Region Growing

Salmi Dorian and Guittard Mehdi
Master 1 Computer Science
University Lyon 1
Villeurbanne, France

***Abstract*—In the field of image processing, image segmentation is a common method to start analysing an image. This document presents the steps we took to design and implement an image segmentation algorithm based on region growing using the OpenCV library (in C++).**

***Index Terms*—image, segmentation, OpenCV, C++**

I. Introduction

This document presents the steps we took to create and implement an image segmentation algorithm based on region growing, and the results we got applying it to some pictures.

This lab can be divided into two main parts, execute the region growing and then merge the adjacent regions following some criteria which will be detailed later on.

The implementation was done using C++ and the OpenCV library.

II. Design and implementation

A. Model

While having a single object representing the regions of our image would be sufficient, some steps would be really inefficient. To address this we chose to have these objects.

First, a 2D table that associates a region ID to some coordinates *regions*, with -1 as the default ID (meaning the pixel in question hasn't been added to a region yet).

Secondly, a structure that properly represent a region with the following attributes :

- an ID
- a color associated
- a list of the pixels included in the region
- a list of neighboring regions
- the average value of the region (that we will explain later)

The different regions will be stored in a 1D table *list-regions*.

In addition, we will have a table containing relative coordinates of the 8 pixels surrounding a given one.

$surroundings = [1;1], [1,0], ..., [-1,0], [-1,-1]$

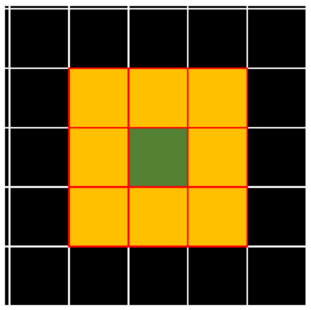


Fig. 1: Surroundings of a pixel

B. Placing the seeds

Having the seeds placed in the same place at each execution is not ideal as the algorithm might get in an edge case where the segmentation might not be done properly and there will then be no work around to correct it.

Thus we opted to generate randomly the positions of our seeds. We divide the images in ten equal section vertically, and ten horizontally, giving us a total of a hundred sections. In each of them we randomly choose one pixel to act as a seed.

That still doesn't solve all our issues though. In particular, once all the seeds have grown, there is a big chance that some pixels will not be included in a region.

To alleviate this, after the initial seeds have been grown as much as possible we add new seeds, then redo the growing, on and on until the number of pixels covered by our regions is equal to the total number of pixels in the image.

C. Region growing algorithm

Each region begins with just one seed. We use a list of points in this part (*to-visit*), which at the start contains only the seed of the region.

For each pixel we want to examine, we compare its value to the ones of the 8 pixels around it, and if the following relation is verified, integrate the surrounding pixel to the region.

In this equation, *v* is the pixel we are visiting (the seed for instance) and *s* is one of the surrounding pixels.

R_x , G_x and B_x are respectively the red, green and blue values of a pixel.

$$|\sqrt{(R_v - R_s)^2 + (G_v - G_s)^2 + (B_v - B_s)^2}| < threshold \tag{1}$$

The threshold is a value passed as a parameter to the program. The reason for it is that depending on the colors present on the image and the overall contrast, different values will produce very different regions.

