An Improved Segmentation Module for Identification of Handwritten Numerals

by

Jibu Punnoose

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

October 15, 1999

February 2000

Author________________

Department of Electrical Engineering and Computer Science
October 15, 1999

Certified by_______

Dr. Amar Gupta
Thesis Supervisor

Accepted by________________

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

An Improved Segmentation Module for Identification
of Handwritten Numerals
by
Jibu Punnoose

Submitted to the
Department of Electrical Engineering and Computer Science

October 15, 1999

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT:

The Winbank program is a segmentation-based recognition system for reading handwritten numeral on bank checks. An improved segmentation algorithm, Extended Drop Fall, has been incorporated into the system for proper segmentation of problematic cases. An improved method of choosing between the different segmentation algorithms has been introduced in order to utilize the Extended Drop Fall algorithm. The improved system relies less on heuristic analysis and can properly segment cases that do no match heuristic assumptions without compromising performance on more common cases. The changes significantly improve the performance of the segmentation module and as a result overall performance for the system is improved. Other modifications have also been incorporated into both the segmentation and recognition modules to increase performance and produce a more robust system

Thesis Supervisor: Dr. Amar Gupta
Title: Co-Director, Productivity From Information Technology (PROFIT) Initiative
Sloan School of Management

Acknowledgments

I would like to thank my parents, Chaluvelil M. Punnoose and Marykutty Punnoose, my sister, Judy Punnoose, and the rest of my family for constant support and encouragement.

I would also like to thank Dr. Amar Gupta for supervising the project and providing direction and guidance.

Finally, I would like to thank the members of the OCR group: Kirimania "Tim" Murithi, Alfred Mutiso, Timothy Chien, and Ishan Sachdev. Special thanks to Ishan for developing the testing program used to generate much of the results and statistics reported in this thesis.

LIST OF FIGURES

LIST OF TABLES

1. Introduction

A common goal in the field of artificial intelligence is to replicate the function of human beings and automate tasks that normally require manual labor or at least supervision. Optical Character Recognition (OCR) is a typical example of such a problem. OCR has been the subject of a large body of research because there are numerous commercial applications for this technology. The problem was once considered a simple and solvable subset of the larger area of pattern recognition. However, after some preliminary success in a few areas, the problem proved to be much more complex than initially imagined.

Character recognition can be divided into two categories: recognition of handwritten characters and recognition of typeset or machine printed characters. Machine printed characters are uniform in height, width, and pitch assuming the same font and size are used. Also, location of characters on the page is somewhat predictable. Recognition of machine printed characters is a relatively simpler problem and a number of low cost solutions exist. In fact, most commercial scanners are now packaged with OCR software that can read documents written in a number of standard fonts. [2]

Recognition of handwritten characters is a much more difficult problem. Characters are non-uniform and can vary greatly in size and style. Even characters written by the same person can vary considerably. In general, the location of the characters is not predictable, nor is the spacing between them. In an unconstrained system, characters may be written anywhere on the page and may be overlapped or disjoint. Furthermore, there is the possibility of ambiguous or illegible characters. Very little information about the characters is available in unconstrained recognition. A typical recognition system will require some sort of constraints, or added information, about the data being processed.

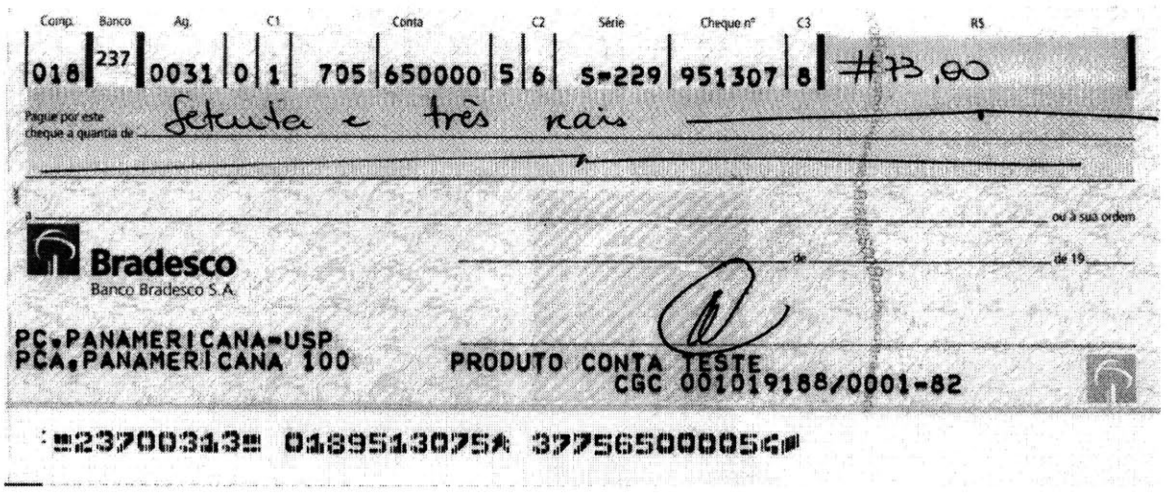
Recognition systems fall into two broad categories: on-line and off-line systems. An online system typically operates in real time and will recognize characters as they are being written down. At a minimum, the system will record pen stroke information in real time and analyze the information afterwards. Examples of online systems are the recognition engines used in PalmPilots and the Apple Newton. Offline systems do not have the benefit of this information and must recognize characters after they have been written. At present, only online systems can read completely unconstrained documents. Offline systems must impose some constraints to make recognition feasible. Many systems require that the characters be consistent in size and position by using preprinted boxes [2]. The software used by the US postal service to recognize zip codes exploits the fact that a zip code consists of exactly five or numeric characters, or nine in the case of extended zip codes [4]. In general, offline recognition is considered a much more difficult problem. [2,3,6,10,11,12,13]

An important application for offline handwriting recognition is the reading and verification of checks for processing in banks and other financial institutions. Verification must be done offline because it is not possible to obtain pen-stroke information from a paper check. In addition, the problem is nearly unconstrained; size and spacing can vary greatly between checks and even the exact location of the writing is not constant. However, by reading only the numerical value of the check, the scope of the recognition problem is reduced to a limited set of character (i.e. numeric characters, and delimiters such as $.) Even with this simplification, the problem is a difficult one; however, processing costs in the financial industry make it well worth the effort.

