General Program for Region Growing Segmentation

Como Adrien and Demeersseman Daniel
Department of Computer Science
University Claude Bernard Lyon 1
Villeurbanne, 69100, France
{adrien.como & daniel.demeersseman}@etu.univ-lyon1.fr

Abstract : Region Growing is one of the main algorithms used to segment objects in an image. Image segmentation is an important first task of any image analysis process. This paper presents a seeded region growing and merging algorithm that was created to segment color images. The approach starts with a set of seed pixels and from these, grows regions by appending to each seed pixel those neighboring pixels that satisfy a certain predicate. Neighboring regions of similar value are merged. together. Fast region growing is of increasing interest as it is more and more used in medicine or self driving cars.

Key words : Region growing, merging, segmentation, openCV, fusion.

I. Technical Introduction

This project uses the openCV library for image pixel access, image opening and writing. It also uses state of the art STL template libraries in C++. A makefile is provided for simple software execution and to obtain quick results.

A. Aim of this project

This project has always kept in mind that large images take a lot of time to process. We have therefore always kept the code as simple and clean as possible to maximize the speed at which the segmentation is done. This allows us to provide a low cost, immediate results, segmentation solution.

One of the by-products of this implementation is that our code is very light (13.0 Kilobytes) and is at most 250 lines long for the largest .cpp file. The main.cpp being very short too, and the .h file as small as humanly possible. It is therefore easy to read and understand the code if ever maintenance or adaptation is needed.

B. Organization

Our software is divided in 3 separate sections. A main.cpp that recuperates all necessary parameters. And the Image.cpp file in 2 parts. One where image segmentation is done. And another where region merging and drawing is done.

All the code is stored in the src/ folder. The images in the img/ folder. Our software is cross platform with ARM based boards **[1]**. Our code is structured with a main.cpp that controls the execution of the Image.cpp/.h class on which the segmentation is done.

C. Recommendations

It is recommended to use the provided blurring functions. The region growing methods are very sensitive to the threshold values. The smaller the value the better. High values have the habit of growing regions larger than they should.

In case not enough parameters are provided when using the command line, default parameters will take over. (default image, threshold, output).

This algorithm is strong enough for any type of image as long as the user has a basic knowledge of what the image looks like so that he can tune the parameters of the threshold.

D. How to run

```
make clean && make && clear && ./bin/main a b c d e f g i j
```

a : input path to image (img/…) (works with png, jpg, jpeg)
b : output path (img/…) (will write new image in .png)
c : 0 < x < 20 : Gaussian blur kernel size
d : number of seeds (1 < x < 4,294,967,295)
e : options (1 : average color of region, 2 : random color)
f : threshold range (0-255)
g : second threshold range(0-255)
h : fusion threshold between region range(0-255)
i : min for contour (0<x)
j : max for contour (min<x)

It is recommended that threshold2 > threshold when many seeds are used to allow small regions to grow to a big enough size.

II. Description of the algorithm

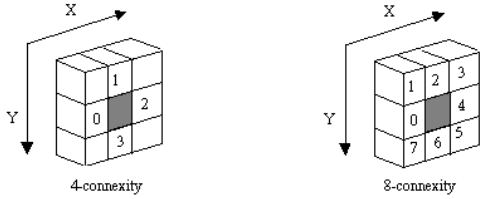
A. *Basic knowledge*

An image if made of pixels. Pixels each having 3 values (r,g,b). Apart from border pixels each pixel has at least 8 neighbors. Usually a single region is mostly composed of pixels of the same color. An algorithm analyzing each pixel could therefore define regions of the same color. We could therefore define each region by applying different colors to each of them.

B. *Region growing*

At first a 2d array of the width+2, height+2 of the image is created. Borders are set to 1 and all pixels to 0.

This approach seeds the image at random. Each pixel has a defined color. Then each pixel grows (in an 4 or 8 connectivity manner) and takes in its region each pixel that is close enough to the original color. The region therefore grows until it is surrounded by pixels of different colors.



In our implementation we used a variant of the Manhattan distance function to differentiate between the color of two pixels.

```
uif8 t = threshold;
if(std::abs(r - r) < t && std::abs(g - g) < t && std::abs(b - b) <  t)
    return true;
else
    return false;
```

Each pixel that corresponds to our criteria is added to a std::deque. The Y coordinate is placed first followed by the X coordinate. This allows us to have only one std::deque and iterate over it at high speed to speed up the process. We iterate until the std::deque is empty which indicates that the region has grown to its maximum. We therefore now take on another seed and start the process again. Each pixel taken by a region recups the index of the region and replaces the 0 value in the 2d array to its index. In this way, can we remember the span of each region.

