Optical Character Recognition (OCR)

Naomi Esserman, Lucas Lawrence, and Nathan Siegel

***Abstract*— Optical Character Recognition is a versatile technology that has been around for a long time and can be applied to many different fields. In general, it involves the identification of written characters, and this has many wide ranging uses that each have their own optimal implementations. Our approach attempts to offer some unique additions and changes to the common algorithms used in OCR programs. Near-perfect accuracy was achieved for the fonts and characters that are currently supported, but improving speed and supporting additional fonts and characters are two areas where our program can be improved.**

I. Introduction

A. History

Optical Character Recognition is the concept of reading in an image and converting it into an editable and searchable text file. The application of this algorithm is extremely useful in digitizing data for both convenience and safety. While this concept has been around for a long time it wasn't until 1951 when computers started emerging that David Shephard came up with the first algorithm that scanned in 23 letters of the alphabet and could interpret Morse code and read it out loud [1]. He named his device Gismo and that was the start of developments for faster and more efficient algorithms. IBM was the first to commercialize their OCR algorithm and exhibited in 1965 in the World Fair in New York [2]. Soon after technology emerged that allowed special characters to be read and the alphabet in multiple languages.

As computers started to emerge the need for OCR became more vital as a way to store files. For the purposes of our project we did not focus on the scanning in of a bitmap image but rather on the recognizing characters of already scanned images. While the technology originally started for government purposes, over the years the technology for OCR has greatly improved and the applications for it have greatly widened and extend to commercial use. Currently it can be used for hand held scanners and wands, recognizing handwritten letters, scanning documents, and many more uses [2].

B. Uses and Applications

Optical character recognition has improved tremendously over the past few years. Currently there are algorithms that are able to predict words and sentences based on previous characters. These algorithms find the probability of the next character or word in order to minimize the time and output more accurate predictions.

In the past, OCR has most commonly been used for tasks such as reading in PDFs or other documents to create modifiable files. Recent advances, however have allowed this technology to be applied to many other fields. Automatic Number Plate Recognition (ANPR), for example, has recently provided a new industry that OCR is key to [3]. Other significant recent applications include extracting business card information into a contact list [4], converting handwriting in real time to control a computer [5], and defeating CAPTCHA anti-bot systems [6]. OCR has also provided a basis for more advance technologies such as

C. Constraints

The main constraint of optical character recognition is that it has to work for multiple fonts as well as different sized fonts. For a program to be robust and effective, it should work on most fonts and have a minimal error rate. Another constraint is that the optical character recognition program needs to be fast and able to be completed in a reasonable amount of time.

II. Motivation

Optical Character Recognition has been around a long time, and it has many wide-ranging applications. There are also many different ways that all of the processes that make up an OCR program can be implemented. Both of these factors makes OCR a very versatile technology, and we were interested in choosing the best implementations for the constraints that we were dealing with. This also allowed us to come up with many ways that our program could be extended. Because OCR is applicable to so many things, it is an ever-changing field, and our goal was to create an implementation that could be accurate and could be scaled to larger projects.

III. Approach

Optical Character Recognition is a multi-step process. It includes preprocessing, processing of input characters, training and testing. The preprocessing that was utilized in this implementation consisted of a line thinning algorithm to help speed up later processes, and a segmentation algorithm that allowed for the program to identify the individual characters. Processing consisted of computing feature vectors that are to be used for training and testing.

A. Organization and Planning

To begin our planning we divided the work into pre-processing techniques and processing techniques. We spent the first half just focusing on pre-processing techniques including reading in the image, line thinning, and segmentation. Once we had those accurately working we were able to start processing the individual characters. We created feature vectors that included the amount of starting points and intersections for each character along with dividing the segmented character into 16 zones and finding the percentage of black pixels in each zone. Finally to predict the character we found the minimum distance between the feature vector with our inputted test characters found from training.

In terms of the division of work, Naomi worked on reading in images while Nathan and Lucas worked on various aspects of preprocessing techniques. For the second half Naomi and Lucas worked on the training and feature extraction algorithms while Nathan focused on testing and accuracy.

B. Line Thinning

For line thinning, we decided to use the Zhang-Suen line thinning algorithm. Given that the pixels are defined as:

P9	P2	P3
P8	P1	P4
P7	P6	P5

Table 1: Pixel definitions

Once you find a pixel that is black, the following criteria are checked in two passes:

First pass:

a) The black pixel has neighbors, meaning that the pixel is not on the edge.
b) The number of neighboring pixels that are black is greater than or equal to 2 and less than or equal to 6.
c) The number of transitions from a white to black pixel starting at P2, going in a circular direction to P2 is 1.
d) At least one of P2, P4, and P6 is white.
e) At least one of P4, P6, and P8 is white.

If all these conditions are met, then add this pixel to a list. After you iterate through this first pass of the image, set all pixels that met the conditions to white.

