Image segmentation project
MIF17 - image analysis course

Judith Millet
Department of Computer Science
University Claude Bernard
43 Bd du 11 Novembre 1918, 69100 Villeurbanne
judith.millet@etu.univ-lyon1.fr

Emie Lafourcade
Department of Computer Science
University Claude Bernard
43 Bd du 11 Novembre 1918, 69100 Villeurbanne
emie.lafourcade@etu.univ-lyon1.fr

I. INTRODUCTION

During this project, we had to develop an image segmentation method. It is based on a color segmentation algorithm which combines region growing and region merging processes. Region growth is indeed the process of combining pixels or regions exhibiting similar properties into larger regions according to predefined growth criteria. The similarity between pixels can be in terms of color, intensity, etc.

We took inspiration on the algorithm and the methods seen in class to make our own. This project was made in C++ language with the OpenCV library[1] where we used some basic functions such as image reading.

II. APPROACH

A. Steps

1) The algorithm

The first step in region growing is to select a set of seed points. Seed point selection is based on some user criterion or randomly. We decided to plant our seeds in a random way and the initial region begins as the exact location of these seeds. The regions are then grown from these seed points to adjacent points depending on a region membership criterion. It could be color, pixel intensity, or grayscale texture for example, but we choose to consider their color only. A pixel joins a region when its distance to the seed is lower to a determined threshold. We take each color as a dimension. In the output image each region has a random color.

2) Options

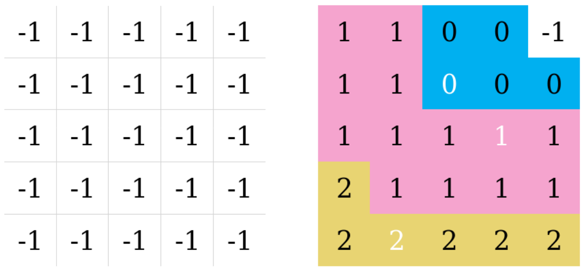
There are multiple parameters to guide the region growing algorithm. We could determine how many seeds to plant with the "-nbSeeds" option, or how much of the image must be covered with the "-tolerance" option. We could also change the similarity threshold between pixels by adding the "-threshold" option. We may want to use threads with the "-threads" option.
In addition, if we want to outline regions we can use the "-outline" option and/or if we prefer to preprocess them, we use the "-smoothing" option.

Moreover, those are just options and then, the default configuration is 100 seeds, a threshold of 5, 80% coverage tolerance, no threads, no outline and no smoothing.

B. Technical aspects

1) Linear approach

We first began to code a linear algorithm. Every seed is randomly planted then we process them in the same order they have been placed. Every seed has an ID. At the beginning, we have a matrix of -1 with the same size as the image. When a pixel is assigned to a region, we set its value to the ID of the parent seed in the matrix (Figure 1). For every seed's neighbor, we check if they haven't already been assigned to a region and if their color is similar. If so we add them to the seed's region and redo the same tests on their own neighbors. We also make sure that a common pixel to two others is not checked twice. We use boolean vectors that are the size of the image to store whether the pixels have been assigned or visited. The region is defined once no other neighbor finds a similar pixel next to them.



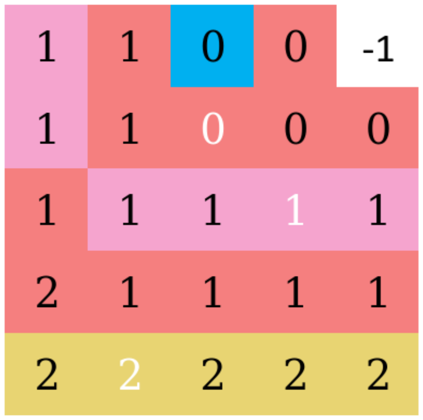
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

1	1	0	0	-1
1	1	0	0	0
1	1	1	1	1
2	1	1	1	1
2	2	2	2	2

Figure 1 : Matrix filling.

2) Coloration, outlines and tolerance

Once the matrix is filled and all the repetitions are done, we assign a random color to each seed's ID and create the final image considering that. When a pixel isn't already processed, we fill a black color into this. To outline regions we check if two neighbors have the same ID in the matrix, and if not, we set the first one to red (Figure 2). Thus the adjacent pixels which aren't similar fill the outline of the regions.



