TJHSST Senior Research Project
Development of a Generic Font OCR
Third Quarter Research Paper
2007-2008

Nathan Harmata

April 3, 2008

Abstract

OCR (Optical Character Recognition) is a very practical field of Computer Science. Since the late 1980's, researchers have been developing systems to identify text from non electronic text sources, like pictures or papers. The use of OCR systems has spanned from making books in Braille to sorting mail by zip code by United States Post Office. This project describes the development of an entire OCR system in Java.

Keywords: OCR, Optical Character Recognition, Image Processing, Computer Vision

1 Introduction

The goal of this project is to create an application that can read text from electronic picture files. One of the main focuses will be developing a generic way to recognize characters of different fonts, rather than hardcoding in definitions for specific fonts. Although OCR is by no means a "new" field, it has still yet to be fully explored. Many common computer users either don't have access to an OCR program or don't know they have one, and some of the ones that are free of charge are lacking in performance and consistancy.

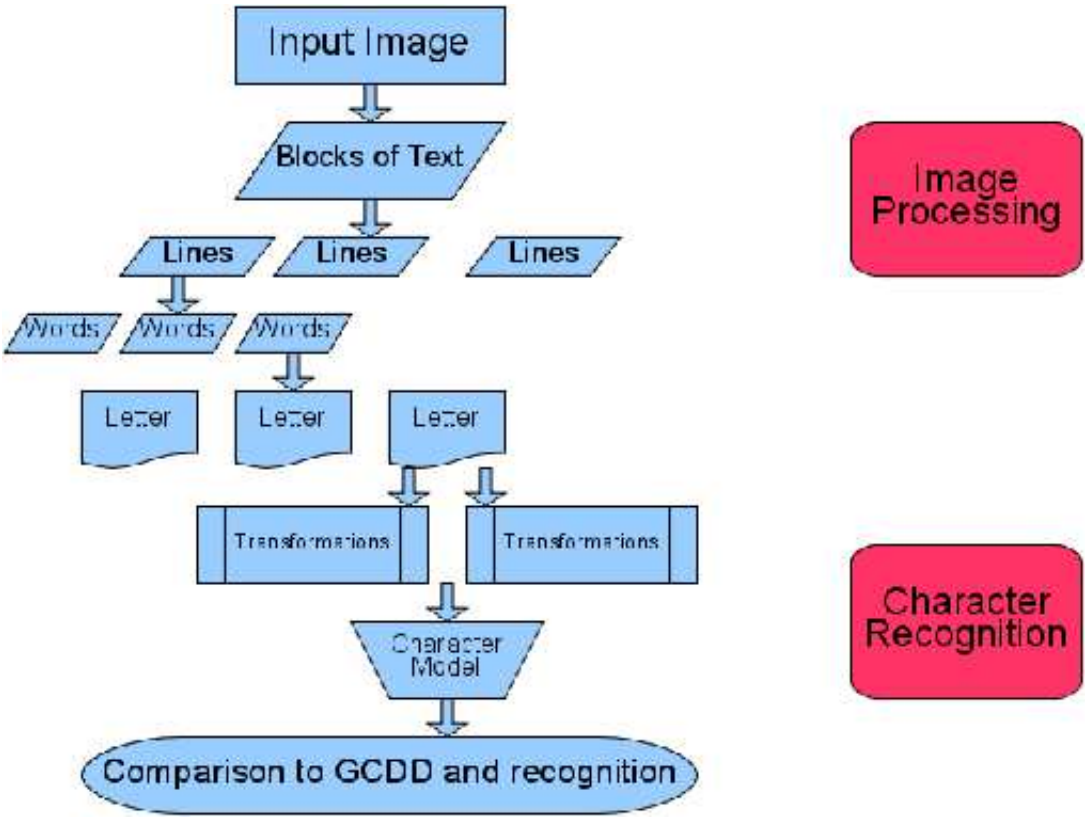

Figure 1: An overview of the OCR process.

