EE 355 OCR PA – You Better Recognize

1 Introduction
In this programming assignment you will write a program to take a black and white (8-bit grayscale) image containing the bitmap representations of a set of decimal digits 0-9 and attempt to recognize/classify each character. This process is similar to what is known as OCR (Optical Character Recognition) where a document is scanned and rather than be converted to an image, is converted to text (ASCII) representation for future editing. Rather than recognize the whole alphabet, we will restrict our characters to decimal digits using a sans-serif (plain) font such as Arial. You will need to read the provided binary file, find the characters, perform certain classification tests on those characters, classify each one, and then output the corresponding text characters.

This programming assignment should be performed INDIVIDUALLY. You may re-use portions of code from previous examples and labs provided in this class, but any other copying of code is prohibited. We will use software tools that check all students’ code and known Internet sources to find hidden similarities.

2 What you will learn
After completing this programming assignment you will:
1. Design a solution utilizing classes and basic object-oriented principles
2. Use STL classes: vector and deque as container classes in your solution
3. Experiment with specific classification techniques from the image processing domain

3 Background Information and Notes
BMP File Format: All our images will be 256x256, 8-bit grayscale images. Thus, in memory we want to store our image data in a 256x256 2D array. We will use BMP files and can utilize the library functions written in class. In particular, the prototype for the BMP file read and write functions are shown below.

```c
int readGSBMP(const char* filename, unsigned char image[][SIZE]);
int writeGSBMP(const char* filename, unsigned char outputImage[][SIZE]);
int showGSBMP(unsigned char outputImage[][SIZE]);
```

Viewing Image & pixel data: To view these BMP images that we provide you can SFTP them down to your local machine and then use ImageJ or DIMIN, each of which is a free image viewer application (http://rsbweb.nih.gov/ij/ or http://www.dimin.net/software/viewer/). You can cursor over individual pixels and view their pixel data values.
However, using your Linux VM, you can get a similar program, 'geeqie', by running the following command and answering yes to all questions.

```
$ sudo apt-get install geeqie
```

Run this program to view .BMP graphics by typing:

```
$ geeqie ocr1.bmp & <or any other image file>
```

Using such a program, open up the input file: ocr1.bmp. Use the Dropper (Pixel Color) tool in the bottom right corner of the window. Positioning the cursor arrow over any pixel and the bottom status bar will show you the row, column coordinates as well as the RGB value of a pixel on the status bar. Note: In 'geeqie' they show the column coordinate first, then row.

**Grouping pixels into characters:** One of the first tasks of your program is to group connected black pixels into a character region (defined by a bounding box). In the image below we may find the black pixel at 1,3 first. At the point we should group any black pixels N,S,E,W,NW,SW,SE,NE of that starting pixel (known as the 8-connected neighbors) and repeat until all connected black pixels have been identified (i.e. perform a BFS search). As we find these pixels we can find the bounding box by finding the minimum and maximum row and column. A bounding box can be defined by the minimum row and column (i.e. 1,1) and the maximum row and column (i.e. 4,3). We could store these points or just the upper-left point (i.e. 1,1) along with a height and width (i.e. 4-1,3-1 = height of 3, width of 2).

![Bounding box example](image)

**Classification Techniques:** To take a collection of independent pixels and classify them as a single character we will need methods to identify and differentiate one from another using certain characteristics or inherent properties. On the spectrum of possible classification techniques is some kind of template matching where we take a known bitmap of a character and compare it. However, this does not account for noise in the scanning, font differences, etc. and thus is usually a poor choice by itself. Better classification techniques will be invariant to (will work even in the presence of) rotation, sizing, stretching, etc. However, to keep things simple we will make some assumptions:

- Characters will always be upright (non-rotated) and not unfairly stretched
- Characters may appear anywhere in the image but will always be at least 2 rows and 2 columns in from the boundaries of the image.
To help classify your characters, assume we find each character and draw a bounding box around it. Then we perform a set of tests on the portion of the main image inside the bounding box region. Some useful tests will include:

