**CSC3060 AIDA – Assignment 1**

Adam Coyle

40178464

October-November 2019

# Introduction

Welcome to my report for AIDA assignment 2. You’re in for a treat.

# Section 1

I wanted to get stuck into this assignment, so I sat down for over an hour and produced all 160 doodles. Using GIMP, it was easy to get into a rhythm for exporting small images out as JPEG files. I initially tried to export a test doodle as a PGM file but found that the code I had written was unable to convert it to a csv. From placement experience, I knew how to iterate through a JPEG formatted image and check the RGB elements to determine if a pixel was white or black. Now I could convert an image to csv format, I needed the Java equivalent of the C# LINQ statements, which I could quickly find on Stack Overflow and use to iterate the doodles in a specified directory. Lo and behold, I had 160 csv files.

# Section 2 – 1600 words

For your information, I am not in any way proud of the code written for this section. It is full of redundancies and violates a ton of programming patterns, but I’m told that I’m not assessed on how elegant it is, so it was good enough to have working code rather than pretty code. I had most of it done the week after the assignment was sent out, and after that it was just a month of resisting the urge to waste time refactoring. I commented it as best I could to explain what was happening, but if something isn’t clear then I apologise.

## Section 2.1 – nr\_pix

This feature was a nice and easy start into the nitty gritty of Section 2. Thankfully, the csv files only contained 1s and 0s where 1s were to represent black pixels. To calculate this feature, I used a double for loop to iterate through the contents of the csv file, and for every 1 I encountered, I increased the blackPixelCount by 1, finally returning the sum of black pixels in the file.

## Section 2.2 – height

I was surprised at having to use some degree of logical thinking this early into the section, and somewhat anxious about what was to come after. After seeing that the previous feature was calculated by iterating through a 2d array, I decided to stick with this as my main method of calculating features. To start, I created two int variables, topmost and bottommost and initialized them to -1 (since arrays don’t have indexes of -1).

After the creation of these variables I iterated through the array, and for every black pixel I encountered, I checked the row indexes against the values of topmost and bottommost.

* Since I iterated from the top-left to the bottom-right, I could assume that the first black pixel would be at the top of the image, so I assigned this to topmost only if topmost was less than 0 (making the assumption that the first black pixel had the lowest row index – making it the top of the image).
* The value of bottommost was simply the lowest row index encountered during the iteration, so a check to see if the row index value was greater than bottommost was enough to find this.

Finally, with the values for topmost and bottommost, I was able to calculate the height by getting the difference (absolute value) between topmost and bottommost and adding 1 to the result (since arrays begin at 0)

## Section 2.3 – width

The calculation for the width feature was identical to that of the height, just turned on its side. I had two int variables, leftmost and rightmost. The difficulty here was that leftmost needed to be initialised to the value of the array size (50) such that it could be assigned to columns that had a lower value than it (i.e. to move left in the array). The rightmost variable followed the pattern of bottommost from the previous feature, such that it had the index of the rightmost black pixel.

As with the height calculation, width was the absolute difference of the leftmost and rightmost plus 1 to account for the array offsets.

## Section 2.4 – span

This feature took some careful thought to implement. I knew that I would need to iterate over every item, *for every item*, but I realised that I only had to check for every black pixel, since white pixels were irrelevant. Therefore, I found it necessary to add a helper function to return a list of the black pixel indexes. A list became essential since using an array requires a predetermined size, which I wouldn’t know until the end. I could loop through each black pixel, checking its Euclidean distance to the other black pixels, returning the greatest distance as the span.

## Section 2.5 – rows\_with\_5

This feature was fairly straightforward to implement. I initialised an int, numberOfRows, to 0. This would be value returned at the end of the function call. For every row in the array, I initialised another temporary int, blackPixelCount, and proceeded to iterate through the columns of the array. If a black pixel was found, the value of blackPixelCount was increased. If this value was greater than or equal to 5 then numberOfRows would could incremented. Once the iteration had finished, the value of numberOfRows was returned.

## Section 2.6 – cols\_with\_5

The implementation for this feature was identical to that of the previous feature, the only difference being that when accessing the values of the array, the column and row indexes were swapped so that the loop would go down the column instead of across the row. So, the same logic as before applied and the function returned the value of numberOfColumns.

