Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Multimedia Systems
Project Report

Image Segmentation - Region Growing algorithm

Academic Year 2019/2020

Group "Tree Friends"

Neli Mujic (ID 16067)
Chiara Mistrorigo (ID 15739)
Matteo Messmer (ID 15725)

1. Introduction

1.1. Region growing

Region growing is a method for segmentation based on regions. This approach starts by finding an initial pixel, called "seed" and examines the nearby pixels. The goal is to find a region, that is a group of connected pixels with similar properties. To determine if pixels are similar, a value called threshold is used. It is used to determine whether a pixel is a valid neighbour of the seed or not. The process iterates to apply the same approach to all the pixels of the image. The output is a segmented image containing all the regions found.

2. Java source code

2.1. First Algorithm

The aim of our project is to develop an application in Java which consists in performing a segmentation of an image using the region growing algorithm. "Region growing" is a segmentation algorithm based on a growing process of regions, which will be composed of several pixels with similar properties.
For this reason, the first step of the implementation to perform is to detect "seed" points. Since this initial step is crucial to obtain a good representation of the segmentation of the image, we decided to detect the initial set of seeds using the Euclidean distance.

```
public static double euclideanDistance(double[] pixelA, double[] pixelB) {
        return Math.sqrt(Math.pow(pixelA[0]- pixelB[0], 2) + Math.pow(pixelA[1]-pixelB[1], 2) + Math.pow(pixelA[2]-pixelB[2], 2));
}
```

The Euclidean distance method calculates the maximum distances of colors from the pixel B, which is initially set on {0, 0, 0} to represent the black color, and pixel A. Pixel A is the pixel that is being analyzed and to which we have to set a value. The value corresponds to the distance between the pixel and the color black. The darker the pixel is, the lower the value will be.

The initial seed is the pixel that gets the highest value of Euclidean distance, which is the lighter pixel. Once it is found, the region growing algorithm starts to detect the nearby pixels and checks if the properties of the adjacent pixels are similar to the initial ones. To move on the image, we defined several methods: *getRightPixel()*, *getLeftPixel()*, *getUpPixel()*, *getDownPixel()* and so on, to which we pass the row and column of the initial pixel to get the value of the nearby pixel.

To define the value of the homogeneity threshold, we decided to make a slider, so the user can see the algorithm performing on the image with different thresholds. If the distance between two pixels is smaller than the value of the threshold, then the pixels are homogenous, otherwise they will not be homogenous.

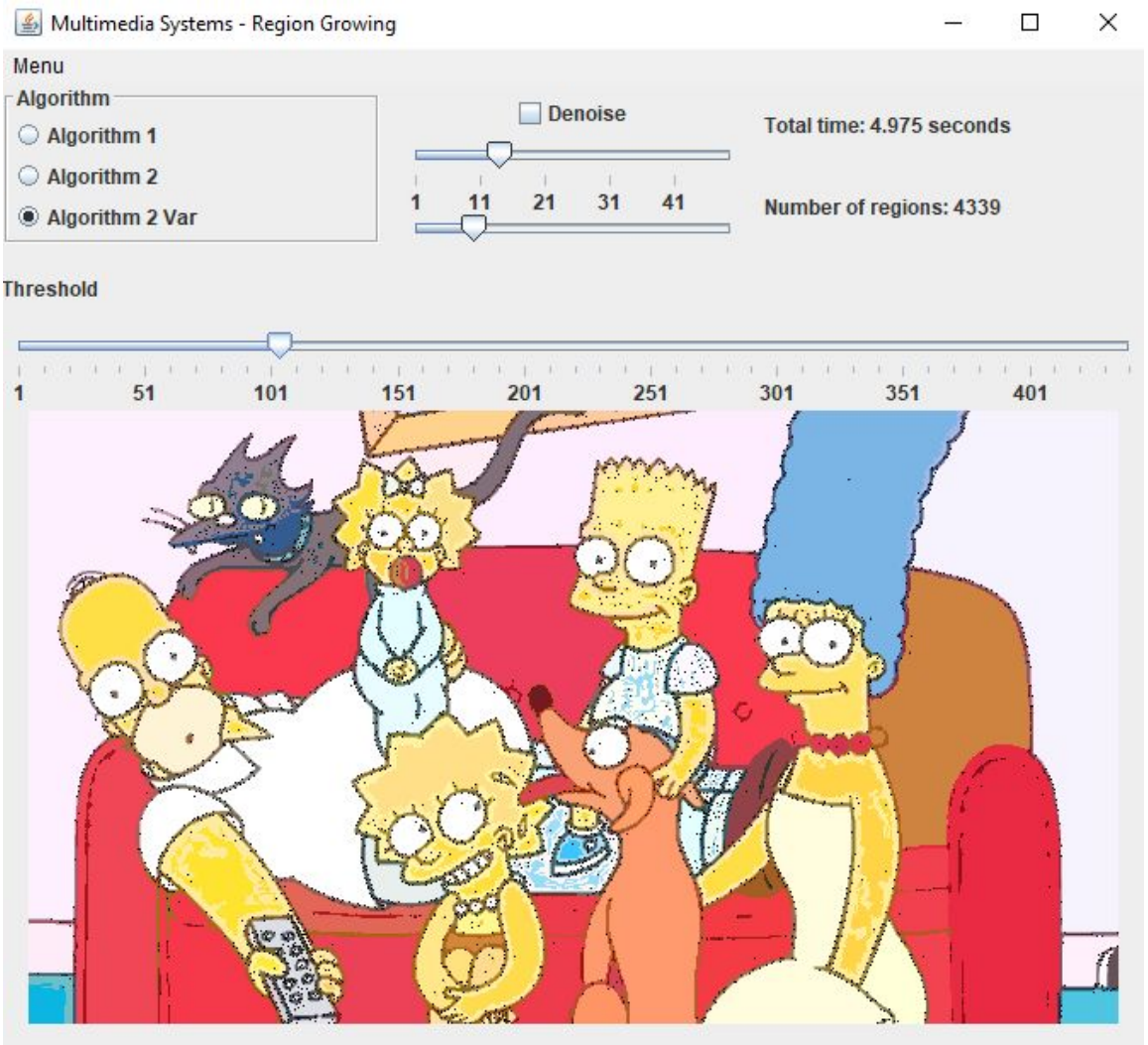
To keep track of the regions, we created an array called "stack", which represents the groups of seeds.

