TD4: Connected components extraction

In this TP, we focus on connected components extraction algorithms. The idea would be to implement two variants and to compare them.

Preliminaries

We suppose that we have all the tools to load/save images in PGM. To identify the different connected regions, we will use gray levels. For example, pixels with gray level `42` will define one (or more) connected components.

In addition, we would like both algorithms to be parametrized by k for the $(k)-$adjacency definition.

Exercise 1 Connected components by double-scan

The algorithm consists on a two-pass scan of the image. During the first scan, we identify potential connected components and during the second one, we validate such components using a relabelling step. The main idea is to scan the image starting from top left using half-connectivity masks (cf Figure **??**).

In the following, the result of the algorithm will be an image L with one label per connected regions. These labels will be represented by letters. The algorithm is the following

- **First scan:** We apply one of the two half-masks. For each point p:
  - If one of its neighbours q has the same gray level, then we propagate the label ($L(p) = L(q)$).
  - If no neighbour has the same gray level, we create a new label and store it in $L(p)$.
  - If p has several neighbours with the same gray level but with different labels (labels a, b), we pick one of them for p (the "lowest" one, for example) and we keep track that both labels correspond to the same region (i.e. b is a synonym of a). so called *Collision list*. For example, you could propagate the lowest label in the neighbourhood and make all labels in this neighbourhood "point" to the lowest one.

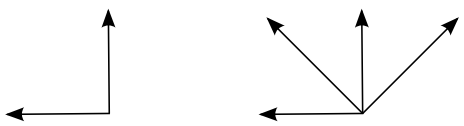

Figure 1: Half-mask for the (1)- and (0)-adjacency.

- Figure **??** illustrates the overall process with four temporary labels and only two remain after the pruning step. As illustrated in this figure, you could implement the collision list as a list where each node has a pointer to its *lowest* parent during the first scan.
- **Second scan:** Process the collision list such that each label now points to either to itself or to its root (for example, D is now connected to A instead of C). Then, for each pixel we simply remap the labels according to the pruned collision list.

