CMP2090M

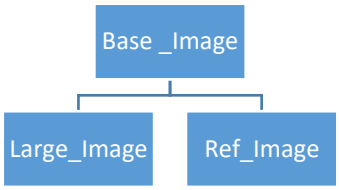
OBJECT-ORIENTED PROGRAMMING ASSIGNMENT REPORT

Gabriella Di Gregorio, [Student ID: DIG15624188]

1. INTRODUCTION

I was tasked with writing a C++ program which would essentially play a game of "Where's Wally". This involved reading greyscale values of two images from text files, one of a cluttered scene and one of Wally which was not an identical match to the Wally located in the scene. Since images are 2D matrices, the task also entailed putting this data into a 2D format. Once the 2D matrices are in place, the program must then compare them to find the best match of Wally within the large matrix. This required using a comparison algorithm to perform a nearest-neighbour search since the match would not be perfect due to the data being unidentical. Once the best results have been found, the program should then display a ranked list of best matches to the user as well as displaying the results on a picture which the program writes to a PGM file.

2. PROGRAMME STRUCTURE



To avoid unnecessary complication and spreading code too thinly, I decided to keep my class structure relatively simple. I implemented an inheritance hierarchy in which two classes, one for each image/matrix, inherit from a base class which contains all the functions that can be used by both image classes. Since they are both images, this means that they can share many methods and features due to having the same properties, just with different specifications i.e. their size and data. This class structure improves the organisation and efficiency of the code since it means pieces of code can be re-used rather than re-written for each image.

The base class contains the majority of the functions since the majority of them can be used for both images. The methods created almost represent the steps taken to carry out this task which I believe makes the code easy to read and follow for a different programmer. As the first step was to read in the data from the text files, this is the first method in the base class. This function is then called in both image classes, passing their specific parameters such as the file name and size. These then get used in the main when instances of the image classes are created. The base class also contains a method to put the data into the 2D matrix format with a 2D Vector. I decided to use vectors because there are many advantages of using vectors over arrays. For example, vectors are resizable unlike arrays which means that the size does not need to be specified in the function used to create a vector and it can then be used to create vectors of different sizes when the function is called and the size parameter is passed. There are many other relevant advantages such as being automatically re-allocated and de-allocated, the size being determined in O(1) time, and are able to be copied or assigned directly (GeeksForGeeks.org). This function to create a vector is also called in both image classes which creates a 2D vector for each matrix. However, the next function in the base class creates another vector for a section of the large matrix the same size as the small matrix so this is only called in the main rather than the other two classes. Other functions in the base class are used for calculating Normalised Cross Correlation which will be discussed further in the following section. Most of these functions are called in the main, however there are some calculations that will remain constant for the small image but change for the large image, so they are called in the Ref_Image class but not in the Large_Image class. The final step of the task was to display the results on an image in PGM format, so the base class also contains a function to do this. However, since it is only the large image being written, it is called directly in the main and the code to create a border around the results is also in the main. While the Ref_Image class does not contain

any functions, the Large_Image class has a function to convert the large 2D vector to a 1D vector so that it can be used to write the PGM.

3. NNS ALGORITHM

The Nearest Neighbour search, also known as the K-nearest neighbours algorithm, is a form of proximity search which finds the point in a given set that is closest (or most similar) to a given point (Wikipedia.org). In this case, a reference picture of Wally is being used to find the most similar area in the cluttered scene. Since this is based on feature similarity, a measure of similarity is needed. I decided to use Normalized Cross-Correlation (NCC) for my similarity measure which is a popular choice for use with template matching algorithms such as this. There was no particular benefit of choosing NCC over the Sum of Squared Differences (SSD) measure since a comparison done by University Malaysia Perlis showed that both techniques produced very similar results – they are both accurate however SSD is generally faster but NCC is more robust (M.B. Hisham 2015). I chose to use NCC because when looking at the maths involved, it was clear to see how the necessary steps could 'translate' into code.

