Image Analysis - Lab Report

Tristan Guillard
tristan.guillard@etu.univ-lyon1.fr

Najib El Khadir
najib.el-khadir@etu.univ-lyon1.fr

***Abstract*—This document is the report of the MIF17 - Image Analysis Lab. The goal of this lab was to implement a germ growing algorithm. A $Germ$ data structure has been implemented using different vectors and queues of the STD library. The seeds are planted pseudo randomly and the growth of the germs happen in pseudo-parallel (each germ does a $round$ one after another) and then fuse to complete a satisfactory image segmentation of a gray scale image. Few parameters can be easily modified on the fly, such as the threshold, the number of germs, if the value of the germs will be modified or static and the neighborhood (4 or 8).**

I. Introduction

This document will describe in depth how the germ growing algorithm has been implemented, with its strengths and weaknesses, and on a bigger scale how all the program is constructed. The code has been programmed on Windows thanks to Visual Studio. The basic data structures such as images data are handled by openCV Mat data structure, and openCV is used for the basic image treatments. Our custom data structure is called $Germ$ and is used to aggregate points of a matrix. A basic openCV Mat called mask is also used as a map to have a quick access to which $Germ$ a specific pixel is affiliated to. Both the map and a list of germs work in parallel to aggregate and check affiliations in constant time. The algorithm uses the dissociated variant of the germ growing for germ fusing.

II. Main Algorithm

A. Initialisation

At the beginning, openCV is used to read the image, thanks to $imread(imgName, IMREAD\_GRAYSCALE)$, with $IMREAD\_GRAYSCALE$ turning the image into a matrix of $uchar$. Then a $mask$ matrix of int is built, with a default value of INT_MAX. This will allow us to check if a pixel belongs to a germ : if $mask(i, j)$ is $DEFAULT\_VALUE$, the pixel (i, j) is free, otherwise it belongs the the germs which number is in $mask(i, j)$ (constant time check).

B. Seeds planting

Once the mask as been initialized for future use, the seeds are planted in a pseudo random way : the image is cut in rectangles (the number of lines and columns can be easily changed above the main) and a seed will be randomly planted in each rectangle. The seed are then used to initialise a germ as its origin point.

C. Germ data structure

The Germ data structure is used for the germ growing algorithm. It contains a handful of private variables which are

- an ID which is unique among all germs and is referred by the pixels in the $mask$.
- a value that starts as the seed's gray scale value and either is static or can be updated on each round of the germ growing.
- a list (std::vector)* of every Germ ID the germ has as neighbor (will be updated) on the run
- a list of all the points of the germ and a list of all the border points of the germ (both dynamically updated on the run)
- a list (std::deque)* of points to be checked on in the "next round"

All these elements allow the program to run faster, as it does have to check only the right amount of pixels.

*list is only a descriptive word here, the concrete data structure used for those $lists$ is a $std :: vector$ or a $std :: deque$ depending on the use

D. Germ growing with rounds

This is the core of the program. On each round, each germ will check all the pixels surrounding its border according to the setup neighborhood. Then, if the pixel doesn't belong to any germ (thanks to the $mask$ - constant time), it is check if the pixel gray scale value can be agglomerated in the germs (relative to the threshold value and the germ value (which can be either static of dynamic)). If it does, the pixel is added to the germ and the border and points to check on the next round are updated accordingly. Once a germ has check all it's surrounding pixels (according to the beginning of the round), the germ value can be updated if set to dynamic and the next germ can do its growth round.

Rounds will succeed each other as long as one pixel has been aggregated in one germ. The growth then stop in a state where no germ can grow to none of it's surrounding pixel, either because they belong to another germ, or because they are too different in term of gray scale. Each germ will now have a list of pixel (and a list of "border" pixel) and a up-to-date list of its neighbor germs.

E. Germ fusion

The germ fusion is completely dissociated to the germ growth and occurs once the growth is completely over. Basically the "fusion" is only a equivalence table (formed by a std::map) that will map germs ID to a $germgroupID$. This

is done by comparing values of adjacent germs, if they meet the threshold, the germs will refer to the same $germgroupID$ in the equivalence table. As the germs are generated column-wise, and checked accordingly, there are some situations where two should-be-fused germs are temporarily assigned different colors. In order to fix this, each newly added germ to the equivalence table will check if it's neighbor could fuse, and if so an entire "germgroupID" will be removed of the table.

F. Display

Both a before-fusion and after-fusion RGB-8bit image of the germs and their border are created and displayed. The first one allows to see how the germs got generated and grew, as the second should give us a reasonably segmented image according to the gray scale value of the pixels. For the before fusion image, the borders are quite easy to display as they belong the the germ data structure, whereas for the fused-germ image, a new border has to be post computed using the unfused germs border : each point will seek if one of its neighbor don't belong to the fused germ

III. Results

A. Flat colors

Here is the result of the algorithm worked on flats colors. It requires only a small number of germs (225 or less) to have a satisfying result. For each image, the algorithm works on the grayscale image, and outputs both an image with every germ and their border (in black), and an image with the fused germs and their border.

