02132 Assignment 1 report

Software implementation of a cell detection and counting algorithm in C

Group: 22

Mikkel Arn Andersen **s224187**
Niclas Juul Schæffer **s224744**
Rasmus Kronborg Finnemann Wiuff **s163977**
github.com/rwiuff/02132Assignment1

October 1st

1 Work distribution

Explain here who has done what, for both implementation and report.

Table 1: *Work distribution on the project*

Name	Implementation tasks	Report tasks
Mikkel Arn Andersen	Erosion, Detection optimisation	Sections 4.3 and 5.2
Niclas Juul Schæffer	Detection, Detection optimisation	Sections 3, 5.1 and 5.2
Rasmus Wiuff	Program structure, Detection, Detection area control	Sections 2, 4.1, 4.2 and 5.1

2 Design

2.1. Data structures

There are two kinds of information needed in the program. Incrementors of various sorts for counting and keeping checks on processes. These are mainly of type `int` as we need to iterate over 950 pixels per loop, thusly an 8 bit char wont suffice, even though they have a larger drain on memory. For storing the image there exists two arrays: One for the original 3-channel image (provided by `cbmp.c`) as well as a grey scale array: `unsigned char tmp_image[BMP_WIDTH][BMP_HEIGTH]`. This is practical as only one conversion is needed to produce a grey scale array, as opposed to convert back and forth before saving. The original image array is only edited when a location is marked with a cross. The temporary array is used in all other calculations and changes at every step in the process. Another good reason for using array is the fact that C always pass them by-reference to functions, meaning no copying is needed when mutating the original arrays.

2.2. Program structure

Firstly the image needed to be converted into an array with the provided function. Another function deals with flattening the array by averaging the 3 channels into one. Hereafter the program follows the provided algorithmic structure. A function applies a binary threshold onto the array. A do-while loop checks if any changes has occured in the previous erosion, or at the start assumes there will be change coming, As long as that is the case a round of erosion is made using a separate function. Hereafter a detection function does the following:

1. Iterate over pixels
2. Use a function to check if there is a valid cell
3. Increment a counter and draw a cross on the original image array using another function
4. Print information about the location to the console

Lastly the main method prints information about runtime, counted cells, etc. Table 2 shows functions and functionality.

Table 2: *Functions and their scope*

Step	Function	Effect
0	read_bitmap	Imports bitmap as a three dimensional array
1	Grey-scaling	Populates two dimensional array with greyscale image
2	Binary threshold	Applies a binary threshold on the greyscale image
3b	Erosion	Erodes image using erosion element
3a	Pixel check	Check for white pixels
3c	Detection	Run detection routine
3c1	Draw	Draw cross on original array
4	write_bitmap	Exports image array as bitmap

3 Implementation

Our final design does not include C specific implementations, due to how heavy they are in time costs to implement, however it will be touched more upon in the test and analysis part, how sections of the program has been using bit wise operations to reduce memory requirements.

3.1. Cell detection

For the detection function, we have tried 3 different approaches.
In the first implementation of the cell detection algorithm a DFS search was used to detect the cells and it worked till a certain extent, besides the scenarios where the cells were touching. It found every cell that were alone on the grid. The concept about a DFS algorithm is finding pixels around the focused pixel, and then work around until it have detected the whole cell.
In the second implementation of the cell detection algorithm we used the 1 pixel exclusion frame and 12×12 pixel capture area described in the assignment. For every pixel for loops would first scan the exclusion frame. If no cells were found, for loops would scan the capture areas content. If a cell was found a cell counter would increment and the coordinates be printed to the console. If a cell was captured, or not, the algorithm would jump a capture area ahead before beginning the scan again. This concept was kinda slow because two for loops were applied over capture areas of the picture after every erosion. For this reason we a new approach was made.
The third implementation was based primarily on erosion. We would erode the picture so much that there was only a single pixel left for each cell and then scan the picture of white pixels in the detection, and this turned out to be a lot faster.

4 Optimisations and enhancements

4.1. Otsu's method

Otsu's method was extensively looked into in order to get an automated optimal intensity for the binary threshold. Following was discovered:

- Otsu's method works best if two distinct classes of intensity exists, i.e. two distinct peaks in a intensity histogram over the image.
- Most pictures in the set does not contain two distinct peaks.
- Existing implementations where tried to find optimal intensities (EBI package for R, OpenCV (using Python)). These tools pointed to intensities way higher than empirically studied values (115-150 against the provided 90 or tested 80-110).

A way of overcoming the problem could be through local thresholding, where ranges of intensities are left out, however time was already wasted on this rather time consuming endeavour.

