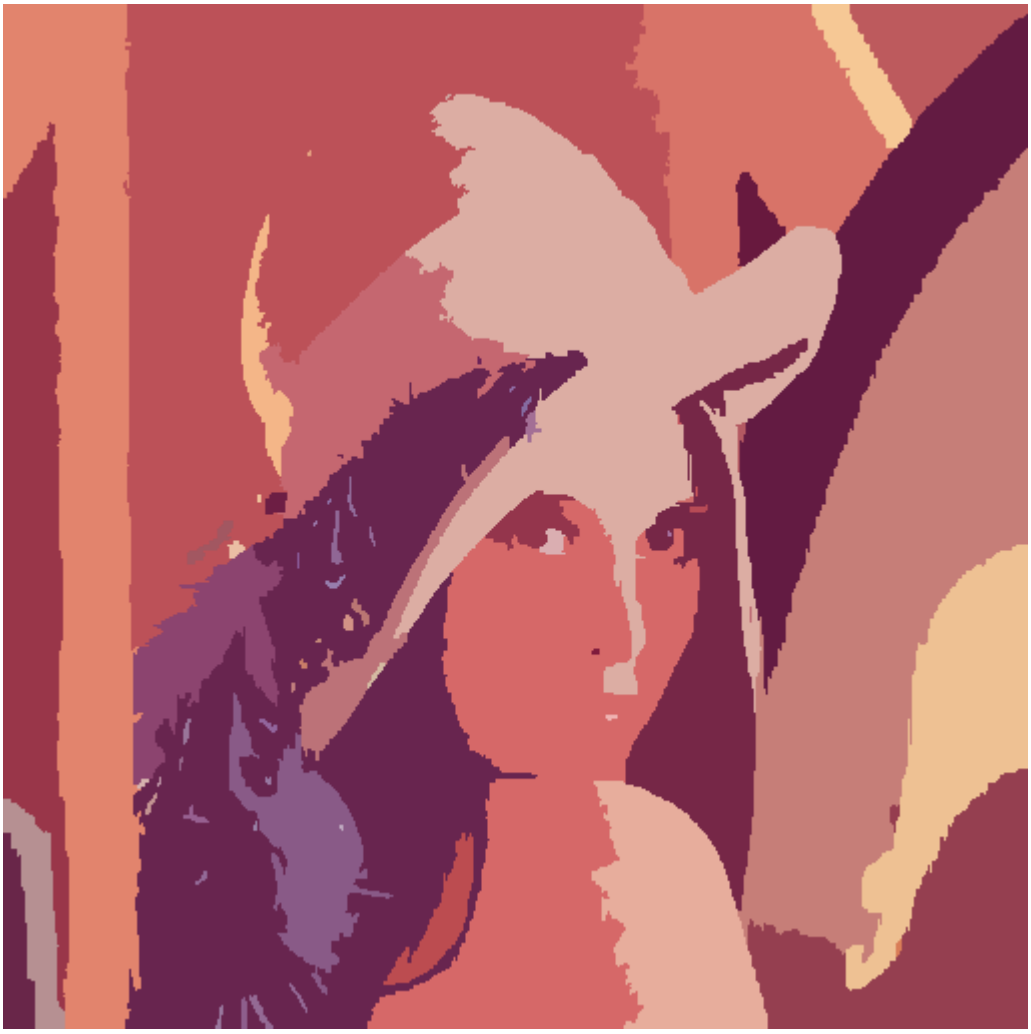
Statistical Region Merging

An implementation of the segmentation algorithm as a web-app



University of Catania – Master Degree in Data Science

Academic year 2020/2021

Project report for Multimedia

Marco Cavalli – 1000024189
Teachers: F. Stanco – D. Allegra

Table of contents

1 Introduction

Statistical Region Merging (SRM) is an algorithm used for image segmentation. The main point of the algorithm is to recognize different regions inside a picture. Regions are created grouping together points of the picture itself using a specific merging criteria.