Pseudocode showing how each part of the NCC calculation is implemented:

1. Calculating the mean value within a matrix:

```
CalculateMean (paramaters: Vector, SizeRows, SizeCols)
{
    Initialise counter to 0
    Initialise mean to 0

    FOR all the rows
            FOR all the columns
                    Mean = sum of current value in vector
                    Increment counter
    Mean / counter
    Return mean
}
```

2. Subtract the mean from all the values in the matrix:

```
SubtractMean (parameters: vector, SizeRows, SizeCols, mean)
{
    FOR all the rows
            FOR all the columns
                    Vector value = vector value - mean
}
```

3. Compute final NCC result:

```
NCCcalculation (parameters: WallyVector, comparisonVector, SizeRows, SizeCols)
{
    Initialise result to 0
    Initialise WallyTimesScene to 0
    Initialise WallySqaured to 0
    Initialise SceneSquared to 0

    FOR all the rows
            FOR all the columns
                    WallyTimesScene = value in WallyVector * value in comparisonVector
                    WallySquared = value in WallyVector * value in WallyVector
                    SceneSquared = value in comparisonVector * value in comparisonVector
    Return result
}
```

The functions above are then used in the main for each new section of the large image stored in a smaller vector to use for comparison…

```
FOR every 18 rows in SceneVector
    FOR every 24 columns in SceneVector
            Initialise comparison vector
```

```
Call function to populate comparison vector
Call mean calculation function
Call SubtractMean function
Result = call NCCcalculation function

IF result > 0
       Put into a vector to store the results

Sort results from largest to smallest
```

4. RESULTS

My program allows the user to input how many results they would like to see. A ranked list is then displayed of the best matches found, with the best being at the top. Depending on how many results the user has asked for, this amount will be indicated on the scene in the PGM picture by boxes around the match. I am very satisfied with the results of my program since it seems to be very accurate.

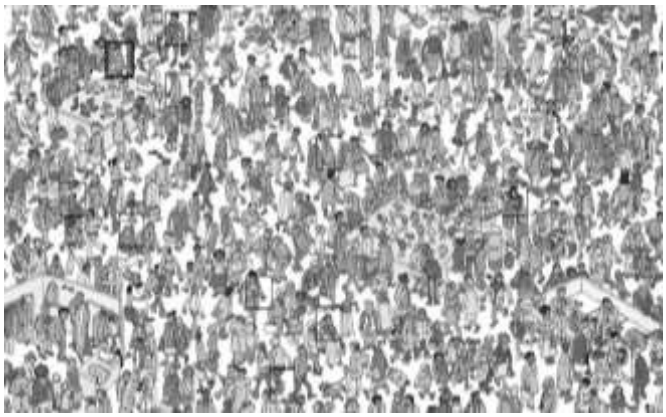
4.1 Best matching

The highest NCC result (closest to 1) was 0.52 and this was found at position row: 144, column: 162. When this is displayed on the image, I can confirm that this is Wally exactly! The whole of Wally is included in the identified section (no body parts have been cut off!) but neither is there lots of irrelevant space around Wally. This means that the implemented algorithm is very accurate and effective since the section identified is perfect.

```
How many results would you like to see?: 1

 OK! These are the 1 best results found...
1. Row: 144, Column: 162, NCC Result: 0.520032

Wally is most likely to be at row 144, column 162 because this scored the highest NCC value of 0.520032
```

4.2 N-best list

```
How many results would you like to see?: 10

 OK! These are the 10 best results found...
1. Row: 144, Column: 162, NCC Result: 0.520032
2. Row: 192, Column: 252, NCC Result: 0.46672
3. Row: 552, Column: 486, NCC Result: 0.410969
4. Row: 360, Column: 90, NCC Result: 0.370572
5. Row: 408, Column: 180, NCC Result: 0.368198
6. Row: 96, Column: 252, NCC Result: 0.36538
7. Row: 288, Column: 486, NCC Result: 0.35273
8. Row: 648, Column: 414, NCC Result: 0.348145
9. Row: 504, Column: 378, NCC Result: 0.341444
10. Row: 360, Column: 774, NCC Result: 0.340508
```