2 Background

OCR systems have been around since the late 1980's. Still, they are not widely available or used by the public. The results from a review of the free ones on the Linux operating system are not very promising. [1] Although most of them had measured accuracies above 94 percent, that is not good enough. The one commercial product tested, Aspire OCR, was accurate only 91.5 percent of the time. The most likely industry standard, Tesseract, is also one of the oldest OCR systems. The review measured it to have an accuracy rate of 99 percent. Development on it started in 1985 and it is still used as the OCR engine for Ocropus, Google's textual analysis application. It is unlikely that this project will be able to achieve similar success, but the goal is have a working OCR system.

Figure 2: Parsing the word "JeFfErSoN" into its letters.

3 Procedures

The OCR system works by accepting an image of text, and, through a series of parsings and transformations, is a able to recognize a basic form on each character. There are two main steps in the OCR process, Image Processing and Character Recognition.

3.1 Image Processing

1. Blocks of Text

   The input image is not just going to be plain text in a nice and simple format. The objective is to get the text into such a form through a series of parsings. First, paragraphs are recognized and separated. Then, paragraphs are broken down into individual lines, and lines into words. This is all done on the premise that the image is of computer generated text. This ensures that there are straight lines of whitespace between adjacent paragraphs, lines, words and words. One key part of the text parsing process is that the optimal separations are made; paragraphs, lines, and words are "boxed" as closely as possible, to minimize errors.

2. Character Parsing

   Each word is then processed into its individual letters. Once again, the premise of the existence of whitespace between adjacent characters is used. Unfortunately, this isn't always the case. In some fonts, there are "ellisions" between certain letters. To get around this, these ellisions have to be recognized and properly handled. This is done by calculating the relative coverage of pixels between two areas that are expected to be separate characters. If this is small enough, the total width of the group of ellided characters is compared to the average width of other letters in the word. A group of ellided characters will be much wider than the average character. Images of individial characters are then reboxed to ensure that there is no extraneous whitespace.

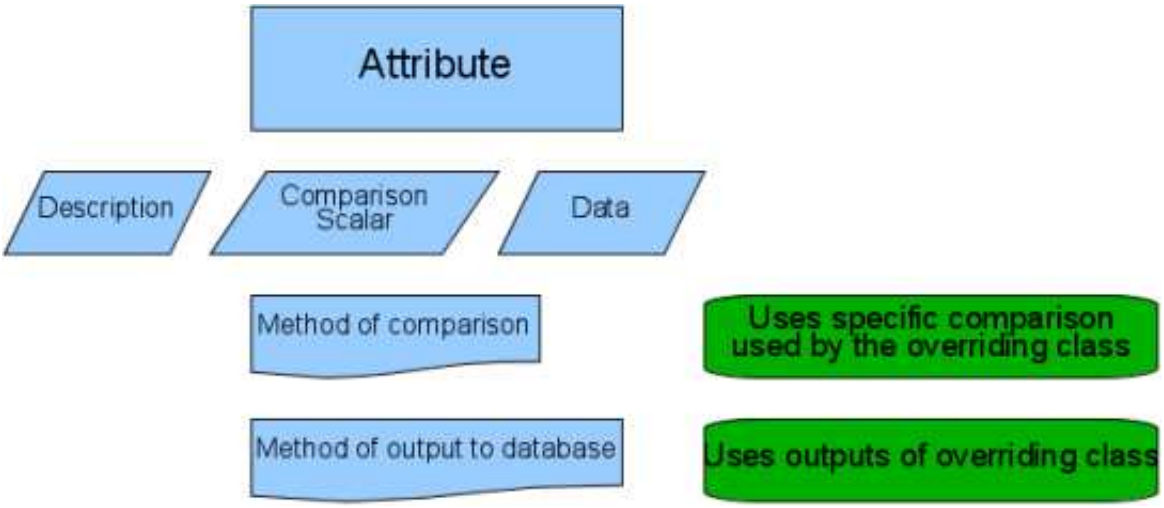

Figure 3: An overview of the Attribute class.

3.2 Character Recognition

Each character image then undergoes a series of transformations into what are called "Attributes." Attributes of the image are then combined into what is called a "Character Model." A Character Model serves as a generic character definition. That is, it defines what the character would be like irrespective of font. The purpose of this is that each Character Model can be compared to a pre-generated database of generic character recognitions for each possible character, and the best match can be found. So then, the steps through which each character image goes are as follows.