The algorithm has been published officially in November 2004 by Richard Nock and Frank Nielsen [1] and it is possible to retrieve executable software implementing the algorithm written in either C++, Java or Python.

The aim of this project is to provide an implementation of the algorithm as a web-app. Users will be able to upload images and then run the algorithm and retrieve the segmented image. The web-app will provide input controls to allow the customization of parameters to provide in input to the algorithm, allowing users to try different configurations.

We will explain first the characteristics of the proposed solution, talking about the merging predicate, the technologies used to implement the algorithm and focusing on the issues experienced while proceeding.

Then it will be shown how to run the software and use it.

Finally we will offer a comparison with the original work made by Richard Nock and Frank Nielsen, comparing the output of their software with the output of this software with same configurations.

2 Proposed Solution

We mostly wanted to reuse what it was stated in the scientific paper [2] since it is not the goal of this project to improve their solution. Still we have added few configurations and utilities to allow users to “play around” with the images they upload.

2.1 Merging predicate

The predicate adopted for this implementation is the same as the one used for SRMB implementation of Richard Nock and Frank Nielsen. Given:

1. a a point in the picture (a pixel) with one dimension if we consider a gray-scale image or with three dimensions if we consider a colored image
2. $\overline{R_a}$ as the average color value for a specific region (which is a scalar for gray-scale images, or a vector with three values for colored images)
3. the function $b(R)=g^2 \div (2Q r_R) * min(g, r_R) * \log(r_R + 1 + \partial)$ where we define the following notations:
   - g is the number of levels (such as 256 for 8-bit images)
   - Q is the number of random variables used to replace each color channel. Generally high values for Q leads to regions being smaller (Q is by default 32)
   - r_l is a set of regions having l pixels

- ∂ is a constant which is defined as $\partial = \log(6I^2)$ having I as the number of points of the image

The merging predicate is:

$$P(R,R') = \begin{matrix} true\ iff\ \forall a \in \{r,g,b\} \Rightarrow \left|\widehat{R_a} - \widehat{R_a'}\right| \leqslant b(R) + b(R') \\ false\ otherwise \end{matrix}$$

2.2 Technologies

The software is a web-app and it is supposed to offer a certain degree of interactivity to the final users. So we decided to implement the software using the following technologies:

1. We have chosen NodeJS (version 12.18.2) for the back-end
2. We have not chosen any specific framework for the front-end, but we used several libraries to setup the graphics such as Bootstrap 4, jQuery and font-awesome

This choices allowed us to implement many interactive operation an user is allowed to do when using the web-app.

The Statistical Region Merging algorithm has been coded in Python. The reason behind this choice is that we didn't want to give up on Python's modules which allow to perform image processing operations (such as to apply morphology operations or to edit the pixels of an image) easily. Users won't interact directly with the Python code, but they will triggers Javascript functions which will execute the Python code.

We used Python 3.6 and we added the following modules:

1. numpy, to simplify array handling
2. opencv-python, to manage any image-processing operation

2.3 Issues

In this section we want to highlight the major issues we experienced while moving forward to the end of this project. In the next section we will look at the proposed solutions to avoid these problems.

2.3.1 Handling occlusions and differences in brightness (Issue 1)

Occlusions can lead to wrong decisions when you merge two regions. Why so? The problem is that not all the occlusions are real objects (they can be noise for example), or they could be part of the same object, but with a different brightness. So, to have better results, we had to add a logic to handle occlusions.

2.3.2 Tiff images representations (Issue 2)

We want to allow users to use any static image (no GIFs or MNGs for example). Generally you have not a lot of problems to display images on a web-page since we mostly use images that can be interlaced withing a web-page. The problem is that the original software [2] worked with TIFF images only. We didn't want to prevent users to use their TIFF images too and here is the catch: TIFF images can't be shown in a web-page.

2.4 Proposed implementations

In this section we will give our solutions to the issues enlisted in section 2.3.