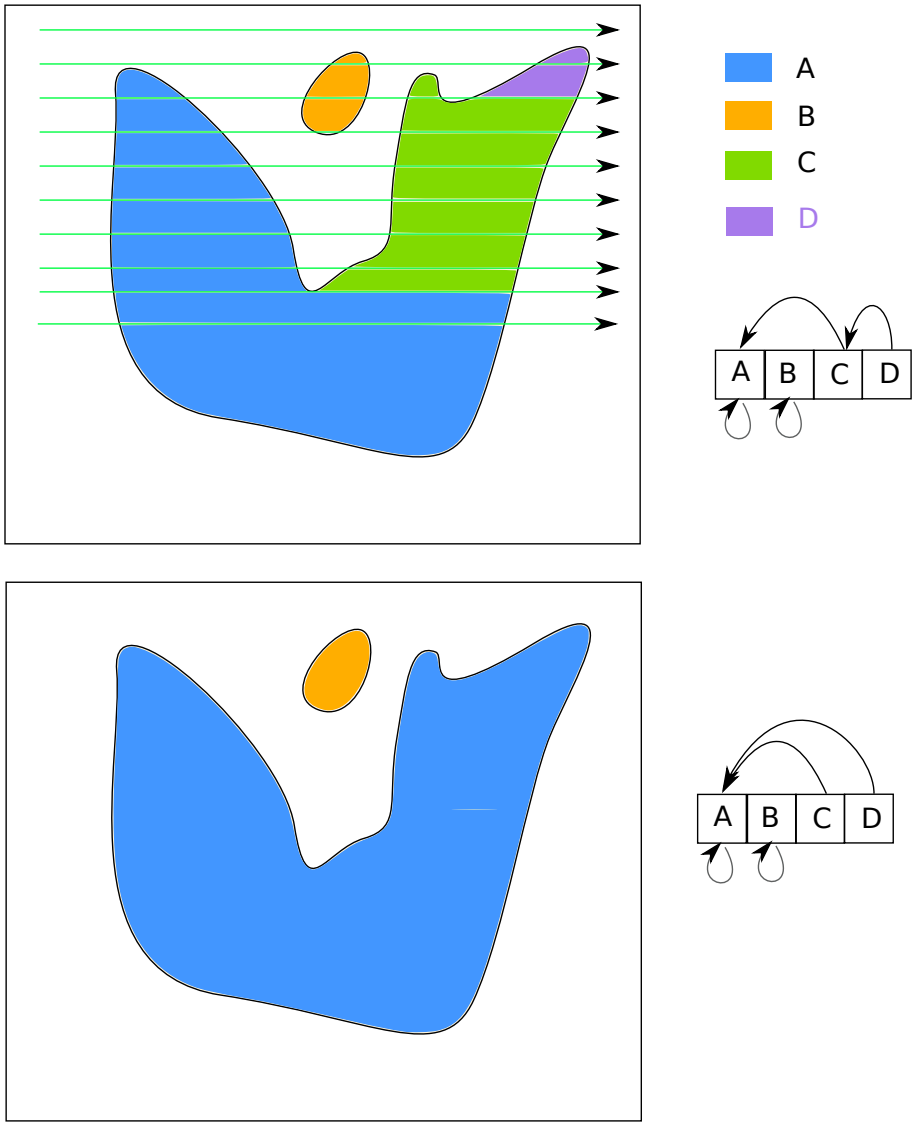

Figure 2: Result of the first pass and relabelling.

Questions:

1. Implement the algorithm described above, for the the (0)− and the (1)−adjacencies. Use this algorithm on a grayscale input PGM, then output the label map in PGM format.

2. What is the complexity of this algorithm (both memory/time) ?

Exercise 2 Connected components by Union-find structure

The *Union-find* structure is a very interesting data structure to represent and maintain disjoint sets. Given a set E of elements, such data structure have the following methods:

`MakeSet(x)` : creates a set with single element $x \in E$

`Find(x)` : returns the set containing x

`Union(x,y)` : compute the union of two disjoint sets.

Each disjoint set is represented by one of its elements in E. The main interest of this structure is that, for any sequence of size n, `Union` and `Find` have an amortized complexity of $O(\alpha(n).n)$ (where $\alpha(n)$ is the inverse of the Ackerman function).

The main idea is to represent the collection of sets as a forest of trees (encoded by the `rank` and `parent` data members) and to perform *path compression* when doing a `Find`. Here is the pseudo-code of each function:

```
function MakeSet(x)
     x.parent := x
     x.rank   := 0

 function Union(x, y)
     xRoot := Find(x)
     yRoot := Find(y)
     if xRoot == yRoot
         return
     if xRoot.rank < yRoot.rank
         xRoot.parent := yRoot
     else if xRoot.rank > yRoot.rank
         yRoot.parent := xRoot
     else
         yRoot.parent := xRoot
         xRoot.rank := xRoot.rank + 1

 function Find(x)
     if x.parent != x
        x.parent := Find(x.parent)
     return x.parent
```

Questions

1. Implement the union-find structure. You will have to define a C `struct` (or equivalent) to represent each element of E as a triplet: a rank (`unsigned int`) a value (which will be the gray value, `unsigned char` for instance) and a pointer to the parent (element of E).

2. Test the structure on small example.

3. Implement the connected components extraction algorithm using such structure. For example : for each pixel, create a set, then for each pixel again, we check the neighbours label and do the union if necessary. Finally, output the final labels using the Find for each pixel.

4. What is the computational cost of this algorithm (both memory/time) ?

5. Would it be possible to merge the two above mentioned algorithms ? What would be the computational cost ?

Exercise 3 Comparative evaluation

The idea is to do some experiments to compare the speed of both algorithms.

Questions:

1. (Image generation) Create two methods parametrized by an image size n and that returns images such that:

   - Random image: generate a $n \times n$ image with m ($m << n$) different gray level for each pixel (randomly selected)
   - *n-fork* image: all even (resp. odd) columns are set to 0 (resp. 1) and the last row contains only 0 pixels.

2. Test the connected components algorithms on these generated images and compare their efficiency. For example, you could generate several runs with increasing n and output a text file with two columns, n and elapsed time measured using for instance.

3. Conclude on the efficiency of the two previous algorithms.

Hints: To measure a delay in C, you can use the library `<time.h>`, `<sys/time.h>`[1].

- To declare a timer: `struct timespec myTimer;`
- To measure the current system time: `clock_gettime(CLOCK_REALTIME, &myTimer);`
- To get the informations: `myTimer` has 2 fields: `myTimer.tv_sec` (seconds) and `myTimer.tv_nsec` (nanoseconds)

[1] If you are using gcc, don't forget the "-lrt" option