3.2.1 Attribute

An Attribute is one specific representation of the image and is used as a comparison heuristic. Each Attribute is able to compare itself to other Attributes of the same type and is able to output its relevant data. Each Attribute also knows what type of Attribute it is. This fact is very important and will be explained later. In the actual code of this project, there is an Attribute class which each specified characteristic extends. Currently, this project uses two Attributes, "Sector Vector" and "Gap Vector."

1. **Sector Vector**

   A Sector Vector consists of three pieces of data: the number of "sectors" in the image, the total number of line segments in the transformed form of the image, and the sign of the slope of the first such line segment. A sector is defined as a region of the image that passes the vertical-line test; that is, for each x-coordinate in the image, there is at most

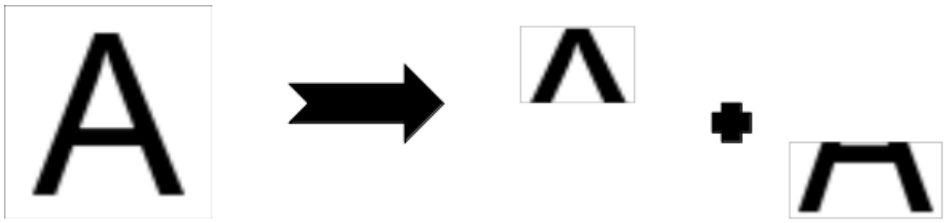

Figure 4: An example of Sector Parsing. The "A" is parsing into two sectors: the upper half and the lower half.

one y-value. The purpose of parsing the image into sectors is so each sector can be parsed into line segments. Since the line segment parsing is based on the slopes of adjacent pairs of pixels, sectors parsing is necessary. The algorithm of transforming an image into a Sector Vector consists of two steps, Sector Parsing and Slope Field Parsing.

(a) Sector Parsing

Starting from the top of the image, the program progesses downward until it reaches a point at which there is a conflict with the portion of the image already processed that would cause a failure of the vertical-line test. This point represents the end of one sector and the beginning of another. This process is repeated until the entire image is parsed in sectors. The result is that each sector is as large as possible, within the constraints. The important piece of data gained from this process is the number of sectors in the image. Each sector is then parsing into line segments.

(b) Slope Field Parsing

Starting with the left-most pixel in the sector, the program progresses to the right. At each x-coordinate, a line segment is constructed between the pixel at that x-coordinate and last pixel considered. If the slope of this line segment is similar, in sign and magnitude, to the slope of the previous line segment, then it is incorportated into the previous by changing the last pixel of the previous one to the last pixel of the current one. If the slope is radically different, then a new line segment is constructed. The results is that the image in transformed into a collection of line segments. There are two important pieces of data derived from this step: the number of line segments and the sign of the slope of the first line segment in this sector.

A sample sector vector is:

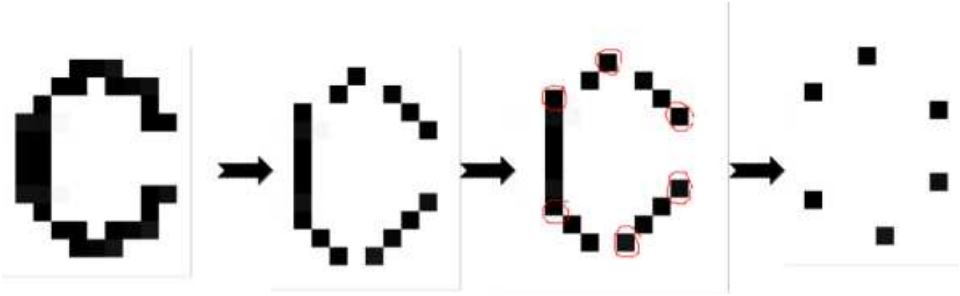

Figure 5: An example of parsing a "C" into line segments. The image on the right contains the pixels which, when connected, form the line segments representing this image. Note that the Sector Parsing of the "C" is not shown in this diagram.

```
-2 3
```

This means that the image contains two sectors, the sign of the slope of the first line segment in the first sector is negative, and there are a total of three line segments. This happens to be the Sector Vector representation of "C" for most fonts.