2.4.1 Solution for Issue 1

To handle occlusions/difference in brightness we thoughts of two mechanisms:

1. Set minimum size values for regions
2. Set maximum numbers of regions to display

Both the solutions follows two different ideas and can be configured by the user, but let's process them one by one.

The first solution is all about giving a minimum number of pixels a potential region must have to be considered a region. If at the end of the algorithm any region found doesn't have the minimum number of pixels defined for that execution,they will be merged to the following region.

The second solution can help tuning the precision of the algorithm. The original implementation in C++ quietly stated - or, to use other words, you can see it by reading the C++ code - that the maximum number of regions is 25, so we can reuse this concept to group smaller regions together with bigger regions. The way we do this is to simply order regions in descending order by the number of pixels each region contains. Then we just consider only a fixed number n of regions (by default n is 15, but users can tune it up and down before executing the software). Then we check if each point of the image is inside one of the biggest regions or not. If it isn't, then we will merge this point to the region next to it. This is a linear complexity operation which doesn't slow down the execution itself (considering that SRM is by itself more complex).

2.4.2 Solution for Issue 2

The software allows the upload of any type of image, except GIFs and (in general) any type of images with animations. Clearly, we can't display TIFF images (we have not reinvented HTML yet), but we found a simple workaround. If the user uploads an image which can't be displayed (such as a TIFF image) the SRM algorithm will still use this version to run, but, meanwhile, the image will be converted in PNG and it will be shown on the box. This way the user can have both the original version and the segmented versions shown as if they would upload a PNG/JPEG image. It is not possible to display the TIFF image before the upload of the image since the back-end handles the conversion of the image and it have to wait for the "upload" event to be triggered to do so.

3 Tutorial

In this section we will go through the requirements to install and to run the software and how users are supposed to interact with it.

3.1.1 How to run

There are two ways to run the software which are:

1. npm scripts

2. docker containers

3.1.1.1 npm

First, you need to install node (we used v12.18.2). Then you must run the commands:

git clone https://github.com/Mirkesx/web_statistical_region_merging
cd web_statistical_region_merging
npm install

Now to run the back-end you can run this command:

npm run start

At this point you can reach the web-app at http://localhost:8080.

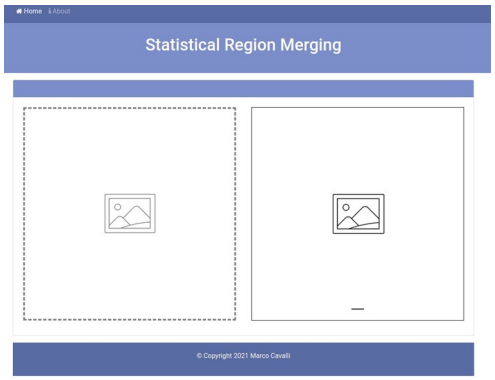
3.1.1.2 Docker

First you must install docker. Then just run:

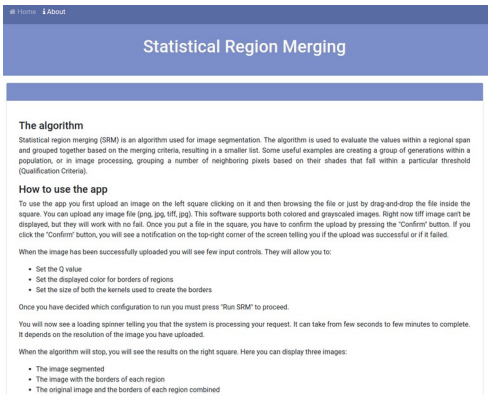
docker run -d -p 8080 mirkesx/web_srm
docker ps

The last command will display the running containers. You have to check which port has been assigned for the container you've just run reading the column "PORTS". Then you can reach the app at http://localhost:<docker-ps-port>.