Second pass:

a) The black pixel has neighbors, meaning that the pixel is not on the edge.
b) The number of neighboring pixels that are black is greater than or equal to 2 and less than or equal to 6.
c) The number of transitions from a white to black pixel starting at P2, going in a circular direction to P2 is 1.
d) At least one of P2, P4, and P8 is white.
e) At least one of P2, P6, and P8 is white.

If all these conditions are met, then add this pixel to a list. After you iterate through this second pass of the image, set all pixels that met the conditions to white.

Keep repeating these two passes until no pixels are changed on either pass. Then the image is thinned.

C. Segmentation

Before any analysis and predictions can be done with the character, they must be separated and identified. This was done by creating segments that each contained a single character.

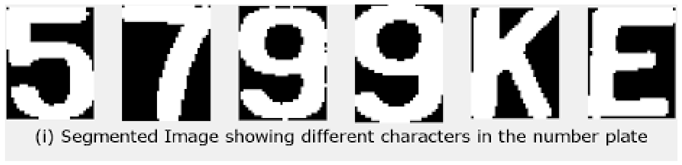
(i) Segmented Image showing different characters in the number plate

Figure 1: Example of segmented characters

https://encrypted-tbn1.gstatic.com/images?q=tbn:ANd9GcSrxtqNVEo3dpHI_mxuvSTU8OR-BxxflGma3_GqnRTiXhplzTp8CA

Because some characters can occupy overlapping regions of the horizontal axis, a method was employed that looked at grouped pixels to determine what was an individual character. A recursive function was written that looks for a black (assuming black characters) pixel as a starting point, and then finds all surrounding pixels. This is continued until all pixels that encompass a certain character. The pixels are "whited-out" after they are identified so that they are not found again. This requires a local constraint in that all of the pixels of each character are directly next to another one, but it allows for a character to be identified quickly without having to create arbitrary segments. The function finds the coordinates of each corner of the character, so the character is now completely defined, and this information can be used later when determining what character it is. This method is also very quick because our OCR implementation thins the image before it gets to this point.

D. Sorting of Segmentation Coordinates

The way that the BMP file is read in causes the pixels, and therefore the segmented characters, to be upside down. The segmentation method also is not guaranteed to have the characters in each row in the correct order, though the rows will be in the correct order after they are flipped (because they are read in upside down). The following implementation was created to sort each row while keeping the rows in the correct order:

```
for(i = 1; i < numCharactersCounted; i++)
{
   if(maxY[i] >= (currentYBottom+10) || maxY[i] <= (currentYBottom-10))
   {
      currentYBottom = maxY[i];
      bubbleSort(currentYBottomIndex, i-1);
      currentYBottomIndex = i;
   }
}
//for the last row
bubbleSort(currentYBottomIndex, numCharactersCounted-1);
}
```

Bubble sort was used because each of the lines will generally contain a fairly small number of characters, so a quadratic sort will be sufficiently fast for the general case, while avoiding the wort-case speed of other sorting algorithms [8].

Once the coordinates of each character have been sorted, these coordinates can be used to predict each character in the order that they are meant to be interpreted.

E. Feature Extraction

1. *Starting Points:* The first feature we get is the amount of starting points in a character. A starting point is defined as a black pixel that has one neighboring black pixel. If a black pixel is denoted by a 1, then if the sum of all the surrounding pixels is equal to 1 that pixel is considered a starting point.

2. *Intersections:* The second feature we extract is the amount of intersections in a character. An intersection is a black pixel that has more than 3 neighboring black pixels. To help define what an intersection is the neighboring pixels are put into one of two categories. The first being a direct pixel which is a pixel that is vertically or horizontally next to the pixel in question. In Table 1 these would be pixels P2, P4, P6, P8. The second being a diagonal pixel, which is a pixel that is diagonal of the pixel in question. In Table 1 these pixels would be P3, P5, P7, and P9. However, there are more criteria that must be met for the pixel to be considered an intersection.

   a. Three neighboring black pixels: the pixel in question is an intersections if none of the direct pixels are adjacent to the diagonal pixels
   b. Four neighboring black pixels: the pixel in question cannot be an intersection if each and every direct pixel is adjacent to a diagonal pixel.
   c. Five neighboring black pixels: the pixel in question is always an intersection

Due to the size we read fonts in, another criteria for a pixel to be considered an intersection is that it must be more than 2 pixels away from any other intersections that were already detected.

Once we have these two features, we can narrow down the possible characters that the current character can possibly be. This helps lower our chances of error since the set of possible characters is smaller. The table in the 'distance' section of the report shows what bins this puts the characters into.

3. *Percentage of pixel in a zone:* Our main method of determining which character the character in question is, was to divide the character into 16 zones and calculate the percentage of black pixels in each of these zones. For example, to calculate the number of pixels in one of the zones in the corner, the following for loop could be used:

```
short sizeOfZone = height / 4 * width / 4
short numPixels = 0;
for(i = 0; i < height/4; i++)
    for(j = 0; j < width/4; j++)
        {
            if(character[i * width + j])
                numPixels++;
        }
unsigned short zonePercentage = (unsigned
short)((numPixels/sizeOfZone) * 100);
```

The character was divided into 16 equal zones and percentages were calculated for each of these zones.