The algorithm to grow a region is as follows:

```
void regionGrow(const Mat& img_source,
    vector<vector<int>>& regions,
    vector<Region>& list_regions,
    Point& seed, int& included_pixels,
    const int& threshold_growing) {


    list<Point> to_visit;
    to_visit.push_back(seed);

    while (!to_visit.empty()) {
        Point current_center = to_visit.front();
        to_visit.pop_front();

        for (int i = 0; i < 8; i++) {
            Point current_check = current_center +
                SURROUNDINGS[i];
            // Checks if the point is in the image
            if (current_check.x >= 0 &&
                current_check.x < img_source.cols
                && current_check.y >= 0 &&
                current_check.y < img_source.rows) {
                if (regions[current_check.x]
                    [current_check.y] == -1) {
                    float distance =
                        distanceBetweenTwoPoints(
                        img_source,
                        current_check,
                        current_center);
                    int region_current_id =
                    regions[current_center.x]
                    [current_center.y];
                    if (distance <
                        threshold_growing) {
                        m.lock();
                        regions[current_check.x]
                            [current_check.y] =
                            region_current_id;
                        to_visit.push_back(
                            current_check);
                        list_regions[
                            region_current_id].
                            content.push_back(
                            current_check);
                        included_pixels++;
                        m.unlock();
                    }
                } else if(regions[current_check.x]
                [current_check.y] !=
                regions[current_center.x]
                [current_center.y]) {
                    m.lock();
                    // Adds the encountered region
                    //the adjacency graph
                    //for both points
                    list_regions[regions[
                        current_check.x]
                        [current_check.y]].neighbors.
                        push_back(regions[
                        current_center.x]
                        [current_center.y]);
                    list_regions[regions
                        [current_center.x]
                    [current_center.y]].neighbors.
                        push_back(
                        regions[current_check.x][
                            current_check.y]);
                    m.unlock();
                }
            }
        }
    }
}
```

This process is threaded to improve performances.

D. Regions fusion algorithm

Now that the whole image is covered by our segmentation, some regions have to be merged. To do so we use the structures we described previously (especially the lists of adjacent regions).

To ensure that all the possible fusions are executed given the parameters, we apply the algorithm twice.

```
void regionFusion(const Mat& img_source,
    vector<Region>& list_regions,
    const int& threshold_fusion) {

    for (int i = 0; i < list_regions.size(); i++) {
       list_regions[i].calculateAverage(img_source);
    }

    for (int m = 0; m < 2; m++) {
        for (int i = 0; i < list_regions.size();
        i++) {
            while (
                !list_regions[i].neighbors.empty()){
                int id_checking =
                    list_regions[i].neighbors[0];
                if (i != id_checking &&
                    abs(list_regions[i].average -
                    list_regions[id_checking]
                    .average)< threshold_fusion) {

                    for (int j = 0;
                        j < list_regions[
                        id_checking].
                        neighbors.size();j++) {
                        if (list_regions[
                            id_checking].
                            neighbors[j]
                            != i) {
                            list_regions[i].
                                neighbors.insert(
                                list_regions[i].
                                neighbors.end(),
                                list_regions[
                                    id_checking].
                                    neighbors[j]);
                        }
                    }
                    list_regions[i].
                      content.insert(
                      list_regions[i].content.end(),
                      list_regions[list_regions[i].
                      neighbors[0]].content.begin(),
                      list_regions[list_regions[i].
                      neighbors[0]].content.end());
                    list_regions[id_checking]
                        .content.clear();
                    list_regions[id_checking]
                        .neighbors.clear();
                }
                list_regions[i].neighbors
                    .erase(list_regions[i]
                    .neighbors.begin());
                list_regions[i].
```

```
                        calculateAverage(img_source);
                }
            }
        }
}
```

III. Results and discussion

In this section, we display some examples of results we obtained with our implementation.

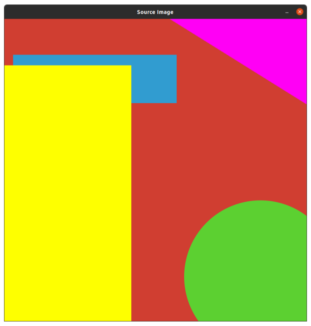
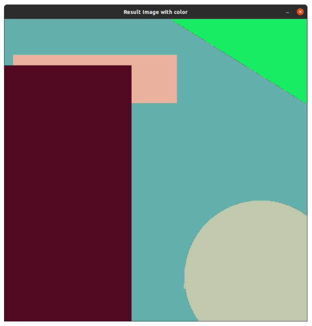
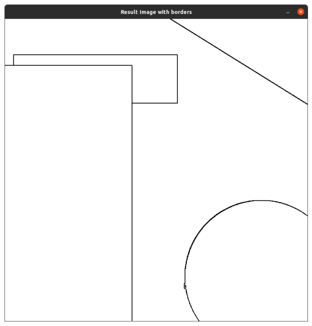