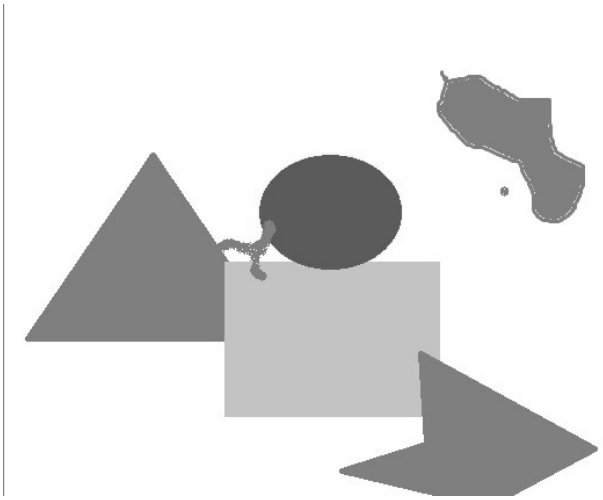

Fig. 1. Grayscale image 1

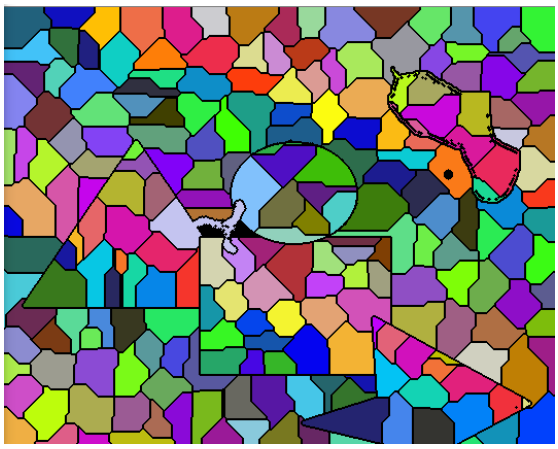

Fig. 2. unfused germs after growth on image 1

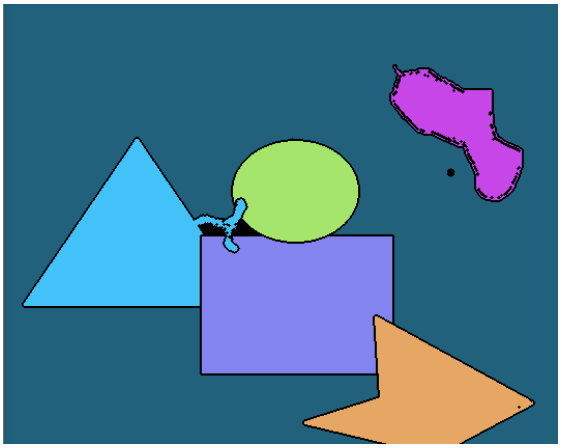

Fig. 3. fused germs after growth on image 1

B. Line art and shades

To segment a more complex picture, such as colored line art, the algorithm outputs a satisfying result with a normal amount of germs (225 or less), but a way better one with a really high number of germs. With a low enough threshold, it can even differentiate shading and glowing as different sections

