Image Segmentation

Marrel Pierre-Emmanuelle p2006524
ABIDA Youssef p2024398
Claude Bernard Lyon 1, 69100 Villeurbanne
https://forge.univ-lyon1.fr/p2024398/m1-analyse-image

1. Introduction

1.1. Overview of the Project

The current project is dedicated to implementing a segmentation method known as "Region Growing." Building upon the foundational tools developed in TP0 for basic image manipulation, this second phase aims to enhance and conclude the image processing workflow by introducing an advanced segmentation technique.Incorporating the principles established in TP0, where fundamental image manipulation techniques were devised, this project delves into the intricacies of the Region Growing method. The primary objective is to achieve a comprehensive and refined image processing workflow, offering a sophisticated approach to segmentation.

1.2. Image Segmentation Workflow

The central focus of this project revolves around the implementation of the Region Growing segmentation method, placing specific emphasis on a variant that concurrently utilizes multiple seeds. This approach facilitates parallel processing, enabling the simultaneous treatment of different seeds. The achievement of parallelism is realized through iterative processes applied to each region, where pixels are grown within the outline vector (i.e., the adjacent pixels).

Critical steps in the execution of this methodology encompass the strategic placement of germs, initiating region growth around these germs to form distinct regions, and subsequently, the merging of adjacent regions with similar attributes. Successful navigation through these stages demands meticulous attention to data structures, criteria governing growth and fusion, and a strategic distribution of germs.

2. Preprocessing

Before delving into the segmentation process, it is crucial to execute preprocessing steps, including operations such as histogram equalization. These tasks leverage tools developed in TP0, and additional functionalities like noise filtering have been incorporated. The utilization of these preparatory tools significantly enhances the overall robustness of the segmentation method.

2.1. Histogram Equalization

Prior to initiating the image analysis, we opted to equalize the image to enhance contrast if necessary. This approach facilitates easier and more effective image segmentation.

2.2. Image Blurring Filter

Prior to commencing image analysis, we also considered the option of filtering the image through a blurring process. This step aims to reduce noise, resulting in pixels with darker shades becoming more grouped or similar, thereby facilitating their organization into distinct regions. Throughout the project, we experimented with various filters, including the basic OpenCV **blur** function, **GaussianBlur**, **medianBlur**, and ultimately the **bilateralFilter**, all employed to effectively minimize unwanted noise.

3. Germ Distribution and Placement

The dimensions of the provided image are divided by 16 by default, employing modulo 4. Subsequently, a random value is selected within the range of 0 to the new dimensions resulting from the division by 16. Within each mini-region, this random value is then multiplied by a coefficient, determining the final position of the germ.

4. Region Computation

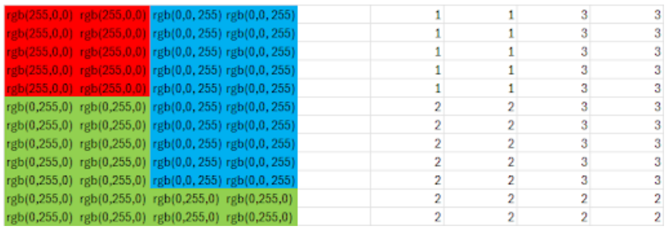
4.1. Initialization of Regions with Distributed Germs

In this step, we initialize our data structure to create various region objects, preparing them for subsequent growth and merging steps.

4.2. Region Growing

4.2.1 Understanding the Region Class

A region is a distinct object that possesses the length and height of the image, along with a pointer to the image in memory. Each region has access to a matrix table with the same length and height as the image, but it contains different IDs. An ID represents the number of the region. This approach enables us to determine which pixel belongs to a specific region.



rgb(255,0,0)	rgb(255,0,0)	rgb(0,0, 255)	rgb(0,0, 255)		1	1	3	3
rgb(255,0,0)	rgb(255,0,0)	rgb(0,0, 255)	rgb(0,0, 255)		1	1	3	3
rgb(255,0,0)	rgb(255,0,0)	rgb(0,0, 255)	rgb(0,0, 255)		1	1	3	3
rgb(255,0,0)	rgb(255,0,0)	rgb(0,0, 255)	rgb(0,0, 255)		1	1	3	3
rgb(255,0,0)	rgb(255,0,0)	rgb(0,0, 255)	rgb(0,0, 255)		1	1	3	3
rgb(0,255,0)	rgb(0,255,0)	rgb(0,0, 255)	rgb(0,0, 255)		2	2	3	3
rgb(0,255,0)	rgb(0,255,0)	rgb(0,0, 255)	rgb(0,0, 255)		2	2	3	3
rgb(0,255,0)	rgb(0,255,0)	rgb(0,0, 255)	rgb(0,0, 255)		2	2	3	3
rgb(0,255,0)	rgb(0,255,0)	rgb(0,0, 255)	rgb(0,0, 255)		2	2	3	3
rgb(0,255,0)	rgb(0,255,0)	rgb(0,0, 255)	rgb(0,0, 255)		2	2	3	3
rgb(0,255,0)	rgb(0,255,0)	rgb(0,255,0)	rgb(0,255,0)		2	2	2	2
rgb(0,255,0)	rgb(0,255,0)	rgb(0,255,0)	rgb(0,255,0)		2	2	2	2

Figure 1. This figure depicts our data structure: each pixel in the image aligns with a cell in the structure, color-coded to represent distinct regions. IDs are assigned to pixels, aiding in efficient region tracking during segmentation

4.2.2 Methods for Region Expansion

Within the realm of region growing, we employ two distinctive approaches for region expansion.