The program also successfully identifies the N-best results depending on what number the user inputs into the console. Wally is clearly identified with a thick border whilst the other N-best matches are indicated with a thinner border to avoid confusion. I wanted to indicate the positions with a border rather than them being entirely covered by a black box so that the user can see what is located in the identified positions. Whilst it is hard to judge how similar the other results are by simply looking at the picture, the NCC results listed are definitely in the correct order, are all less than 0.52, and seem realistically spread apart. Without doing the math by hand to check if they are correct, I am confident and content with the results of my program and I do not think that this could or should be improved.

5. DISCUSSION & CONCLUSION

As stated above, I am pleased with the results my program produces and I believe that it is as accurate as it possibly could be while maintaining the current level of efficiency. Whilst there is always ways to make

programs more efficient, I am also satisfied with the speed of my program. On my high-spec PC, it takes about 12 seconds to compare the matrices (run the NNS algorithm) so it seems unlikely that it would ever usually take any more than 20 seconds on any computer. The main way I achieved this efficiency was by jumping through the large image 18 rows and 24 columns at a time to select a section to compare, rather than cycling through every value one at a time. This may mean some of N-best results are not as accurate as they could be but since the best result is perfectly accurate and the program is efficient, I believe it has the right balance between efficiency and accuracy.

On the one hand, I believe I have successfully applied concepts of advanced software development and programming methods to solve a computational problem, however, on the other hand, the code is kept relatively simple since advanced techniques and principles have not been widely used. For example, all of the class members are currently set to public. While this ensured access from all parts of the solution, it also meant that careful management was required to avoid 'breaking' the member elsewhere in the code. Further development on this project would definitely involve controlling accessibility – members not needed elsewhere in the solution should be set to at least protected, if not private. Another advanced feature not currently present in the solution is operator overloading. "Operator overloading is a specific case of polymorphism in which some or all operators are treated as polymorphic functions and as such have different behaviours depending on the types of its arguments" (Wikibooks.org). While "you can write any C++ program without the knowledge of operator overloading, it is profoundly used by programmers to make programs intuitive" (programiz.com). For this reason, further development of the solution would definitely involve incorporating operator overloading in order to improve the quality of my code.

As a new programmer, I found this task rather difficult overall. The first challenge was learning how to properly use effective class structures and inheritance. While this then gave me an improved understanding of C++, I was presented with many more hurdles to overcome. My original plan was to use 2D arrays to store data but after discovering the limitations and difficulties of implementing the solution in this way, I learned how to use STL Vectors instead. Although vectors are a more advanced feature of C++, they were far easier to use for this task. My final challenge was to 'translate' mathematical formulas into code – creating the functions to carry out the NCC calculation was a large and difficult portion of this task. To conclude, the results of my program are as expected and so the task was completed successfully, however, the solution certainly has room for improvement since advanced object-oriented features could be implemented.

REFERENCES

GeeksForGeeks.org | Advantages of vector over array in C++ [online] | Available at: https://www.geeksforgeeks.org/advantages-of-vector-over-array-in-c/ [accessed 19/12/2018]

Wikipedia.org | Nearest Neighbor Search [online] | Available at: https://en.wikipedia.org/wiki/Nearest_neighbor_search [Accessed 19/12/2018]

M. B. Hisham, Shahrul Nizam Yaakob, Raof R. A. A, A.B A. Nazren, N.M.Wafi | 2015 IEEE Student Conference on Research and Development (SCOReD) | Template Matching Using Sum of Squared Difference and Normalized Cross Correlation

Wikibooks.org | C++ Programming/Operators/Operator Overloading [online] | Available at: https://en.wikibooks.org/wiki/C%2B%2B_Programming/Operators/Operator_Overloading [Accessed 19/12/2018]

Programiz.com | C++ Operator Overloading [online] | Available at: https://www.programiz.com/cpp-programming/operator-overloading [Accessed 19/12/2018]