Interestesting, a similar method of segment parsing was independently developed by two researchers. It does not use a process similar to Sector Parsing; instead it parses the image into a set of predefined line segments. [2]

2. **Gap Vector**

   A Gap Vector is simply what, if any, "gaps" are present in the image. A gap is defined as a breakage of pixels on one of the four edges of the image: top, right, bottom, and/or left. The purpose of having such a comparison heuristic is the assertion that gaps are more representative of a character than line segment parsing. That is, same characters of different fonts are more likely to have the same gaps than they are to have the same Sector Vector. Also, Gap Vector provides information that is exclusive from the information given by Sector Vector. The presence of a gap isn't likely to have any correlation to the presence of sectors. Unlike sectors, the definition for a gap isn't simple. Both concepts were invented for the purpose of this project, as were their working definitions. This project defines a gap using the algorithm by which they are located.

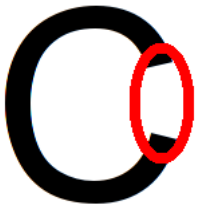

Figure 6: The portion of the image circled in red is a "gap" in the "C".

(a) Corning Finding

The first step in the gap finding process is to locate the four corners of the image. Each image is defined to have four corners; even if it is not visually obvious to a human that there are four corners, such as is the case with a letter like "O," four corners are forced on the image. There are four corners: top left, top right, bottom left, and bottom right. A corner is defined as the intersection of the path starting from one corresponding extremum of the image and the path starting from the other corresponding extremum of the image. For example, the top left corner is the intersection of the path starting from the bottom left extrememum of the image and the path starting from the top left extremum of the image. The corner is, of course, a single point. Therefore this intersection is the one that occurs when the path progress towards each other at the same rate. This process is repeated to find the locations of the four corners of the image.

(b) Path Tracing

The next step is to use the corners to determine if there are any gaps in the image. Between any two adjacent corners lies one of the sides of the image. Any gap on that side, by definition, must be between those corners. The algorithm uses this fact to its advantage. For each pair of adjacent corners, it iterates across the straight line between them. As each point on this line, it determines if the corresponding point in the actual image is, in respect to the side of the image on which the computation is occuring, in front of or behind of the line. For example, consider finding a gap on the left side of an image. On the left side, the slope of the path between the two corners forming that path, the top left and bottom left, is in respect to a vertical line. That means that, for a coordinate on the line (a) and a coordinate on the image (b), a comparison of the x-coordinates of those coordinates can be

Figure 7: An example of the process of finding the top left corner of "A". The image on the left shows the path from the bottom in red and the image on the right shows the path from the top in green.

used to determine the relative location of the coordinate on the image. Since the left side of the image is being considered, if the difference between them is positive, then the one on the image is behind the one on the line. That is a "is in front of" b if:

$$a_x - b_x > 0 \tag{1}$$

The sum of all the distances between the line and the corresponding points is calculated, keeping in mind whether the point was in front of or behind the line. The result is that if more of the pixels are behind the line, this sum is negative. That means that more of the image itself is behind the line, which implies a gap. This computation is actually simply comparing the area of the part of the image in front of the line with the area of the image behind the line, in respect to either a straight line or a horizontal line, whichever is more appropiate. Thus, a gap exists on a side if there are more pixels behind the line between the corners forming that side than in front of it.

The result is a list of gaps, represented by the strings, 'T', 'R', 'B', and 'L' for 'Top', 'Right', 'Bottom', and 'Left', respectively. A sample Gap Vector is:

```
R
```

The means that there is a gap on the right side of the image, such as is the case for a 'C'.

The importance of each Attribute knowing what type of Attribute (Sector Vector, Gap Vector) it is, called its "description", is so that different Attribute representations for an image can be easily grouped together. This grouping is called a "Character Model."