Fig. 4. Grayscale image 2

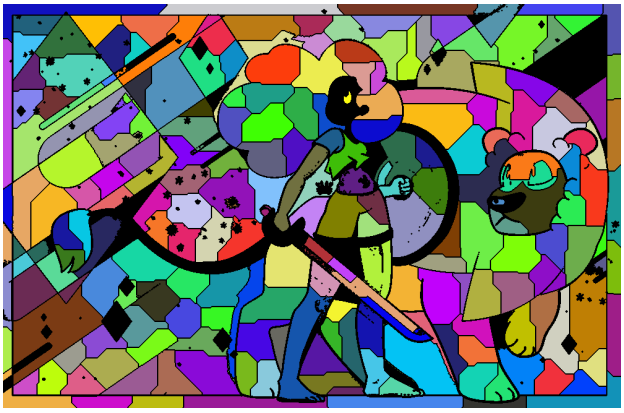

Fig. 5. unfused germs after growth on image 2

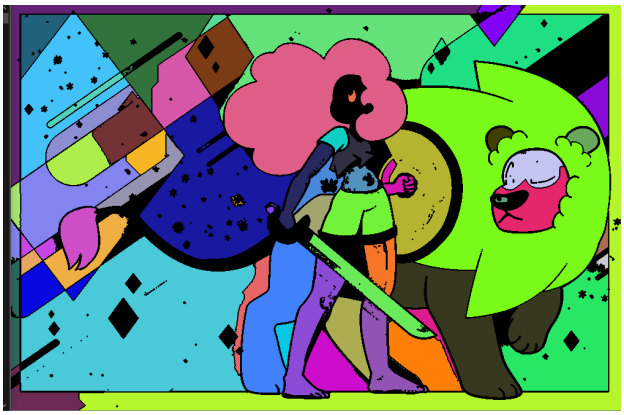

Fig. 6. fused germs after growth on image 2

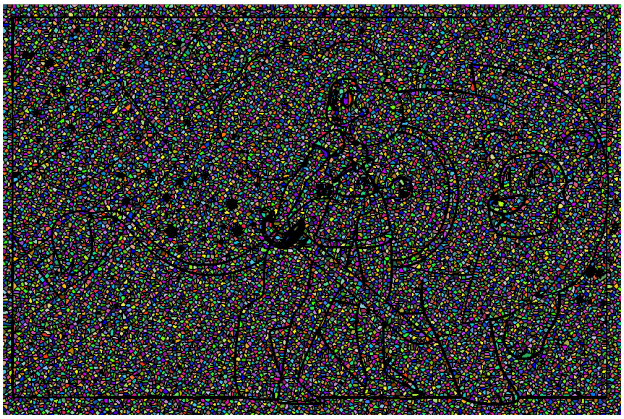

Fig. 7. unfused 22500 germs on image 2

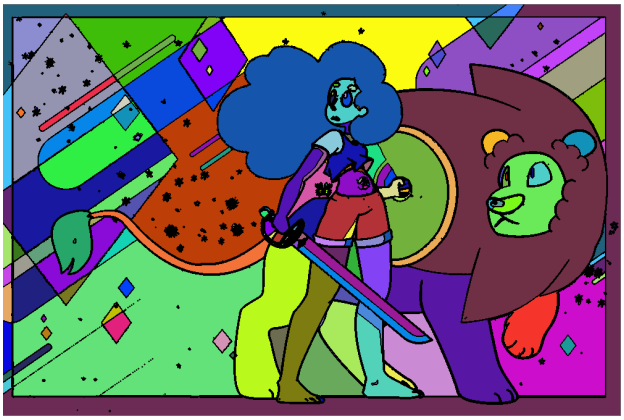

Fig. 8. fused 22500 germs on image 2

C. Painting (textures and life-like images)

This is the weak point of our algorithm as the threshold is static. It can be a little counterbalanced with a dynamic germ value. The algorithm should be able to differentiate the major parts of the image though, but a high number of pixels won't be in any germ due to the high variation on the pixel values.

Fig. 9. Grayscale image 3

Fig. 10. unfused germs after growth on image 3 with static germ value

Fig. 12. unfused germs after growth on image 3 with dynamic germ value

Fig. 11. fused germs after growth on image 3 with static germ value

Fig. 13. fused germs after growth on image 3 with dynamic germ value

IV. Reflection on the program

A. Weaknesses and what could be improved

- The main weakness of the program is the threshold used to discriminated germ belonging. It basically check if the gray scale value of a pixel is within a certain range of the germ value. This could be greatly improved by checking things such as number of points in the germ, or variance of values, etc...
- The program downscales any image to gray scale as it doesn't support colors differences due to the threshold. This leads to the fusion/growth of regions that shouldn't happen as the RGB colors are different.
- The program is fairly slow (takes few seconds to compute) for a preprossessing process.
- The program work on a pixel perfect scale, but we could develop techniques to agglomerate chunks of pixels at once, which could fasten the algorithm on large sized images.
- There is no preprossessing of the image before the application of the germ growing algorithm, which could explain why the algorithm isn't as effective on life-like images.

B. Strengths

- The program has flexibility on the following points :
  - Number of seeds planted (per column and row)
  - neighborhood used (4 or 8)
  - Staticity/dynamism of the germ values
  - threshold

  This flexibility allow the user to adapt the code to the image up to a satisfying result.
- The program is very resilient and shouldn't ever crash whatever the input (even though it can take quite a while on very big images) and the parameters. For example, it can work with more than 22500 germs in less than 10 seconds.
- The program is well segmented, which allow future updates to be developed easier.
- The coloration of the germs is almost random and non cyclic which allows to clearly distinguish the germs even if they are really small. This is due to prime number and modulo properties

C. Problems encountered and solutions

- Originally designed with dynamic germ values, we had a lot of struggles computing satisfying regions. We developed static germ values as a temporary check, but the regions were always more in line with what we wanted to have. We let dynamic germ value (which, on each round, will update the germ value according to the mean of it's current value (weighted by its size) and the mean value of the added-on-this-round pixels (weighted by the number of pixels added)) as an option because it is implemented, but on the images we tested, the results were always better with a static germ value.
- Having a proper data structure was a quite a problem at first. We started with only the Germ structure, but the more they grew, the slower it was to check if a pixel belonged to a germ or not. Then we had the idea to add a matrix as a map to check if a pixel belonged to a germs without having to check all the germ's list of pixel.
- The borders of the germs were quite hard to keep tack of at the beginning, along with the to-check-next-round pixels, both of which we added to the germ data structure to update in a proper way each round. Thanks to the map both can really fast check if they touch another germ and can grow the germ easily.
- All data structures and idea were found on the run. Despite reading a lot of literature, none provided ideas to help on our problems, which we handcrafted solutions for. This result in a kinda clunky pair of structures, which works but is still quite slow.

V. Conclusion

This germ growing algorithm developed here works in pseudo-parallel and pseudo-random as requested, and give really satisfying results, especially on line-art styled images. It may be a little slow for a preprossessing process, but it is really stable and can support a really high number of germs.