• **Fixed Iteration Approach:** Utilizing this method, we apply a predefined number of iterations. This provides a controlled and predetermined expansion, ensuring a specific count of iterations to gradually grow the regions.

• **Dynamic Iteration Strategy:** In contrast, the second method adopts a dynamic approach. Iterations continue as long as regions still possess the capacity for growth. The algorithm dynamically updates the state during each iteration for subsequent expansion cycles, with a predefined limit in place to prevent indefinite iterations.

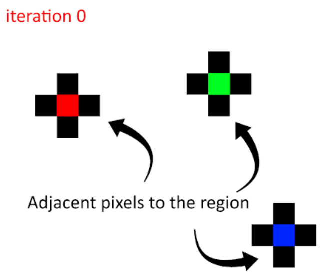


Figure 2. This illustration showcases the initial iteration, emphasizing pixels marked in black as **candidate** pixels. These candidates are potential additions to the outline vector, where subsequent decisions determine their inclusion or exclusion to the region.

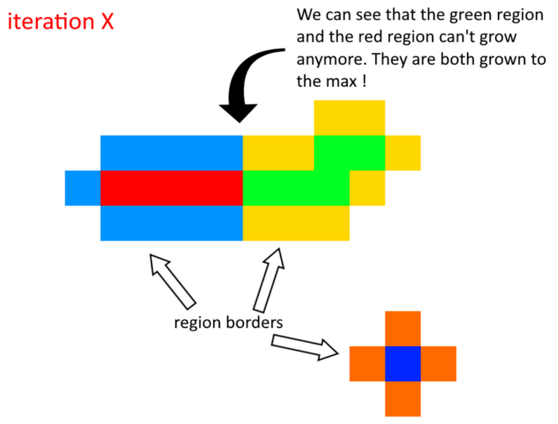

iteration X
We can see that the green region and the red region can't grow anymore. They are both grown to the max !
region borders

Figure 3. The green region has reached its growth limit. To address this, we consider enlarging its threshold. Consequently, the previously unaffected borders (depicted in purple) become part of the outline. Subsequently, a critical decision is required to determine whether these pixels will assimilate into the region or remain as part of the region's border.

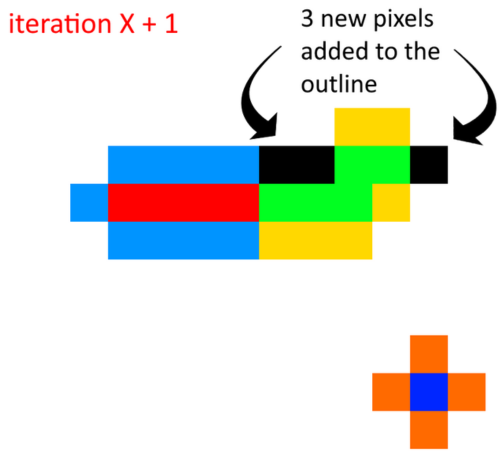


Figure 4. In this illustration, three pixels have successfully grown into the region, while the remaining pixels maintain their status as borders. These three newly added pixels represent potential expansions of the region.

4.2.3 Diverse Growth Criterion

During this phase, we dynamically compute the average color of the region in each iteration. As pixels grow, their colors are added to the vector of pixel colors, and we subsequently calculate their average. The decision to include a pixel in a region is contingent upon its color, determined by checking if the pixel's color falls within a user-defined range. This range is computed based on the average color across all regions.

In each iteration, the interval $[\mu - \sigma\theta, \mu + \sigma\theta]$ serves as the criteria for pixel inclusion in a region. Here, μ is the average of the region's pixel colors, σ is the standard deviation, and θ is a coefficient (set at 1). When the region grows sufficiently, we shift from a fixed threshold approach (previously set at 10) to a dynamic strategy. This dynamic strategy involves calculating the standard deviation and multiplying it by a coefficient (set at 1) to determine the new range.

Throughout iterations, these coefficients dynamically adapt to the specific requirements of each region. To ensure adaptability, we recalculate these coefficients regularly. If the interval is unable to expand further, we force coefficient updates. Subsequently, we multiply our standard deviation (σ), by these coefficients (θ) to recompute the new interval $[\mu - \sigma\theta, \mu + \sigma\theta]$.

This dynamic approach ensures that the growth criterion remains flexible and responsive to the evolving characteristics of each region during the segmentation process.