3.1.2 How to use it - Tutorial

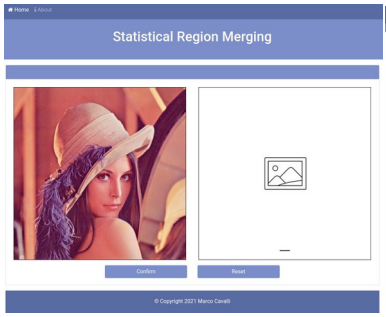


Picture 3.1

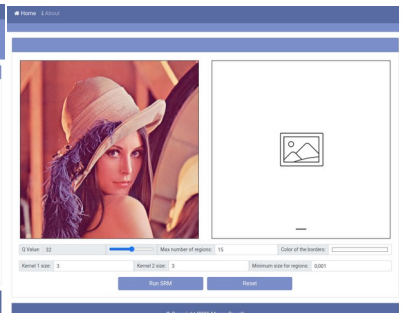


Picture 3.2

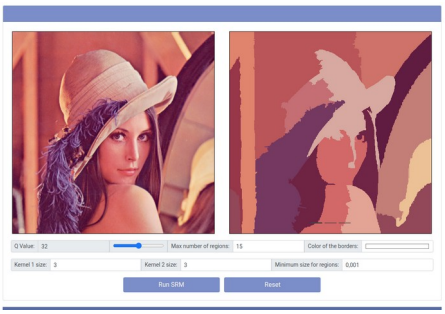
The homepage is simple: you can just see two boxes and two menu links. Clicking on "Home" will display the view of picture 3.1. Clicking on "About" will display the view of picture 3.2. This view contains few words about the software and a brief tutorial.



Picture 3.3



Picture 3.4



Picture 3.5

To upload an image just drag-and-drop the image on the left box or click on it and pick an image file (picture 3.3). When you have decided which image to pick just press “Confirm”.

You will now see many controls (picture 3.4). They will allow you to customize the execution of the Statistical Region Merging algorithm leading to different outputs. Once you are confident with your decision click “Run SRM”.

You will receive a notification when the execution has completed telling you if it was successful or not. If it was, you will see a set of three pictures in the right box. You can browse them by clicking the arrows keys on your keyboard (left and right keys) or by clicking on the small indicators on the bottom of the box. You will be able to see three images:

1. The image with the average color for each region
2. The image with borders of each region
3. The original image with the borders of each region highlighted

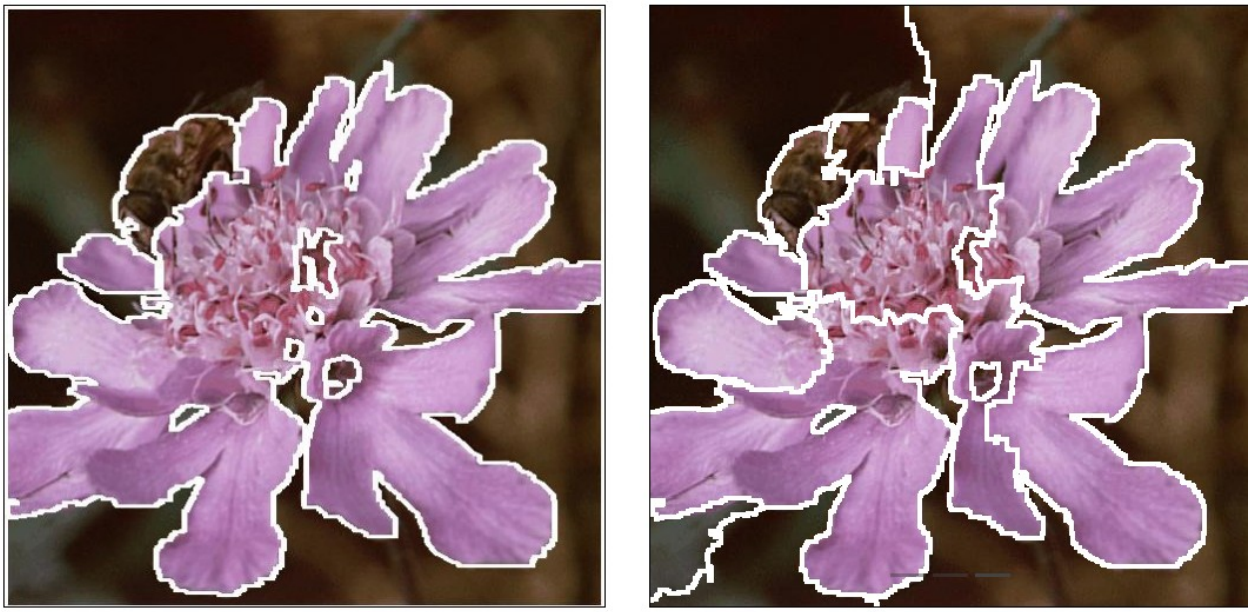
Clicking on the “Reset” buttons will reset the boxes and will bring you to a situation similar to picture 3.1.