- **Euler Number**: # of unique components to the character (always 1 for our characters) minus # of (enclosed) holes in the character.
- **Aspect Ratio**: Height of the character divided by the width of the character.
- **Vertical centroid** (1st moment): Intuitively measures the vertical center of mass:
  \[
  \left( \frac{\sum_{i,j=0}^{H,W} i \cdot \text{isblack}(i, j)}{\sum_{i,j=0}^{H,W} \text{isblack}(i, j)} \right),
  \]
  where \( \text{isblack}(i,j) = 1 \) if the pixel at \( i,j \) is black, 0 otherwise.
- **Horizontal centroid** (1st moment): Intuitively measures the horizontal center of mass:
  \[
  \left( \frac{\sum_{i,j=0}^{H,W} j \cdot \text{isblack}(i, j)}{\sum_{i,j=0}^{H,W} \text{isblack}(i, j)} \right),
  \]
  where \( \text{isblack}(i,j) = 1 \) if the pixel at \( i,j \) is black, 0 otherwise.
- **Vertical Symmetry**: A measurement between 0 and 1 that indicates whether pixels in the upper half are a mirror image to the lower half (e.g. if a pixel in row 0, column c matches the one in row N, column c; if a pixel in row 1, column c matches a pixel in row N-1, column c). 1 indicates a perfect match of the upper and lower half.
- **Horizontal Symmetry**: A measurement between 0 and 1 that indicates whether pixels in the left half are a mirror image to the right half (e.g. if a pixel in row r, column 0 matches the one in row r, column N; if a pixel in row r, column 1 matches a pixel in row r, column N-1). 1 indicates a perfect match of the left and right half.

**Computing the Euler Number using Bit Quads**: Identification of optical characters and images is often facilitated by computing important geometric and topological properties. These geometric properties, which include area, perimeter and the Euler number, can be calculated by using Bit Quad Pattern. **Bit quad counting provides a very simple means of determining primary attributes such as, the Euler number, area and perimeter of an image. Hence, with the establishment of these primary characters, other geometric attributes can be developed.**

A *bit quad* is a 2 X 2 array of bit cells. Bit cells, which are squares, having the values 0 and 1, are the main composition of binary images. A systematic permutation of these bit cells could result in any binary character or image. There are 16 possible bit quads, and six distinct types considering rotational equivalence: \( Q_i, 0 \leq i \leq 4 \).

\[
\begin{align*}
Q0 (0 \ \text{black pixels}): & \quad \begin{array}{cc}
W & W \\
W & W
\end{array} \\
Q1 (1 \ \text{black pixel}): & \quad \begin{array}{cc}
B & W \\
W & W
\end{array} \\
Q4 (4 \ \text{black pixels}): & \quad \begin{array}{cc}
B & B \\
B & B
\end{array}
\end{align*}
\]
To calculate the Euler number of a (sub-)image, count the number of each class of bit quads (for each pixel assume it is the top-left pixel in the 2x2 quad and see which class it and its 3 other neighbors fall into. Once the number of each class of bit quads has been calculated we can find the Euler number using the simple formula:

\[ E = \left( \frac{1}{4} \right) \left[ n\{Q1\} - n\{Q3\} - 2 \times n\{Qd\} \right] \]

where \( n\{Qj\} \) is the count of the number of matches of that class of bit-quad.

We can also compute the area (exactly) and perimeter (approx.) of an object using bit quads:

\[ A = \frac{1}{4} \left( n\{Q1\} + 2 \times n\{Q2\} + 2 \times n\{Qd\} + 3 \times n\{Q3\} + 4 \times n\{Q4\} \right) \]

\[ P = \left( n\{Q1\} + n\{Q2\} + 2 \times n\{Qd\} + n\{Q3\} \right) \]

**Example 1:** The image below would have an Euler number of 1 (1 component – 0 holes)

<table>
<thead>
<tr>
<th>255</th>
<th>255</th>
<th>255</th>
<th>255</th>
</tr>
</thead>
<tbody>
<tr>
<td>255</td>
<td>0</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>255</td>
<td>255</td>
<td>255</td>
<td>255</td>
</tr>
</tbody>
</table>

\[ n\{Q1\} = 4, \ n\{Q2\} = 4, \ n\{Q3\} = 0, \]
\[ n\{Q4\} = 1, \ n\{QD\} = 0; \]
\[ E = (1/4)*(4-0-2(0)) = 1; \]
\[ A = (1/4)*(4+8+0+0+4)=4 \]
\[ P = (4+4+0+0)=8 \]

**Example 2:** The image below would have an Euler number of 0 (1 component – 1 hole)

<table>
<thead>
<tr>
<th>255</th>
<th>255</th>
<th>255</th>
<th>255</th>
</tr>
</thead>
<tbody>
<tr>
<td>255</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>255</td>
<td>255</td>
<td>255</td>
<td>255</td>
</tr>
</tbody>
</table>

\[ n\{Q1\} = 4, \ n\{Q3\} = 4, \ n\{QD\} = 0; \]
\[ E = (1/4)*(-4-4-2(0)) = 0; \]
\[ P = (4+20+0+4)=28 \]
Important Note: When calculating bit quads you need to make sure you start on a blank row and column and end on a blank row and column. Thus rather than just iterating over the bounding box region, which likely is the row and column where black pixels exist we need to iterate over the bounding box expanded in each direction by 1 row and column. In the images we give you we will guarantee that these rows and column are white. Thus, if your bounding box has an upper-left point at $(sr, sc)$ and height and width of $h$ and $w$ respectively, you should count bit quads over the region using the following loops to iterate:

```c
    for(i=sr-1; i <= sr+height + 1; i++){
        for(j=sc-1; i <= sc + width + 1; j++){
            ...
        }
    }
```