1	1	0	0	-1
1	1	0	0	0
1	1	1	1	1
2	1	1	1	1
2	2	2	2	2

Figure 2: Outline with filled matrix.

A tolerance threshold can be set which indicates the percentage of the image that must be associated in regions. If after the first iteration this threshold is not reached, we loop on the remaining pixels by planting new seeds randomly chosen on untreated pixels and reprocessing the algorithm until crossing the threshold.

3) Optimizing and Threads

The more seeds there are, the longest the program takes. Indeed, each seed has to check if its neighbors haven't already been visited then, if not, determine whether they have to join their region or not. If they join, we repeat these steps on the next neighbors. If there are too many seeds the last ones can then be useless because every pixel is already in a region. To speed the program up, we tried to use threads. Seeds check alongside their neighbors and grow a region. Matter of fact, the first version of our program took longer to execute with the threads than without. Indeed the seeds being randomly planted all over the image, threads interfere with each other. We decided to use only four threads and dedicated a part of the image to each one but the linear version of the program is still faster.

4) The region merging

We used a region adjacency graph (RAG) to merge our similar regions. It is actually a map containing the ID seed as key and an list of ID as value like this :

```
std::map<int, std::list<int>> graph_adjacency
```

An adjacency list would just be a set of IDs representing the edges of a graph. Instead of this, we could have used an adjacency matrix for the graph implementation [2]. By comparing them, we noticed that a matrix will lead to higher time complexity (Figure 4).
For small graphs, the difference will probably be minimal. But in our case, graphs can be quite large and so the adjacency list will be faster.

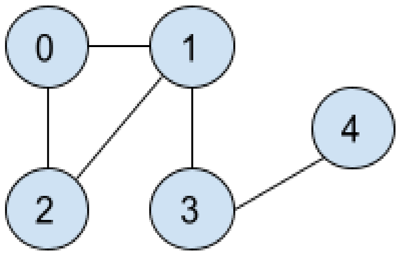
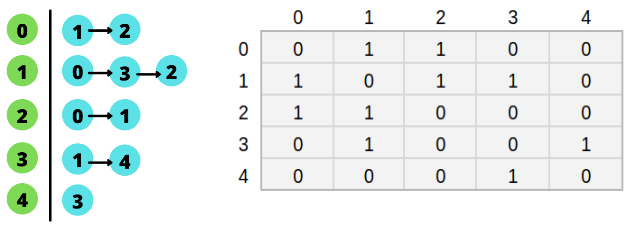


Figure 3 : Unoriented graph of regions with edges representing adjacency between regions.



	0	1	2	3	4
0	0	1	1	0	0
1	1	0	1	1	0
2	1	1	0	0	0
3	0	1	0	0	1
4	0	0	0	1	0

Figure 4 : Comparison of graph implementation[3] from Figure 3 between an adjacency list (at left) and an adjacency matrix (at right).

During the region growing function, we complete our map by adding a list of the ID of adjacent regions for each seed.

Then we perform the following step for each region : For each adjacent region of the current region, we test if they are similar. To judge the similarity of adjacent regions, we compare the color mean of their pixels found thanks to the matrix containing seed's ID. If we find similarities, we merge them, and modify the RAG by merging their list in the current list and removing the list of the adjacent region.
By doing this, the adjacent regions to the merged region are now in the list of the current region and the merged region will not be processed because it has already been merged.

III. RESULTS

A. With threads

When each thread is dedicated to one corner the program is almost twice as fast as when threads are all over the image. Therefore we used this method. Then, the merging is done on homogeneous regions with a similar average color.

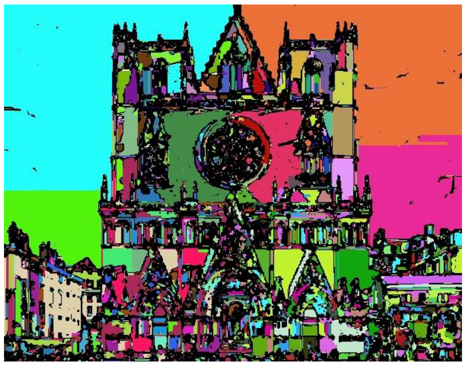


Figure 5 : The Cathedrale image segmented with a 70% tolerance, with smoothing and using threads.