4.2. Size of detection area

There are two ways of changing the granularity in the search for cells. One can either change the erosion element or one can increase/decrease the size of the detection area. In here the later is discussed. The idea is that cells will

need to be much smaller and therefore disconnected from neighbouring cells, addressing the issue of connected cells being counted as one. Practically the implementation is passing a variable to detection functions and use the variable to constrain search and exclusion indices. The smaller the detection area, the more erosion rounds are needed and therefore runtime is increased. It later turned out that our implementation of smaller detection areas yielded worse results and therefore it was abandoned.

4.3. Erosion and Design enhancement

Our first iteration of the program was quite slow; on the testing machine it took approximately 2000 ms to complete just 1 image and print it out. We felt this was unacceptable for the task given, as a small embedded system with much slower running speed could take multiple digits of seconds, and worse if 1 CPU has to run through multiple droplets. Therefore we decided to optimise for speed to suit this task better. Instead of going through a loop of erode → detect → erode and so on, we opted to do all the erosions at once, and then a detection round. This change sped up the program by about 8-12 times by sacrificing a bit of potential accuracy. However the accuracy might be raised if it was possible to take multiple pictures a second and average them in the real setting with all the time saved, thusly improving speed and likely accuracy too, as impossible cells that are overlapping might have shifted and given a clearer view.

4.4. Memory optimisation

We tested and tried changing the data structure and function of our code to package together bits, as after the threshold is applied, we only care whether they are black or white. We packed 8 different coordinates into 1 which reduced the tmp-array down to 950x119, a drastic reduction in byte size, Listing 1 and 2 showing this concept in code snippets. However it was costing a very large development time, leading to an unsuccessful implementation. It also still didn't touch upon the main issue, being the image array, which needs full RGB 0-255 data, which means we don't have any unused bits for the unsigned char coordinates to package together. If we wanted to reduce the size-load of this array, we considered to represent it in another way entirely through processing, or only load part of the image at a time.

Listing 1: *Greyscale conversion*

```
1  void grey_scale(unsigned char input_image[BMP_WIDTH][BMP_HEIGTH][BMP_CHANNELS],
   ↪  unsigned char tmp_image[BMP_HEIGTH][CompressSize])
2  {
3      for (unsigned char i = 0; i < BMP_HEIGTH; i++) // Iterate over Height
4      {
5          for (unsigned char j = 0; j < CompressSize-1; j++) // Iterate over Bit_Width
6          {
7                     unsigned int Step = j*8;
8                       for (unsigned char BitLoop = 0; BitLoop <= 7; BitLoop++)
9              {
10                 ((input_image[i][Step+BitLoop][0]
                   ↪  + input_image[i][Step+BitLoop][1] + input_image[i][Step+BitLoop][2])
                   ↪  / 3) <= 90 ? continue : tmp_image[i][j] |= 1 << BitLoop ;
11                      }
12         }
13         for (unsigned char BitLoop = 0; BitLoop < 6; BitLoop++)
14         {
15              ((input_image[i][(944)+BitLoop][0]
                 ↪  + input_image[i][(944)+BitLoop][1] + input_image[i][(944)+BitLoop][2])
                 ↪  / 3) <= 90 ? continue : tmp_image[i][118] |= 1 << BitLoop ;
16         }
17     }
18 }
```

Listing 2: *Erosion method*

```
unsigned char erode(unsigned char tmp_image[BMP_HEIGTH][CompressSize])
{
        unsigned char Change=0;
    unsigned char ErosionMap[BMP_HEIGTH][CompressSize] = {0}; // tmp_image sized matrix of 0's

    for (unsigned char k = 0; k < CompressSize-1; k++)
    {
                tmp_image[0][k] =0;
        tmp_image[BMP_HEIGTH-1][k] =0;
    }
    for (int i = 1; i < (BMP_HEIGHT-1); i++)
    {  //&= ~(1 << 0)
           // Cut off Border to ensure erosion and less fringe cases
        tmp_image[i-1][0] &= ~(1 << 0);
        tmp_image[i-1][CompressSize - 1] &= ~(1 << 5);
        ErosionMap[i][0] = (tmp_image[i+1][0]&tmp_image[i-1][0]&(((tmp_image[i][0]&=
         ↪ ~(1<<0))>>1)+128*(tmp_image[i][1]&(1<<0))))
                ErosionMap[i][CompressSize-1]
                 ↪ = (tmp_image[i+1][0]&tmp_image[i-1][0]&(((tmp_image[i][0]&=
                 ↪ ~(1<<0))>>1)+128*(tmp_image[i][-1]&(1<<7))))
        }
    for (int j = 1; j < (CompressSize-1); j++)
    {
                ErosionMap[i][j] = (tmp_image[i+1][j]&tmp_image[i-1][j]&(((tmp_image[i][j]&=
                 ↪ ~(1<<0))>>1)+128*(tmp_image[i][j+1]&(1<<0)))&(((tmp_image[i][0]&=
                 ↪ ~(1<<0))>>1)+128*(tmp_image[i][-1]&(1<<7))))
    }
```

5 Test and analysis

5.1. Proof of concept version

This section will discuss the non-optimised version of the program.

