02132 Computer Systems F23 – Cell Detection

Group 3

Mads Dan Eriksen
s204170
Cyberteknologi
DTU
s204170@student.dtu.dk

Mari Balke Fjellang
s231450
Cybernetics and Robotics
DTU
s231450@student.dtu.dk

Danmarks Tekniske Universitet
Cyber Technology (BSc Eng.)

October 1, 2023

Contents

1 Work Distribution

We started the project working in parallel so each member got an intuition for how the handed out code snippets worked. When we reached the capture part we began to delicate tasks. Mari was in charge of capture optimization and Mads was in charge of Otsu optimzation. Each member have their own method file with code they are in charge of. There is also a collective main method where you can select between Mari's and Mads's test.

2 Design

We decided to implement the different steps in the algorithm as functions, each responsible for one specific part. This is according to the single-responsibility principle, which also makes the job of debugging easier, in addition to keeping the code tidy. Further on, we decided to keep the algorithm as an independent and separate module, with a corresponding c- and header-file. The functions are declared in the header-file, facilitating a clear and accessible overview of the module's interface.

3 Implementation

Pointers are a fundamental element of the C programming language, serving various purposes, including storing the memory address of a variable. We decided to use pointers to be able to change the value of variables inside functions, i.e. passing them by reference. This was very useful for deciding when to exit the erosion-loop in the algorithm, and for printing out how many cells were captured in total for each sample. Compared to passing by value, passing by reference is more efficient if the goal is to assign a new value to a value, because the argument is not copied.

As mentioned in section 2, the program is designed with different modules. Another key feature of the C programming language is that it allows for splitting code into different modules, which both improves the visuality of the code as well as it opens for reuse of code.

4 Optimizations and Enhancements

4.1 Erosion of edges

The first optimization that was implemented was the handling of the edges of the image during erosion, because we observed that the cells located close to the edges wasn't always captured. We solved this by eroding all pixels along the edges of the image, in order to detect the cells that typically are splitted by the edge. Once having the pixels at the edges removed, the exclusion window allowed us to gradually erode the rest of the cell.

4.2 Different functions for erosion

The next optimization, with regards to both cell detection and execution time, was to implement two different functions for erosion. The only difference between the two functions is the structuring element they are using, which makes the behavior somewhat different. The first function, *erosion()*, was implemented based on the structuring element given in the assignment description, meaning that a pixel is eroded unless the pixels above, below, to the left and to the right are white. The second function, *erosion_strict()*, requires that all pixels in a 3x3 structuring element are white in order to keep the specific pixel white, meaning that this function has a more aggressive approach with respect to erosion, and performs better in terms of splitting the cells from one another.

From extensive testing, we found that *erosion_strict()* gives us good results during the first few iterations of erosion, but is less impressive as soon as there are few pixels left, as pixels are removed faster than we can capture. This led to the decision of using both functions, as can be seen in the code snippet in figure

1. The number of iterations in the for-loop was also decided after testing with different values, where it was found that six iterations gave the most improvement for the different samples.

```
start = clock();
//Load image from file
read_bitmap(argv[1], input_image);

//Run gray_scale
gray_scale(input_image, output_image);
copy_image(output_image, final_image);

//Run binary threshold
binary_threshold(final_image, output_image);
copy_image(output_image, input_image);
int i = 0;

/// First a few iterations of erosion_strict()
for(int i = 0; i < 6; i++){
  erosion_strict(input_image, output_image, countPtr);
  copy_image(output_image, input_image);
  capture(input_image, final_image, capturePtr);
  i += 1;
}

/// Then continue with erosion() until there are no more cells to erode
while (*countPtr > 0){
  erosion(input_image, output_image, countPtr);
  copy_image(output_image, input_image);
  capture(input_image, final_image, capturePtr);
  i += 1;
}
end = clock();
```

Figure 1: Code snippet of program - using two different functions for erosion

4.3 Utso Optimzation test

The Otsu method is a an image processing algorithm that returns a threshold that separates the background and foreground. Using this we hope to get a better threshold then the one provided in the assignment. We implement it using [1.Tay] tutorial where we build a histogram of shades and then perform some math operations to find the optimal threshold.

4.3.1 Building the Histogram of shades

For the Utso implementation we build a histogram of the grey shades (256 shades) using an array of 256 entries and initialize all it's indexes to 0. Then we run a nested loop on the image and for each pixel increment the histogram index depending on its value.

4.3.2 Utso Math part

The optimal threshold is between 0 and 255, we run an Otsu test on each possible threshold and pick the test that maximizes the in between class variance. For a test we go through the histogram to calculate the

sum and mean of the foreground and background depending on the test-Threshold. For this part we use int variables to make it go faster. For the second part when we have to calculate weight, intensity and variance we convert into float types for the division and power operations.

5 Test and Analysis

5.1 Memory use analysis

When developing the program, we aimed for maximum reuse of the arrays for storing images. In order to use the correct image as input-image to the next function, the output-image is copied after every operation. By doing this, we are overwriting the previous input-image with the newest output-image, and therefore reusing the memory allocated to these images. An additional array was allocated to store the gray-scale image, which is the final output image where detected cells are located.