B. Without threads

We found a better time execution by not using threads. Indeed it takes half time for the same configuration. We obtain this result for a cathedrale image with

C. Preprocessing

Depending on the picture complexity, adding the smoothing option to reduce noises and details allows us to get a better segmentation. For example, we actually obtain a more satisfying result on the cathedrale image with 80% tolerance and the smoothing option (Figure 5) than without the last option (Figure 6).

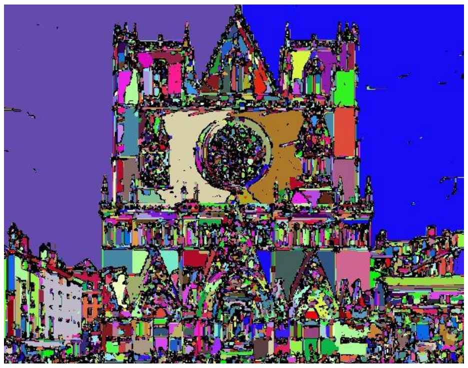


Figure 6 : The Cathedrale image segmented with a 80% tolerance and with smoothing.

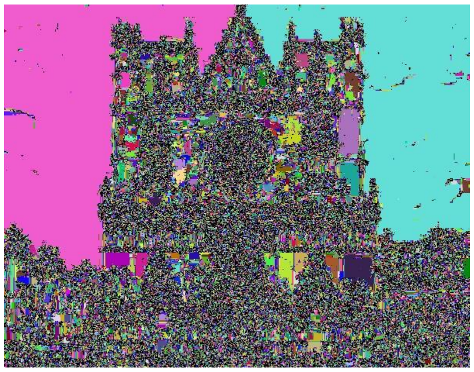


Figure 7 : The Cathedrale image segmented with a 80% tolerance and without smoothing.

When we compare both of the time results, it takes 4 times longer to not smoothing the image because it has more details to segment.

D. Repetition of the process

The repetition depends on a tolerance threshold editable in option. Both thread choices (with or without) used this threshold and we obtained consistent results. Depending on the data, the execution could take more time with a bigger tolerance threshold because the untreated pixels will be fewer.

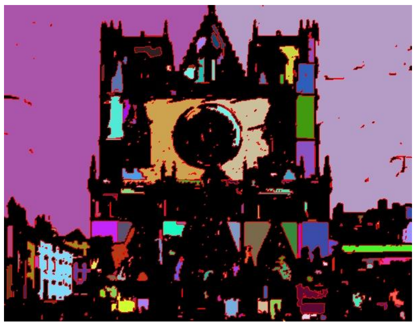


Figure 8 : The Cathedrale image segmented with a 50% tolerance, with smoothing and outline options.

Our final program works best without threads and when preprocessing complex images (smoothing them).

IV. CONCLUSION

We are quite pleased with our program. However we have a few more ideas we'd like to set out.

A. Pixels comparison

Depending on where the seeds are planted we obtain different regions. It's because when growing a region, we always compare a pixel to the seed. We could compare it to the color of its closest neighbor in the region or to the color mean of all the pixels of the region. That's what we do when merging regions. Moreover when merging regions when using threads, the regions color comparison not being the same as the pixel to pixel comparison creates not well defined regions.

B. Seeds

Here we randomly plant seeds. We could determine where they should be or not be after an edge detection or use the image bar chart and place seeds on the pixels where it peaks.

C. Tolerance

Regarding the percentage of the image not covered (tolerance) we thought about filling the remaining pixels with the same color as their closest region. The thing is, the fact that they are not already in this region means they do not have a similar color. So we thought about filling missing pixel groups with another random color. But it was not typical of a region growing algorithm so we left the black pixels as they were.

D. Outlines

To outline regions we decided to browse the region matrix only once and check only two neighbors for each pixel to have a better complexity. We could have made a better outline with a more complex algorithm or even by using an edge detection one but we chose the simplest option.

REFERENCES

[1] OpenCV online documentation, https://opencv.org/

[2] Tanmay Sakpal, "Graph Implementation | Adjacency Matrix vs Adjacency List | Full C++ Program Code", October 31, 2021, https://simplesnippets.tech/graph-implementation-adjacency-matrix-vs-adjacency-list-full-c-program-code/