Fig. 2: Test image (Source)

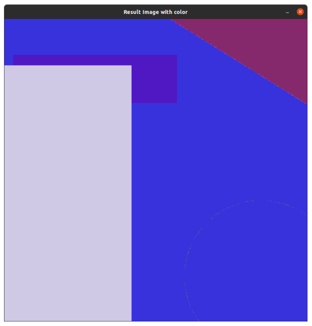


(a) result colors, thresholds: growing=15, fusion=4

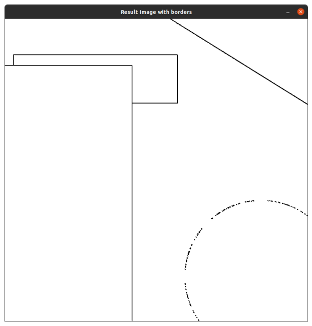


(b) result borders, thresholds: growing=15 fusion=4

Fig. 3: Results on the test image, thresholds: growing=15 fusion=4



(a) result colors, thresholds: growing=15 fusion=14)

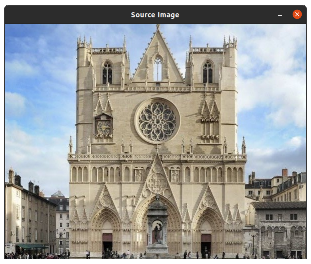


(b) result borders, thresholds: growing=15, fusion=14)

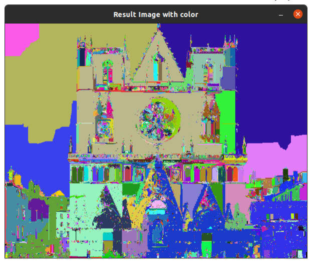
Fig. 4: Results on the test image, thresholds: growing=15, fusion=14

The figure above is a great is a great example of the impact of the threshold used for the different steps. Comparing the two result images obtained with different thresholds for the fusion, we can clearly see that the circle in the bottom right corner has disappeared, meaning some regions merged that we didn't want.

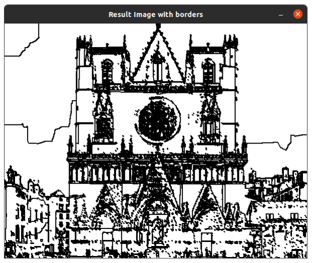
This illustrate how depending on the image's contrast, and elements that we want to detect, one threshold might not be ideal or even work at all.



(a) Source

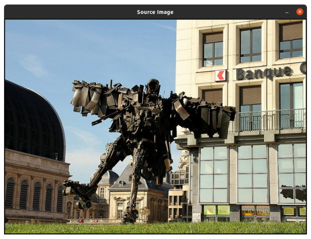


(b) Result (colors)

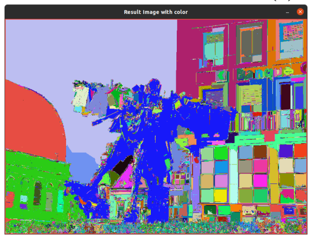


(c) Result (borders)

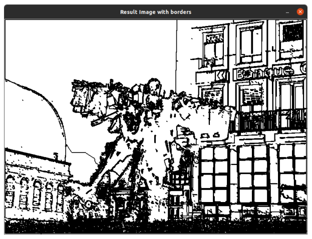
Fig. 5: Results on the "Cathedrale de Lyon" image (thresholds: growing = 18, fusion = 17)



(a) Source



(b) Result (colors)



(c) Result (borders)

Fig. 6: Results on the "Patineur de Cesar" image (thresholds: growing = 19, fusion = 25)

Here, having a merge threshold above the growing one actually gives more convincing results.

IV. Conclusion

We implemented a multi threaded segmentation algorithm and the merge of the generated regions, putting in light the influence of the merge criteria (threshold) and the variability that we have from a image to an other.

The issues we encountered were for the most part caused by the necessity of having an efficient implementation.

We also experimented with some other criteria for the merging, such as normalizing the threshold with the average value of red, green and blue of the entire image. However the results we got were not good enough in our opinion.