Once we get the nearby pixels, we add them to the "neighbors" array and we check if the value of the neighbor pixel is smaller than the threshold value: if true, the neighbor pixel

will be added to the stack array, which composes the region containing pixels with similar properties.

```
// add pixel in region to stack
for (int i = 0; i < neighbours.size(); i++) {
        if (neighbours.get(i) != null) {
                if (!visited[neighbours.get(i).getRow()][neighbours.get(i).getCol()]) {
                        if (neighbours.get(i).getDistance(seed.getValue()) <= thresholdValue) {
                                stack.add(neighbours.get(i));
                                visited[neighbours.get(i).getRow()][neighbours.get(i).getCol()] = true;
                                segmentedImage.put(neighbours.get(i).getRow(), neighbours.get(i).getCol(),
                                                seed.getValue());
                        }
                }
        }
}
stack.remove(0);
```

After that, since we know that this pixel composes a region, we set it to “visited”, which is an array of boolean values.
In this way, the number of visited pixels will gradually grow till we obtain the final result, that is the segmented image.



2.2. Second Algorithm

Since the code takes very long to execute, we decided to optimize it. The main problem was the complexity of the first part of the code, which had $\theta(n^3)$ complexity. This part is meant to find the initial seed by iterating through all the pixels of the image.

The optimization that we decided to perform uses lists. All the pixels of the image are put inside a list, which is ordered based on the Euclidean value of each pixel. In this way, the darker pixels will be at the beginning of the list, followed by the light pixels.

The first seed in this case will not be the lightest pixel, but the darkest one.

```
// create arraylist for all the pixels of the image
ArrayList<Pixel> pixels = new ArrayList<Pixel>();

// put pixels in the arraylist
for (int row = 0; row < src.height(); row++) {
        for (int col = 0; col < src.width(); col++) {
                double[] value = src.get(row, col);
                pixels.add(new Pixel(row, col, value, Pixel.euclideanDistance(value, new double[] { 0, 0, 0 })));
        }
}

// sort the pixels by "distance from black"
Collections.sort(pixels, new Comparator<Pixel>() {
        @Override
        public int compare(Pixel p1, Pixel p2) {
                return ((Double) p1.getDistanceFromBlack()).compareTo(p2.getDistanceFromBlack());
        }
});
```

Using the lists, the first part of the code becomes more efficient. Then the algorithm continues like the first algorithm, by finding the neighbour pixels and constructing the regions.

2.3. One step ahead

After implementing a second optimized version of the algorithm, we decided to write a modified version of it that starts from the lightest pixels. In this case, we maintain the structure of the second algorithm, but we modify the order in which the pixels are arranged in the initial list.

Moreover, after uploading different photos to our application, we noticed that there was some noise in some photos. For this reason, we decided to add a “denoise” button, that can be selected by the user. The denoise button calls a method of the OpenCV library:

```
Photo.fastNLMeansDenoisingColored(src, dest, h, hColor);
```

After concluding the implementation, we run tests on the application using several images, collecting some results.

In the first algorithm, we noticed that, with a particular image, the lower the threshold, the slower the algorithm to calculate the region growing is. So, the speed of the algorithm was depending on the value of the threshold. We noticed that if the threshold was set higher than “200”, the running time of the first algorithm was much faster than the second one.

However, for other test images, it was not the case. So we concluded that the performing time of the algorithm is not equal in all the situations, where the value of the threshold changes the performing time also changes. There are several factors that affect the performing time of the algorithm: the bigger the image, the longer the algorithm takes to perform. As well as the the colors inside the image, if the image has many colors inside, the algorithm will take a long time to perform.

3. Conclusions

Since the first algorithm we developed was inefficient, we decided to implement a second algorithm with a higher level of performance. In this way we saw the difference between the two algorithms, performing the segmentation of the image. We concluded that there are different approaches to perform the region growing algorithm, depending also on the picture that needs to be analyzed.

4. References

Project repository: https://github.com/matteomessmer/MultimediaSystemProject