4 Comparison with cvpr04 implementation

We proceeded comparing the outputs of our software with the output shared in the CVPR04 web-page [3]. You will always see on the left the output for CVPR04 SRMB software and on the right the output of our software.

We tried as we could to keep the comparison consistent giving the same parameters. If there is any difference between the two pictures, that’s because of the logic behind occlusions handling and edge pairs sorting. The parameters we gave to our software are:

1. Q = 32
2. Max number of regions = 15
3. White borders
4. Kernel 1 = 5
5. Kernel 2 = 5
6. Minimum size for regions = 0.001 * width * length



Picture 4.1- Left CVPR04 output, Right this software's output

The first image we compared was the flower with the insect (picture 4.1). Our algorithm seems to recognize more regions, especially in the background. It fails though to recognize the back of the insect as a part of the region containing the body of the insect.

Picture 4.2 - Left CVPR04 output, Right this software's output

The second image the tower of the castle (picture 4.2). Here our algorithm fails recognizing many windows (this is because we limited the software to recognize only 15 regions), but it performs fairly well with the plants.

Picture 4.3 - Left CVPR04 output, Right this software's output

The last comparison is with the image of the road with buildings on the background (picture 4.3). Here our algorithm seems to perform a little better since it recognizes more details and it doesn't fail too much (except for the plant on the top). The fails are clearly a result of the occlusion handling logic we introduced.

5 Conclusions

With this project, we've shown how Statistical Region Merging performs. It is clear that this is not anymore a viable way to segment images both for the problems about optimizations and handling occlusions. It is not a good solution even if there are too many subjects in the picture. Anyway it is still a fascinating algorithm to study when someone approaches segmentation for the very first time.

Probably the biggest point about this algorithm is the predicate: if you don't choose it wisely, the algorithm will drastically under-perform and it will fail to recognize the main regions inside your pictures.

It can't obviously compete with semantic segmentation and segmentation algorithms which use machine learning, but it's still very amusing how this algorithm can recognize most of the regions just by "watching" at the value of each pixel.

What could be done to improve this software? Actually most of the improvements we can think of are about the back-end and the front-end. For instance, it would be nice to have a web-socket to prevent users to accidentally overwrite images of other users and to correctly delete the pictures uploaded by users who disconnected. Another big improvement could be to keep showing all the pictures created by previous executions of the SRM algorithm so the users can browse them and see the differences. Speaking about the implementation of the SRM algorithm itself, the only very important improvement would be to switch from Python to C/C++ to speed up the execution, maybe still keeping Python to perform easily the morphology operations used to retrieve the borders and/or to save the files once the C/C++ software has completed the execution of SRM.

6 References

[1] http://www1.univ-ag.fr/~rnock/Articles/Drafts/tpami04-nn.pdf

[2] http://www1.univ-ag.fr/~rnock/Articles/Drafts/cvpr04-nn.pdf.gz

[3] http://www1.univ-ag.fr/~rnock/Articles/CVPR04/