5.1.1. Functionality tests Table 3a shows an overview over detection on the various images. The detection rate decreases as difficulty of the pictures increases. This is largely due to cells clumping together, and is expected from the get-go.

Table 3: *Functionality test on provided images. Percentages are relative to the actual number of 300 cells.*

(a) *Proof of concept implementation*

	Easy	Medium	Hard	Impossible
1	100 %	88.3 %	87.3 %	72.3 %
2	99.7 %	88.7 %	79.0 %	73.7 %
3	100 %	88.0 %	81.3 %	73.0 %
4	99.3 %	88.0 %	83.0 %	72.7 %
5	99.0 %	83.0 %	85.3 %	76.3 %
6	99.0 %	92.0 %	86.7 %	N/A
7	99.3 %	86.7 %	86.7 %	N/A
8	100 %	82.3 %	87.7 %	N/A
9	99.7 %	86.0 %	87.3 %	N/A
10	99.7 %	88.0 %	83.0 %	N/A

(b) *Optimised implementation*

	Easy	Medium	Hard	Impossible
1	97.3 %	98.7 %	95.3 %	88.3 %
2	97.0 %	96.3 %	89.3 %	86.3%
3	96.0 %	96.3 %	91.0 %	87.7%
4	96.3 %	95.7 %	95.3%	84.0%
5	95.7 %	91.3 %	99.0%	88.0%
6	98.0 %	99.0 %	97.7%	N/A
7	96.7 %	93.0 %	95.3%	N/A
8	99.0 %	91.0 %	96.0%	N/A
9	96.3 %	93.0 %	95.0%	N/A
10	95.7 %	97.3 %	93.3%	N/A

5.1.2. Execution time analysis We are using a imported the tool "time.h" to track the time used in the program, where we set a start and end value of clock types. We can then calculate the time spent in the program by subtracting those values, multiply by a thousand and divide with the value called CLOCKS_ PER_ SEC that are set in the time.h header file. This produces the runtime in milliseconds.
Most functions on the grey scale image array iterates on one axis while iterating the other in a nested loop. If cells

are found a small nested for loop is performed in the capture area and therefore considered negligible. To save time, whenever a cell is found it is removed. Therefore the number of erosion and detection iterations depends highly on how quickly cells erode to a detectable size. Runtime is estimated as $O(n^2)$ for an image of size $n \times n$.

5.1.3. Memory use analysis Two arrays are consistently used throughout the program. The arrays have sizes:

$$\text{image\_array} = n \cdot n \cdot 3 \cdot 8.00 \text{ bit} = n^2 \cdot 24.0 \text{ bit} \tag{5.1}$$

$$\text{tmp\_array} = n \cdot n \cdot 8.00 \text{ bit} = n^2 \cdot 8.00 \text{ bit} \tag{5.2}$$

Additionally, for the erosion we temporarily use an array Mask of the same dimensions as tmp-array thusly, for the 950×950 pixel images this means the sum of memory allocated at the heaviest memory load time is

$$950^2 \cdot 24.0 \text{ bit} + 950^2 \cdot 8.00 \text{ bit} \cdot 2 \approx 4.51 \text{ MB} \tag{5.3}$$

The memory is largely depending on the arrays and as such uses $O(n^2)$ space for an image of size $n \times n$.

5.2. Optimised version

Here the program version with an optimised erosion and detection algorithm is analysed.

5.2.1. Functionality tests Table 3b shows the results of the functionality test using the optimised version. After we tested the newer version and put them up side by side and then we can see it was better in testing overall. It was worse than the other one on easy but was better in the other difficulties. We much prefer this result, as it should be very apt for the task given, a camera picture of real life cells could prove clumps that are fully impossible to prove the amount of cells within from a picture, thusly there should never be a reliance on specifically the program to be correct every time, and it would make more sense to use it as a tool in either calculations or understanding, wherein a 5% margin of error should suffice quite well.

5.2.2. Execution time analysis The final implementation uses a lot less time because we are getting rid of the reading an image multiple times every erosion. Since we got rid of the border frame detection, now the program uses erosion until it has no changes left. Then it will call detection and search for white pixels, because the erosion has left out a single pixel for every cell, this way it's way faster, with an avg time of 387 ms on a poorly equipped laptop.

5.2.3. Memory use analysis The final implementation uses the same memory as mentioned in Section 5.1.3 This is a large amount of memory required for a micro computer, and will have to cost a small investment. Small computers like the Arduino board Portenta H7[1] has sufficient memory therefore a requirement of less than 8.00 MB seemed a sufficient answer for an immobile micro-scope computer however, Which makes us comfortable with this memory requirement as to not hurt the runtime by reloading and splitting memory.

References

[1] Arduino, José Bagur, Taddy Chung *Arduino Memory Guide (19/09/2023)*
https://docs.arduino.cc/learn/programming/memory-guide