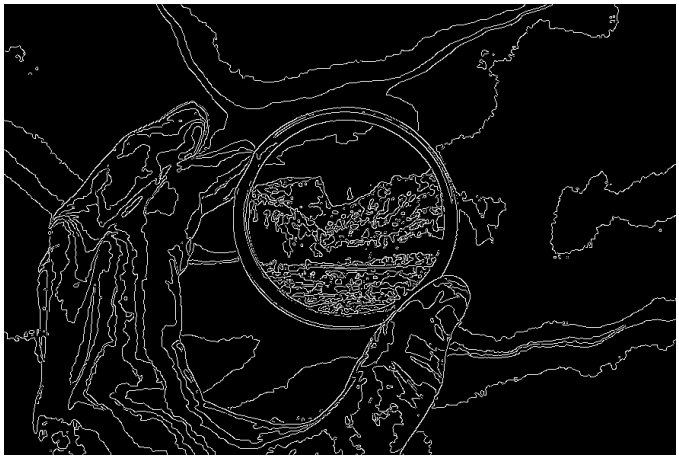
In case the number of seeds was too low or the threshold not high enough to accept all pixels in one region our algorithm is quite forgiving. It passes over all pixels which are still at 0 and start the region growing sequence from this coordinate. Eventually we can be confident that all pixels have been seen at least once and added to a region. This is done in the 'growsShadows' procedure.

Each region has its own color. All is stored in an std::deque. At the end of the growing sequence it is possible that regions have a very close color. We can merge these regions in the function 'fusionSystem' which compares the average color of the 2 regions and chooses whether or not to apply the same color to both regions. This operation is particularly quick as we only change the value of the color stored in the std::deque and do not write over all the index of regions as some algorithms do to make one big region.

III. Results

It is possible to generate an image containing all the different regions (called output_1.png) and one with all the borders of the regions (called output_2.png). Everything works just fine. Threshold will have to be modified depending on which image is segmented.

This image represents the average color of the regions that have been found by the execution of the algorithm. Precise border can be seen below.



Here is a picture of the different regions of the image.

IV. Speed and Power

A. *Makefile optimization*

Several optimizations are presented in the image segmentation project. In the makefile we can see the use of "-Ofast" **[2]** which astoundingly speeds up the process by applying many compiler optimizations. It is important to know that these optimizations are felt more strongly when used on slow computers or for example (NVIDIA) Jetson Nano boards. (Which we did just for fun and it runs just fine).

B. *Variable type optimization*

To accelerate even more the computer in the segmentation task we have used 'uint' because this allows the compiler to optimize the calculations as it knows the values can only be positive. We have also added on top of that the 'uint_fast_8' or 'uint_fast_16' as this forces the computer to find and use the fastest 'uint' with at least 8 or 16 bits. **[3]**

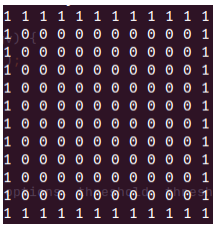
It is also important to only declare variables of appropriate size. No need for a uint_fast16_t variable for values between 0 and 255. A uint_fast8_t would have sufficed.

C. *Large array initialization optimization*

Array flag initialization was done with std::memset() which is significantly faster than traditional methods. **[4]**

D. *Out of bound testing optimization*

A significant portion of the code when executing the segmentation is checking if each pixel is already in a region. And if not, test wherever it belongs to the same region if it fits the certain criterias. Each pixel has also to be tested to not exceed the width and height of the image. Therefore for each pixel, 2 operations are needed. Which for a large image represents literally millions * 2 tests. We have thought '**out of the box**' and came up with an ingenious idea. We 'captured' the image inside a 'barrier' of '1' like below. (Image is therefore 2 pixels larger and wider).



When we now test wherever a pixel belongs to a region if it fits the color criterias, we only test if it belongs to another region. If it doesn't we add it to our region. We don't need to test the position of the pixel as we will never exceed the boundaries, as the border 'belongs' to a different region to all other pixels. In one test we now combine both tests of 'out of bound test' and 'already belongs to a region test'.

Classical example without '1' border:

```
if(x > 0 && x < img.width && y > 0 && y < img.height){
        if(flag[x][y] == 0)
            testColor();
        else
            addToSetIfNotYetSeen();
}
```

Our solution with '1' border:

```
if(flag[x][y] == 0)  //Never seen pixel and is not outside of image
    testColor();
else
    addToSetIfNotYetSeen();
```

E. *Fast random number generation*

Each seed is chosen at random inside the image. The Mersenne Twister random generator (std::mt19937 generator) is rather slow. **[5]** We preferred using a fast random generator instead. Which further reduces the time spent during the initialization of the algorithm. This also helps when drawing random colors on the image.