4 Requirements

Your program shall meet the following requirements for features and approach:

1. Your file I/O should read in the raw image data into a 2D array. It should convert the grayscale info to either black $= 0$ or white $= 255$.
2. Define a Character class to store information regarding a character in the object. It will implement functions to perform classification tests and store the results as well as other functions to print and return information to the main program. The Character class must contain the following public and private data and function members and be defined in ‘character.h’ and ‘character.cpp’:

<table>
<thead>
<tr>
<th>Public Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character( unsigned char (*myimage)[256], int ulr, int ulc, int h, int w)</td>
</tr>
<tr>
<td>Constructor to initialize the character object.</td>
</tr>
<tr>
<td>• myimage is a pointer to the 2D image array so that each character can access any of the pixel data from the image</td>
</tr>
<tr>
<td>• ulr and ulc represent the upper left row/column indices of the bounding box around the character.</td>
</tr>
<tr>
<td>• h and w are the height (number of rows) and width (number of columns of the bounding box around the character</td>
</tr>
<tr>
<td>void perform_tests()</td>
</tr>
<tr>
<td>Calls any private member functions to perform desired classification tests. Should store the results of each test in appropriate private data members.</td>
</tr>
<tr>
<td>void classify()</td>
</tr>
<tr>
<td>Uses the results of the classification tests by examining the test data to determine what character the pixels represent. This should determine the character and set it as a private data member</td>
</tr>
<tr>
<td>char get_classification()</td>
</tr>
<tr>
<td>Returns the private data member storing the character that this object was classified as</td>
</tr>
</tbody>
</table>
### void get_bounding_box(int &sr, int &sc, int &er, int &ec)

Returns two r,c points representing the bounding box of the character (top-left point and bottom-right point)

- sr, sc, er, ec are references to arguments from the caller that should be filled in with the start (upper-leftmost) row and column and end (bottom-rightmost) row and column

### void print_calculations()

Prints the results of the classification tests to the screen for debug purposes

#### Private Data Members

(You may add any other functions or data members of your choice).

```cpp
int ul_r, ul_c; // Upper-left most row, column
int height, width; // height and width of character
int euler; // Euler number
... // Add more data members as needed
```

- char recognized_char; // Recognized character
- unsigned char (*image)[256]; // Pointer to 2D image array

---

3. Write a main() routine in ‘test_ocr.cpp’ that performs the following:

   a. Accepts two command line arguments
      i. The first argument should be a string containing the filename of the image to perform character recognition
      ii. The second argument should be 1 or 0 with 1 indicating that debug information [i.e. the results of each test on each character should be printed to the screen]

      **Sample debug data:** Your output can be different based on what tests you implement and how you do it. But you should show the bounding box and any test results.

      ```
      Character in bounding box: 102,179 to 145,212
      ===============================
      BQ0=1008 BQ1=44 BQ2=112 BQ3=40 BQ4=620 BQD=0
      Euler number=1
      VCentroid=17.2469 / HCentroid=17.2162
      Shifted VCentroid=-4.25314 / Shifted HCentroid=0.716179
      VSym=0.504011 / HSym=0.407754
      Aspect ratio=1.30303
      Character 6 is classified as: 5
      ```

   b. Scan the image in row-major order (scan all columns of row 0 before moving on to row 1). Each time a black pixel that is part of a new character is encountered, find the bounding box and create a new Character object, and add it to a vector of Characters (or Character *’s). Be careful not to find the same character twice. Note by scanning in this order we will implicitly have the characters in order from top to bottom, left to right. In some cases a character on the right side of a
line of text will be slightly higher (start 1 row early) and thus be found first. This is fine, but can be confusing when looking at your output. We might suggest printing the bounding box coordinates so you can relate your character output to the image.

c. Run classification tests on each character object
d. Use the results of the tests to classify each character as 0-9
e. Print each character’s classification in the order that they were found (i.e. top to bottom, left to right).
f. If the argv[2] command line option is ‘1’ then also print the results of the tests for each character.