5.2 Functionality tests and performance

The algorithm manages to detect close to every cell from all of the easy samples. An example of this can be seen in figure 2. When running the algorithm on the samples of the medium difficulty, we observe that we are not able to capture every cell. In addition, the program uses a few more iterations and the execution time is consequently increased. An example of this can be seen in figure 3. Lastly, when running the program with the hard samples, we observe an additional decrease of detected cells, and a small increase in execution time. The results can be seen from figure 4.

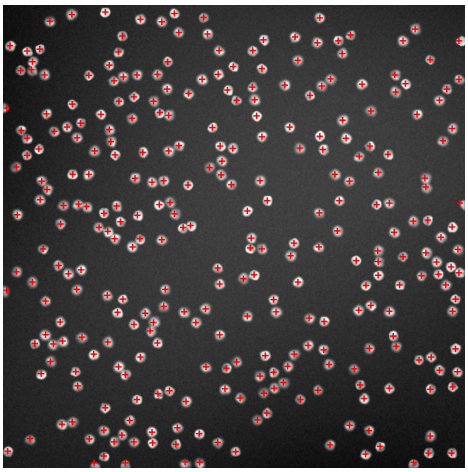

Figure 2: Detected cells for sample 1EASY, where 300 cells are detected

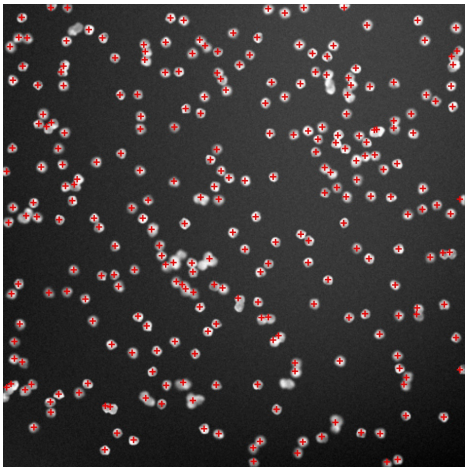

Figure 3: Detected cells for sample 1MEDIUM, where 270 cells are detected

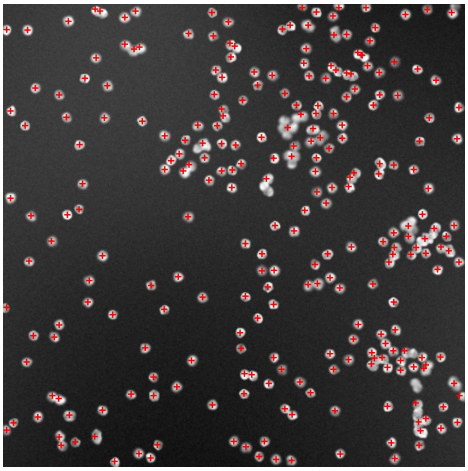

Figure 4: Detected cells for sample 1HARD, where 263 cells are detected

5.3 Execution Time Analysis

A table with the respective execution-times, number of iterations and detection rate for each difficulty level is displayed in table 1. We notice that with the increased difficulty of the samples, the program require extra iterations of erosion in order to distinguish between the cells that are overlapping, and this is resulting in increased execution time.

Sample	Execution time	No. of iterations	No. detected cells
1EASY	$[3.1 - 3.3]s$	6	300
1MEDIUM	$[3.9 - 4.1]s$	9	270
1HARD	$[5.4 - 5.6]s$	15	263

Table 1: Performance with optimization of the erosion-loop

5.4 Results from optimization

We started by having only one function for handling erosion, which is the one using the same structuring element as in the assignment description. When having the algorithm as shown in figure 5, we could

observe that the number of iterations and the execution time was higher compared to after implementing the optimization. Table 2 shows the approximate execution time, number of iterations and number of detected cells without this optimization. Compared to table 1 we can easily see that the optimization is valid.

```
start = clock();
//Load image from file
read_bitmap(argv[1], input_image);

//Run gray_scale
gray_scale(input_image, output_image);
copy_image(output_image, final_image);

//Run binary threshold
binary_threshold(final_image, output_image);
copy_image(output_image, input_image);
int i = 0;

erode_edges(input_image);
while (*countPtr > 0){
  erosion(input_image, output_image, countPtr);
  copy_image(output_image, input_image);
  capture(input_image, final_image, capturePtr);
  i += 1;
}

end = clock();
```

Figure 5: Algorithm without optimization of the erosion.

Sample	Execution time	No. of iterations	No. detected cells
1EASY	$[3.7 - 4.0]s$	11	300
1MEDIUM	$[4.9 - 5.2]s$	15	269
1HARD	$[6.2 - 6.4]s$	20	265

Table 2: Performance without optimization of the erosion-loop

5.5 Utso result

For most test we get a best threshold of 114-115 which is 25 shades away from the given threshold of 90. But when running the test we see that it does not increase performance in capturing, we get more duplicates on easy and less captures on hard. So in conclusion we can say that Utso method does not help plus all the extra computation overhead it adds increases the computation time making the program worse. It might be usefull in other problems where the images vary more in shades.

6 References

Jian Wei Tay - Otsu's Method - 20/09/2023
https://www.youtube.com/watch?v=jUUkMaNuHP8