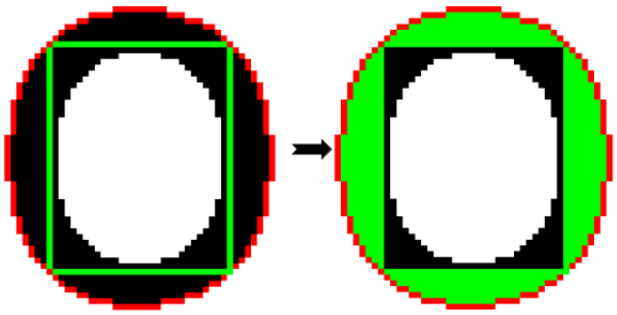

Figure 8: An example of the Path Tracing algorithm for "O". The pixels in red are the ones on the actual image. The green lines in the left image show the straight line paths between the adjacent corners. The areas in green in the right image show the portions of the image "in front of" their respective straight line paths. Since all of these areas are greater than the areas behind the paths, which happens to be 0 for "0", there are no gaps in the image.

3.2.2 Character Model

A Character Model contains a collection of Attributes and the processes to use those Attributes for character recognition purposes. The Attributes are stored in a HashMap based on the hashcode of their description; this is done so that they are always stored in the same order, making comparisions between like Attributes easier.

Comparisons are relatively simple; two Character Models are treated as vectors and the magnitude of the vector difference between then is calculated. The "elements" in the "vectors" are Attributes; therefore the definiton of a difference of Attributes is the result of the comparision between them defined by their Attribute class. Thus, a comparison between Character Models A and B is:

$$\sqrt{\sum_{i=1}^{n}(attribute_i A.compareTo(attribute_i B))} \tag{2}$$

A Character Model also has a method of outputting its important data. It does this by using the output of its Attributes and their respective descriptions. For example, the Character Model output for the "C" image, which has been used as an example, is:

```
SectorVector -2 3 GapVector R
```

Character Models are hashed based on the hashcode of this output string, much like the hashing process for Attributes. This is done to ensure that Character Models that are identical in respect to their data have the same hashcode.

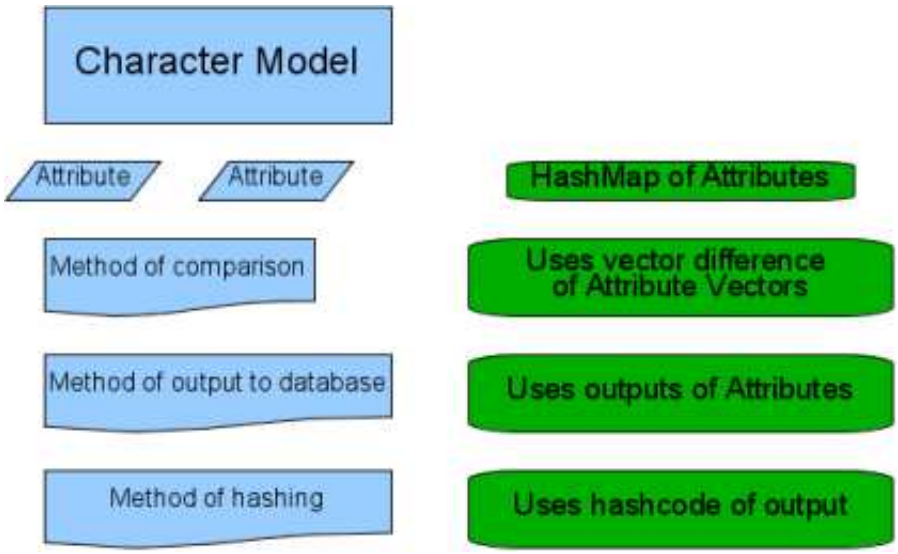

Figure 9: An overview of the Character Model class.

3.3 Generic Character Definition Database

The purpose of all these parsing and transformations is to get the input image into a generic form. This form can then be compared, using the methods of comparison already outlined, with a pregenerated database of generic forms. Such a database, called a Generic Character Definition Database (GCDD), is created by performing the analysis procedure on each character (letters, numbers, and other symbols) for several fonts and averaging the results, which are then outputted along with the characters they represent into a file. This database is handled by the system using the outputs of the Character Models in it. Because of this, it is important that the output includes the Attribute descriptions in addition to the Attributes data. The system can simply assume that the first token in the output is the name of a Attribute class. Using Java Reflections, it can then determine how many pieces of information are needed for that Attribute and assume that number of tokens immeadiatly following the first one to be the arguments for the constructor of such an Attribute.