F. *Region proximity testing*

To test wherever 2 regions are of the same color it is important to know which region is in close proximity to the current region. It would be very time consuming to compare all colors of all regions together. We therefore used the 'std::set' container which only adds a number in its 'set' if no similar variables already exist**[6]**. (The variable being the 'index' of the region which is close to the current region). As we already test during region growing if a pixel belongs to a region we just add an 'else' that is executed when a pixel already belongs to a region and we add it in our std::set if we hadn't already 'seen' that region. This is the fastest possible way as we don't add any 'if' conditions. We use already

existing conditions for the region growing algorithm.

```
Region 2 in close proximity with : 5, 6 , 9, 50, 250
Region 4 in close proximity with : 3, 8 , 19, 20, 56
Region 5 in close proximity with : 7, 76 , 79, 750
Region 6 in close proximity with : 15, 16 , 19, 46, 80
```

```
if(flag[x][y] == 0)  // Pixel has never been seen
    testColor();
else
    addToSetIfNotYetSeen(); // Add if region not already inserted
```

G. *std::deque vs std::queue*

Our algorithm tests each neighboring pixel (4 or 8 depending on the connection) and adds it to a std::deque for later processing (to test its own neighbors in the future). After some research and experimentation we have found that a std::deque is somewhat 20% quicker than a std::queue. It also brings the advantage of using front and back insertion and deletion which is very useful in our case. **[7]**

H. *Function and procedure calls*

Each function or procedure call bears a certain cost depending on how many parameters are given. That's why we have limited the amount of calls to the minimum and have put the c(color Index), rI (region Index), r (red), g(green), b(blue) in Image.h private members as that is more efficient than declaring new ones for each seed as all 5 of them are needed at each growing iteration.

I. *cv::Mat quickest access method*

Pointer method was used to access pixel colors as this is described as the quickest way compared to the classical cv::mat.at<> method. In large images containing millions of points this is a highly recommended practice. **[8]**

J. *Performance numbers*

An image of 600*450 pixels can be segmented in around 80 ms(milliseconds).

Much larger images (1200*700) can be segmented on average in 240 ms.

Images of 1000*1000 containing a large amount of different regions take at most 320 ms. It is good to see that the number of regions does not affect the performance of our solution.

We can conclude that this algorithm is capable of sampling ~3.350.600 pixels/seconds.

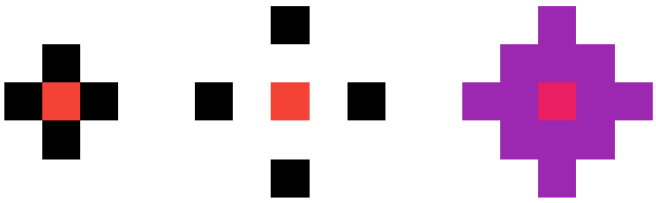
All tests are done on 16gb RAM and a Ryzen 5600x (AMD).

V. Conclusion

One of the greatest slowdown in the algorithm is that we access the pixel values in a non-linear manner. This is unfortunately obliged as we grow the algorithm in a 4 or 8 connectivity manner. I tried implementing a 2 connectivity method. A seed would acquire a pointer for each line. It would grow from left to right and only move up and down when needed too. This would remove the fact that a 4 connectivity method sometimes has to access a coordinate at the far left and suddenly at the bottom right. **[9]**

As of now, we test each neighboring pixel for similarity. It would be interesting to test the neighbors of our neighbors and grow the region in this way. In some cases like images with large regions this could significantly speed up the process.

In the 4 connectivity normal method, we test the black pixels and add them or not to the region if they fit the criterias. This can be seen in the left 'star'. In the new method we just described we would test the pixels two 2 pixels away from the center like in the middle 'star'. We would therefore add to the region 12 new pixels (violet squares) for only 4 tests if the 4 extreme pixels correctly fit the color criterias for our region growing. The violet pixels are added to the region when the 4 black pixels fit the criterias for the region.



This would significantly speed up the process as we do much fewer tests. But accuracy could suffer a little bit.

To go further and better our results we could implement a function that fuses small regions with bigger adjacent regions. This would minimize the noise in certain images. We can currently work around that limitation by using the threshold more accurately.

VI. Acknowledgment

We would like to thank the teacher for giving us tips on how to better our algorithm during the classes.

VII. References

[1] https://qengineering.eu/install-opencv-4.5-on-jetson-nano.html
[2] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
[3] https://stackoverflow.com/questions/8500677/what-is-uint-fast32-t-and-why-should-it-be-used-instead-of-the-regular-int-and-u
[4] https://stackoverflow.com/questions/7367677/is-memset-more-efficient-than-for-loop-in-c
[5] https://stackoverflow.com/questions/1640258/need-a-fast-random-generator-for-c
[6] https://www.cplusplus.com/reference/set/set/
[7] https://stackoverflow.com/questions/59542801/why-is-deque-faster-than-queue
[8] https://docs.opencv.org/2.4/doc/tutorials/core/how_to_scan_images/how_to_scan_images.html#howtoscanimagesopencv
[9] https://stackoverflow.com/questions/997212/fastest-way-to-loop-through