## Section 2.7 – neigh1 and neigh5

As before, I begin the feature calculation with an iteration over the entire array. I also create two int variables, pixelsWithOneNeighbour and pixelsWithFiveNeighbours. If a black pixel was found during the loop, then I would have to check all 8 neighbours for black pixels and increment the aforementioned variables appropriately. This was where I really became conscious of making reusable code for scanning the neighbours of a tile.

I wrote a helper function called CountNeighbours, that took in the row and column index of the current pixel in the loop. I would have to loop over everything from the top left (column-1, row-1) to the bottom right (column+1, row+1) to check for black pixels neighbours, and increment a neighbourCount value for each one. This is where I experienced a real issue which was that the pixels at the edges of the array would throw out of bound exceptions when trying to access some of their neighbours. Now I needed yet another helper function to produce a solution – IsNeighbourValid

The IsNeighbourValid function simply checked if either the row or column to be checked was outside of the array (less than 0 or greater than or equal to the size of the array). Once this function was implemented, I could check all neighbours of a pixel without any issue, and thus the rest of the function would count the neighbours and return the result.

## Section 2.9 – 2tiles,

## Section 2.15 – horizontal3tile

## Section 2.16 – vertical3tile

## Section 2.17 – nr\_regions

This feature was fairly challenging, but also fairly enjoyable to calculate. I took up a notepad and sketched up some dummy data to simulate how I might find the number of regions in the image. When I finally found a potential solution, I opened up IntelliJ IDEA and got cracking.

As with previous solutions, I created a 2d Boolean array to indicate whether I had already visited the pixel in question. I then iterated through the array in my usual fashion and only continued if I came across a black pixel that hadn’t be marked. If this was the case, I would then proceed to mark the pixel and call the recursive function MarkAllBlackNeighbours.

The feature does exactly what it sounds like it does – checks the neighbours of a black pixel for neighbouring black pixels, marks black neighbours, and then calls itself on said neighbour. The logic was that for every unmarked black pixel I came across, I would exhaust all its neighbours recursively and mark each one so that the make loop could not begin the process again. Therefore, I could safely assume that the unmarked black pixels in the main loop were the beginning of a new region, since doing so would mark their black neighbours and so on.

## Section 2.18 – nr\_eyes, hollowness and (custom) image\_fill

Despite completing the previous feature relatively easily, I couldn’t find the same success for this feature. Only by asking the Computer Science group on Facebook did I get the answer I needed (big thanks to Stephen McVeigh), which was the count the number of *white* regions in the image. So simple, yet I couldn’t think of it on my own. However, upon closer analysis of the idea, I noticed several pitfalls with the function that had to be accounted for:

* The algorithm would have to count the white space outside of the image, so the actual number of eyes in the image would always be 1 less than what the algorithm calculated.
* As per the assignment pdf, two neighbouring white pixels might not be in the same eye, so I would need to check for black pixels.
* It would also be smart to keep count of the number of white pixels in each eye for the hollowness feature, so I would have to adjust the algorithm for this.

Like the previous feature, I created a 2d Boolean array to mark white pixels which had already been visited. I then created an array list to keep count the number of eyes and the number of pixels in each eye. The function then loops through the pixels and checks for white, unmarked pixels, marks them and adds the function return of MarkEyes to the list.

Mark Eyes has a number of changes on the Mark Neighbours from which it was adapted. It has an int size, which is initially set to 1 to indicate the pixel from the main loop beginning the region. It then checks its neighbours for white, unmarked pixels, and does either one of two things if a candidate if found:

* If the neighbour is diagonal, then the two non-diagonal neighbours cannot both be black. If that is the case then the white diagonal neighbour is marked and the function is called again, with the result being added to the size.
* The neighbour is non-diagonal, so it is marked and the function is called again, with the result being added to the size.

This would return the pixel size of each eye to the array list. Now I can assume that the first white pixel belongs to the white space around the image, so I stored the value of the first element before removing it. The number of eyes then was the size of the list, and to calculate the hollowness, the values for each element were added and then used to divide the number of black pixels in the image.

The image\_fill feature was calculated using the pixel count of the image (white pixels in eyes and black pixels) divided by the pixel count of the image.

# Section 3 – 2000 words

## Section 3.1

# Conclusions