The program then finds the three best matches for the image from the GCDD. Using a dictionary reference, all possible combinations of the letters forming words are considered. This has been shown to dramatically improve OCR accuracy. [5]

4 Testing and Results

There was not enough time to generate new results for the progess made third quarter. The second quarter results can be referenced if the reader desires to see them. There was enough time, however, to generate a working version of the GCDD:

```
a SectorVector -5 5 GapVector
b GapVector  SectorVector 4 3
c SectorVector -2 3 GapVector R
d SectorVector -1 3 GapVector
e SectorVector -2 3 GapVector
f SectorVector 0 3 GapVector R
g SectorVector -1 5 GapVector
h SectorVector 0 1 GapVector
i SectorVector 0 2 GapVector L
j SectorVector 0 4 GapVector
k SectorVector -2 3 GapVector R
l SectorVector 0 1 GapVector
m SectorVector -3 1 GapVector T
n SectorVector -1 1 GapVector
o SectorVector -3 3 GapVector
p GapVector  SectorVector 4 3
q SectorVector -1 3 GapVector
r SectorVector 0 1 GapVector R
s SectorVector -2 6 GapVector
t SectorVector 0 3 GapVector
u SectorVector 0 1 GapVector T
v GapVector T  SectorVector -2 1
w SectorVector -5 1 GapVector T
x SectorVector -4 3 GapVector T L
y SectorVector -2 3 GapVector T L
z SectorVector 1 4 GapVector L
```

4.1 Testing Programs

Throughout the development of the system, various programs were developed to help analyze intermediate results. The most important one of these

analyzed the working version of the GCDD to determine the following:

1. The deviation between a Character Model used to generate the GCDD and the contents of the GCDD, for each character in the GCDD.
2. How "close" members of the GCDD are to each other.
3. For each member of the GCDD, the actual characters who share that member.

Ideally, the value of the first relationship would be relatively small, and the value of the second relationship would be relatively large, and the value of the third relationship would be relatively large. This would mean that character definitions are more discrete; the results are more "spread out."

4.2 Resources

The following computer languages, algorithms and programs are being used, in addition to the ones already described.

1. The OCR system is written entirely in Java.
2. Java's ImageIO class is used for picture input and output.
3. Java's BufferedImage class is ued to handle pictures.
4. KolourPaint is being used to make picture files for input and to precisely view pictures.

5 Conclusions

A lot of progress has been made since the second iteration of the OCR system. The original version was based off of direct comparisons to a cache, meaning that only text of the font that was cached could be read. The current version, however, makes generic comparisons based off of a database of pregenerated definitions using the algorithms outlined in this paper.

A GUI will be developed so that the OCR system can actually be put to use. Users will be able to open a supported picture file and specify which portion(s) of the image they want to be "read." Methods might be implemented that can recognize the presence of text in images so that the user

doesn't have to manually select it. [6] The program will print the text to a text box from which it can by copied into a text editor or saved to a text file.

Further work will also have to be done to improve the current methods for the detection and removal of noise. There are various methods that can be used to accomplish this. [4] In addition, there might be a prelimineray check to ensure that the portion of the image selected is, in fact, text in the proper orientation. There have already been methods developed for this purpose. [3] Overall, the successes the current version has had shows that, with improvement, it will be a viable way to implement an OCR system.

References

[1] Austin Acton. A review of free optical character recognition software, 2007.

[2] Maher Ahmed and Rabab Kreidieh Ward. An expert system for genral symbol recognition, 1998.

[3] Hrishikesh B. Aradhye. A general method for determining up/down orientation of text in roman and non-roman scripts, 2004.

[4] Faisal Shafait, Joost van Beusekom, Daniel Keysers1, and Thomas M. Breuel. Page frame detection for marginal noise removal from scanned documents, 2007.

[5] Kazem Taghva, Julie Borsack, and Allen Condit. An expert system for automatically correcting ocr output, 1994.

[6] Victor Wu, R. Manmatha, and Edward M.Riseman. Finding text in images, 1997.