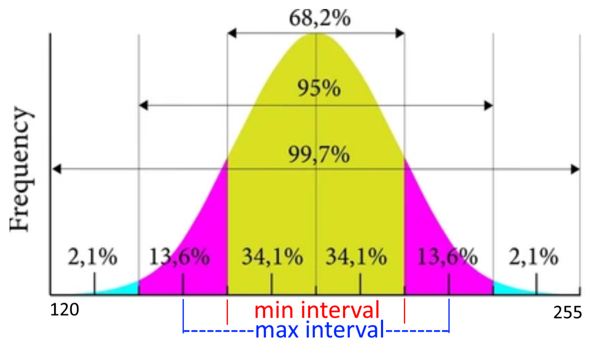


Figure 5. Standard Deviation Criteria

4.3. Region Merging

The process of merging different regions doesn't necessitate iterating over the entire image. Instead, we identify all the regions capable of merging. Once a region is merged, it is removed from the array of regions to be merged. To execute region merging, we iterate over the border of a region. If a border pixel belongs to another region, we prioritize merging the other region first. This process is recursive and carefully limited to prevent RAM overload.

Within the **Region** class, a function determines whether two regions are eligible for merging. Developing an effective merging criterion posed a considerable challenge. The most successful results were achieved by employing a fixed range of -25 to +25 of the average color of the regions.

Subsequently, we merge the outlines, accounting for any remaining outlines, and unify the borders. The borders are implemented using **unordered sets**, ensuring unique points. Redundant points are unnecessary in the border representation. We maintain a record of all merged IDs to facilitate memory cleanup. This step is crucial for reorganizing the array that indicates the regions of the pixels within the image.

5. Post-Treatment

5.1. Smoothing

Following the merging process, there might be pixels within a region that remain unmerged. To minimize the occurrence of isolated pixels not integrated into a region, we've implemented a smoothing function. For each individual pixel, if it is surrounded by pixels of the same region, we incorporate it into that region.

5.2. Absorption of Small Regions by Larger Ones

In certain images, there exist small areas within large regions that resist merging. Expanding the growth and merge criterion to address this issue might lead to sub-optimal segmentation. To tackle this challenge, we devised an "encompassment()" function. If a region, constituting less than 5% of the image, shares more than 60% of its border with another region, and the latter is four times larger, we initiate a merge between the two regions.

5.3. Recomputation for Improved Segmentation

Upon completion of the program execution, a brief analysis of the results ensues. This analysis scrutinizes whether more than 10% of pixels remain unassigned to any region. In the presence of such pixels, we trigger a reiteration of the growth, merging, and smoothing processes until the specified criterion is met.

The significance of this post-processing functionality lies in its ability to impart flexibility to our segmentation. For instance, when aiming for an exceptionally precise segmentation, we can reduce the growth criteria (and consequently fusion), prompting the program to incrementally incorporate pixels into regions through successive executions. Conversely, for images that are relatively straightforward to segment and require expedited processing, the growth and fusion criteria can be heightened. This approach results in a diminished number of unassigned (black) pixels[1] at the conclusion of the initial execution.

6. Run the Program

6.1. Compilation Process

For a successful compilation of the program, it is essential to have the OpenCV library installed. On Linux or macOS systems, the compilation process is streamlined by using the 'make' command. Once OpenCV is set up, simply execute 'make' to compile the code. In our Git repository, a dedicated 'tp0' branch has been created, encompassing

[1]Black pixels denote those that do not belong to any region.

solely the code pertinent to TP0. This branch facilitates a focused and efficient retrieval of the code relevant to the TP0 project.

6.2. Program Execution

Launching the program is a straightforward process. Using the command './tp1' will execute the program with a default image. The standard runtime for the basic program is just under 30 seconds. Throughout the execution, your terminal will provide a concise summary of the operations performed, along with the time taken for each function.

6.3. Arguments

To maximize flexibility, we have implemented a series of arguments that modify the program's behavior. Thus, based on the results, the user can relaunch the program with different criteria or functionalities. The list of arguments is explained in the project's readme, but here are the most important ones:

• **pathImage=Path/to/Your/image.png**

This argument allows you to select an image other than the default one.

• **pourcentSeed=0.8**

This argument allows you to set more or fewer initial regions.

• **pourcentReCal=1.2**

Represents the percentage of points not assigned to a region that will be used to generate new region seeds.

• **tauxPixelNoir=10.0**

Represents the maximum number of black pixels in the final image.

• **nbRepart=16**

Represents the number of parts when dividing the image.

• **nbIteration=100**

Represents the number of iterations for placing seeds and enlarging regions.

• **blur=3**

Represents the kernel size for filtering the image using the **medianBlur** method.

• **seuilMax=30.0; seuilMin=10.0**

These parameters modify the range values $[\mu - \sigma\theta, \mu + \sigma\theta]$, where μ is the average color of the region, σ is the standard deviation, and θ is a coefficient.

• **coefSDMin=2.2; coefSDMax=2.8**

These coefficients adjust the standard deviation during the growth and merging process.

All these options allow modification of the growth criteria and, by extension, the merging criteria.