5 Procedure
Perform the following.
1. Create a ‘ocr’ directory.
   $ mkdir ocr
   $ cd ocr
2. Copy the sample images down to your local directory:
   $ wget http://bits.usc.edu/files/ee355/ocr.tar
   $ tar xvf ocr.tar
   This will download 4 sample images:
   * bmplib.h & bmplib.cpp – BMP reader/writer library
   * point.h – model and X,Y point (use this only if you want to)
   * test_ocr.cpp – main program code
   * Makefile –
   * ocr1.bmp – Sample image 1
   * ocr2.bmp – Sample image 2
   * ocr3.bmp – Sample image 3
   * ocr4.bmp – Sample image 4

Checkpoint 1:
3. Read in the BMP file using the filename provided from the command line and convert the grayscale info to only black=0 or white=255 using some threshold value.
4. Write the code (function) to find the bounding box of a character when you encounter its first black pixel
5. As a simple test of your code draw a GRAY (color=128) box representing the bounding box around the character you found. After all characters are found, just show the image to the screen (with the showGSBMP(...) function) and quit.
6. Run your program
   $ make
   $ ./test_ocr ocr1.bmp

Final Product:
7. Write the Character class and member functions (character.h & character.cpp) to provide the indicate functionality including classification and related tests
8. Write the main application (e.g. test_ocr.cpp) that reads in the file, finds the characters and their bounding box, creates the character objects storing them in a vector and then calls the classification and output operations on each character.

9. Compile and build your program

10. Run your program with debug

   ```
   $ ./test_ocr ocr1.bmp 1
   ```

11. Perfect your classification parameters so that your scheme works as best as it can on the provided input images.

    Note: This assignment is open-ended...Correctly classifying each character in each image may or may not be possible with the same set of classification parameters. Do your best and be creative. A major portion of your grade will be based on your test/classify implementation and another portion on its actual ability to correctly identify characters.

12. The output of your program when DEBUG information is turned off should be all the digits on a single line in the order they were found, separated by spaces. For example, running your program on ocr1.bmp should yield the output:

   6 7 3 9 0 4 8 1 2 5

   ***Important***: Ensure that when the debug flag is turned off (i.e. we run your program as ./test_ocr ocr1.bmp 0) that the only output from your program is the actual classified characters (as the example shows above). Failure to do so will lead to MAJOR deductions since our checkers will not work.

13. Comment your code as you write your program documenting what abstract operation a certain for, while, or if statement is performing as well as other decisions you make along the way that feel particular to your approach.

14. 

6 Lab Report

1. Turn in a PDF (must be named report.pdf) report (in neat, computer generated form...no hand-written reports) that describes what tests you implemented and how you used the results to classify character, providing insight as to why you chose the parameters you did. Essentially this should summarize your classify() function and how you developed it.

2. Submit all your .h and .cpp files (as well as the Makefile and PDF report described just above) on our website. Specifically, we expect the following files: [character.h, character.cpp, test_ocr.cpp, report.pdf]. Note: We don’t expect you should make any changes to bmplib.h & bmplib.cpp, so you need not submit these.
<table>
<thead>
<tr>
<th>Item</th>
<th>Outcome</th>
<th>Score</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code broken into approp. files</td>
<td>Yes / No</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Compiles successfully</td>
<td>Yes / No</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Image I/O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Reads file correctly</td>
<td>Yes / No</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>• Converts to binary (255 or 0)</td>
<td>Yes / No</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>• Performs 8-connected neighbor search to find bounding box of the character</td>
<td>Yes / No</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Character class</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Implements bit quad computation correctly</td>
<td>Yes / No</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>• Computes Euler number</td>
<td>Yes / No</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>• Computes Aspect ratio</td>
<td>Yes / No</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>• Computes Vertical Centroid</td>
<td>Yes / No</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>• Compute Horizontal Centroid</td>
<td>Yes / No</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>• Classification function provides a mutually exclusive classification of the character</td>
<td>Yes / No</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>main(int argc, char *argv[])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Accepts filename of OCR image</td>
<td>Yes / No</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>• Maintains a vector of character objects</td>
<td>Yes / No</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>• Scans and finds characters in appropriate order</td>
<td>Yes / No</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>(top-most then left-most)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Checks for debug argument and prints debug information</td>
<td>Yes / No</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Performance</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Characters classified in ocr1.bmp [10,7,4]</td>
<td>___ / 10</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>• Characters classified in ocr2.bmp [10,7,4]</td>
<td>___ / 10</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>• Characters classified in ocr3.bmp</td>
<td>___ / 5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>• Characters classified in ocr4.bmp [10,7,4]</td>
<td>___ / 10</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>• Characters classified in instructor test 1 [8,5]</td>
<td>___ / 10</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>• Characters classified in instructor test 2 [8,5]</td>
<td>___ / 10</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Quality Lab Report Explanation</td>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

**SubTotal**: 40

**Late Deductions (-10 for one day, -20 for two days)**: -10 / -20

**Total**: 40

**Open Ended Comments:**