One issue that can arise from this is if the character is really thin, then the code cannot properly divide it into 16 zones. The one character this was an issue for was 'I'. To combat this, we had a if statement that checked if the character's width was less than a certain value, it had to be an 'I'.

F. Distance Vector Calculation

In order to match the correct character we used the feature vector found in the previous section and found the minimum root mean square of the error. We created bins of potential character guesses based on the number of starting points and intersections in each character.

Starting Points	Intersections	Potential Characters
0	0	0, O, D
0	2	B, 8
1	1	P, Q, 6, 9
2	0	C, G, I, J, L, N, S, U, V, Z, 1, 2, 5, 7, M, W
2	1	Q, 4
2	2	A, R, Q
3	1	E, F, M, T, W, Y, 1, 3, G, N, Z, 7, 5
4	1	X
4	2	K, H, X

Table 2: Potential Characters based on starting points and intersections

The formula for root mean square error is:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

The benefit of using the root mean square error is that it highly penalizes large errors. Within each bin we then found the root mean square of the vector using the current characters vector values and the training set character values in the current bin. The character that produced the smallest root mean square error is the character that we output. Some characters were put in multiple bins because they had potential options based on the different fonts and the way in which they were thinned.

IV. Results

A. Training

To train the program, bitmap files were input for each of the 36 characters that the program can currently work with. Each of these files contained 1 character each from the 3 font that are currently supported (Calibri, Arial, and Microsoft Sans Serif). These files were read in one at a time using a separate function for training. The results are currently hardcoded in the program so that the OCR testing function can use these values to determine what each character is without having to train it again each time that the program is run. The training function can be used later to get results for addition fonts and characters and the OCR program can easily be scaled to support these additions.

Figure 2: Examples of training inputs

B. Accuracy and Speed of Testing

Sentences of different lengths and containing all three fonts were used as test inputs, and the computation time was recorded to see the relationship between number of characters and the time that it took to process them.

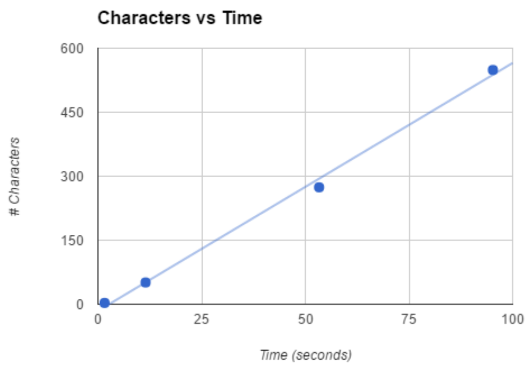

Figure 3: Graph of number of characters vs. computation time

The relationship is seen to be strongly linear. It can be seen that the program can process about 6 characters per second. It is important to note these times include reading in the file, thinning and segmenting, and actually computing the feature extraction vectors and guessing the correct character.

The other major benchmark was accuracy. Because this program is intended currently only to run on typed characters, it can be determined which characters will always be guessed by the program currently, and which will always result in errors.

Font	Number of Errors (Character)
Calibri	1 (O)
Arial	0
Microsoft Sans Serif	0

Table 3: Errors per font

It can be seen from the table that the program is 100% accurate for two of the three fonts that are currently supported. The other font still works with 35 of the 36 supported characters. The only character that is incorrectly identified is O (Oh), which is interpreted incorrectly by the program as 0 (zero).

C. Discussion

Some challenges we faced include reading in small characters, dealing with similar characters, achieving higher speed, and scaling to more fonts or including handwritten characters. Reading in small characters was a challenge because of the way we read in the image. The image would set each pixel to either a 1 or a 0 based on some threshold that took into account the RGB values. Because of this, smaller fonts would result in the character not being connected all the way through.

Dealing with similar characters such as 0 (zero) and O (Oh) is also difficult since we are only using a zoning algorithm. This algorithm produces very similar outputs for each character. A potential solution to this would be to either use hidden Markov models to further lower the amount of possible characters a character in question can be, or to extract more features.

Our current program is pretty slow, recognizing only six characters per second. One reason for our program being so slow is all the processing we do before we decide which character it is. We read in the image, thin the image and segment the image before we begin to extract features.

Lastly, increasing our training set to recognize more fonts is a challenge as well. As you add more fonts to the training set, more errors are introduced into the working set you have already. As far as handwritten fonts go, one issue that would arise is that you may have to level the image if the handwriting is not written in a straight line or written diagonally.

Overall, we were able to make our program very accurate. Granted, we were able to limit our inputs to just three different fonts and only capital letters but it is a good start. Our program is pretty slow, which is a result of all the overhead and pre-processing we do to recognize each character. One way to fix this would be to implement parallel processing to speed up our program. To expand on our program, we could add punctuation and more symbols, such as '#' or '*' or '!'. We could also add more fonts to our training set to get a more complete program.

REFERENCES