *Read the temp. tag of north/east pixels (already processed)* *N/E neighbors not yet tagged?* *Create new tag* *With no equivalence set* *N or E is already tagged? Set the minimal non-zero tag* *N and E have different tags? Note that these tags are now equivalent.*

4.1.2 Algorithm: Find

Algorithm	Comments
Inputs : equivalence table $\boldsymbol{T}$, tag e **Outputs** : r, the root tag of equivalence class that contains tag e **Initialization :** - $r \leftarrow e$ **Procedure** : - while $T[r] < r$: ○ $r \leftarrow T[r]$	Note that this recursion terminates always, since by construction $\forall\ e$, $T[e] \leq e$.

4.1.3 Algorithm: Union

Algorithm	Comments
Inputs: equivalence table $\boldsymbol{T}$, two tags e_1 and e_2 **Outputs** : modified equivalence table $\boldsymbol{T}$ **Initialization :** - $\boldsymbol{r_1} \leftarrow \boldsymbol{Find}(\boldsymbol{T}, \boldsymbol{e_1})$ - $\boldsymbol{r_2} \leftarrow \boldsymbol{Find}(\boldsymbol{T}, \boldsymbol{e_2})$ **Procedure** : - if $r_1 < r_2$: ▪ $T[r_2] = r_1$ - else : ▪ $T[r_1] = r_2$	Find the root of the equivalence classes containing tags e_1 and e_2 Mark equivalence $r_2 \sim r_1$

4.1.4 Algorithm: Renum

Algorithm	Comments
Inputs : table $\boldsymbol{T}$ **Outputs** : table $\boldsymbol{N}$; number of connected components n_C **Initialization :** - $\boldsymbol{N} \leftarrow [0, \ldots, 0]$ of size n_E^{max} - $n_C \leftarrow 0$ **Procedure** : - For e from 1 to n_E : ○ if $T[e] = e$: ▪ $n_C \leftarrow n_C + 1$ ▪ $N[e] \leftarrow n_C$ ○ else : ▪ $N[e] \leftarrow N[T[e]]$	If $T[e] = e$: tag e is the root of an equivalence class; create a new equivalence class number (definitive tag).

	Otherwise: tag e is the child of $T[e]$, and should use the same definitive tag

5 Provided implementation

You will find attached a correct sequential implementation of the Rosenfelf&Pfalz algorithm

5.1 Useful commands

Clean up temporary files: `make clean`

Run in debug mode: the code prints out intermediate results, and generates additional visualization files.

```
make clean && make DEBUG=1
./main img/test1.pbm
```

Visualize intermediate data outputs:

```
code tags.pgm         # read temporary tags
code classes.pgm      # read definitive tags
display color.ppm     # read the color-coded tags
```

Run in performance mode (`DEBUG=0`): the code prints out less detail, and does not generate color images or temporary tags→ *use this mode to measure execution times*.

```
make clean && make DEBUG=0
./main img/test1.pbm
```

If you encounter memory problems:

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

If you need to convert a ppm (color), pgm (grayscale) or pbm (binary) image to/from another file format:

```
convert file.ppm file.png
```

To convert an image of yours to binary black/white:

```
convert file.png -threshold 50% file.pbm
```

5.2 Code structure

```
├── img                 a few test files
│   ├── boulons.pbm     a few “larger” images to test performances
│   ├── cadastre.pbm    idem
│   ├── ocr.pbm         idem
│   ├── test0.pbm       smalled examples, to understand how it works (or doesn’t)
│   └── test1.pbm
├── inc                  header files
│   ├── image_connected_components.h
│   ├── image_file_io.h
│   ├── image.h
```

```
│    ├── image_lib.h
│    ├── pixel.h
│    └── utils.h
├── Makefile
└── src                    source code
     ├── main.c            launch CCL algorithm
     ├── image.c           basic image manipulation
     ├── image_connected_components.c   the main CCL algorithm
     ├── image_file_io.c        read/write NetBPM files
     └── pixel.c           pixel color format conversions
```

Please only change the `image_connected_components.c and main.c` files.

Here is the main sequence of functions:

```
main() [main.c]
└── test_image_connected_components [main.c]
     └── image_connected_components [image_connected_components.c]
          ├── ccl_temp_tag  Initial marking algorithm
          ├── ccl_reduce_equivalences  renum algorithm
          ├── ccl_retag  Definitive tag marking
          ├── ccl_analyze  Analyze each connected component
          └── ccl_draw_colors  Generate a color image output
```

6 Questions

Q1 (0pts) : Describe the machine you run this project on: how many physical/logical cores? What system environment? *(e.g. Linux natif, WSL/Windows10, Mac, Machine virtuelle/Windows 8, etc.)*

Q2 (3pts) : Identify the following variables between the algorithm above, and the code. Explain their role.

Symbol in algorithm	*Variable name in code*	*Role (explain in 1 sentence)*
I		
E		
T		
N		
n_E^{max}		
n_E		
n_C		

Q3. What functions take most time to run? *Explain, using drawing or diagrams if applicable, how you will parallelize these functions.*

Q4: Compile in performance mode, and measure the execution times on image "large.bpm", using 1 to 20 threads, and draw the curve $S(n) = \frac{Real(1)}{Real(n)}$.

Q5: Prepare an oral presentation, 10min long; in English, covering at least:

- What challenges you faced and how you solved them.
- Your results, in terms of performance & speedup,
- Try and illustrate with aesthetic examples ☺

For submitting, please insert your names into the Makefile, then use the make submit command to compress your program and report.