Banks and other financial institutions process over 66 billion checks a year in the United States alone [16]. The value of each of these checks must be entered manually by an operator. Because this manual entry is sensitive to error, checks are usually entered redundantly by more than one operator. Each operator requires approximately 5-10 seconds to process a check. Check verification is a labor-intensive process that costs billions of dollars a year. Automating this process can save a great deal of money and resources.

In order for a system to be commercially viable it must recognize a significant percentage of the checks processed with an extremely low error rate. It *is acceptable* to reject a large number of checks; even if half the checks processed are rejected as unreadable and must be verified manually, the costs have still been reduced by 50%. On the other hand, it is not acceptable to read even a small fraction of checks incorrectly. Checks that are read incorrectly can have disastrous consequences and must be kept within hundredths of a percent.

The paper will discuss modifications to improve the performance of the Winbank system presented in [1] and [2]. The system was designed for automated verification of Brazilian checks, using scanned-in images of the checks. All necessary information relating to the amount of the check can be found in the Courtesy Amount Block (CAB) of the check, a small box in which the amount is written numerically. Conceptually the system has three stages: CAB Location, Segmentation, and Recognition. First, the CAB must be located. This is not a trivial task because the CAB may vary in size and position. Specifications vary from bank to bank and some conventions vary across countries. Once the CAB is found the second step is segmentation, a process by which individual digits are separated. The final step is recognition; a neural network classifier is used to identify each of the digits.



Comp. Banco Ag. C1 Conta C2 Série Cheque nº C3 R$

018 237 0031 0 1 705 650000 5 6 S-229 951307 8 #73,00

Pague por este cheque a quantia de setenta e três reais

a ou à sua ordem

Bradesco

Banco Bradesco S.A.

de de 19

PC.PANAMERICANA-USP
PCA.PANAMERICANA 100

PRODUTO CONTA TESTE
CGC 001019188/0001-82

:237003130 0189513075A 377565000054

Figure 1.1: A sample check from Brazil

The main focus of this paper is the segmentation module of the Winbank system. Segmentation is done using the Hybrid Drop Fall algorithm described in [13, 1]. Conceptually, the algorithm replicates the motion of an object falling onto the digits from above. The object rolls along the contour of the digit until it can fall no further. At this point it cuts downward and continues. This algorithm uses the implicit assumption that cuts should be very short. In general this assumption is valid and the Drop Fall algorithm works well. However, this approach consistently causes errors in certain cases such as connected zeros. In these cases, there are often a large number of overlapping pixels and a long cut is required in order to separate the two characters. The Hybrid Drop Fall algorithm will make errors in these cases because they do not conform to heuristic assumptions.

Relying on heuristic assumptions results in error in cases were these assumptions are incorrect. The modifications described in this paper are designed to reduce the degree of reliance on heuristic methods and to consider more options for performing segmentation, with the intention of finding the correct solution more often. The modifications affect the system at two different levels. Recognition is used instead of heuristics analysis to choose between the results of different segmentation algorithms. At a lower level a modified segmentation algorithm, referred to as Extended Drop Fall, is developed to correctly segment cases in which the standard heuristics assumptions do not apply. More specifically, the Hybrid Drop Fall algorithm is modified to prolong the length of a cut in order to segment digits that require a long cut.

The Extended Drop Fall Algorithm and the recognition based method for choosing between segmentation algorithms are further described in Chapter Four of this document. Other modifications to the Winbank system, designed to improve the accuracy of the overall system, are presented in Chapter Five. Chapters Two and Three present background material on the subject. The former presents related research in the field while the latter provides an overview of the Winbank system. An evaluation of the accuracy and performance of the modified system is presented in Chapter Six. Chapter Seven describes possible extensions to the work presented in this paper and suggests other modifications or research that may improve the performance of the Winbank system. Finally, concluding remarks are presented in Chapter Eight.

2. Related Research

There is a long history of research in character recognition and segmentation [3,4,6,10,11,13,17,18]. Several different approaches to character segmentation are in use today [1,2,3,4,6,10,11,13,17,18]. Segmentation methods can be classified as either structural or recognition-based techniques although several approaches have elements of both types. Recognition based methods attempt to pick out recognizable digits from an image whereas structural methods try to divide up a given image into individual digits. All structural methods inherently depend on some sort of segmentation algorithm. Increasingly effective and complex segmentation algorithms have been developed for the purposes of structural segmentation. This chapter summarizes the major approaches to segmentation and the various segmentation algorithms available. Section 2.1 will discuss the most common strategies for segmentation and Section 2.2 describes specific segmentation algorithms. Finally section 2.3 describes previous research involving the Winbank Character Recognition System on which this thesis is based.

2.1 Segmentation Strategies

Segmentation is a necessary step in optical character recognition. It is performed either explicitly using a segmentation algorithm or implicitly as a result of recognition. If done explicitly, the image is first broken down into regions believed to contain digits and recognition is done separately on each region. Alternately, recognition can be done on the image as a whole and digits are separated out as a byproduct of recognition. This section describes a few of the major segmentation strategies used.

2.1.1 Iterated Segmentation and Recognition

Congedo *et al* [4] and Dimuaro *et al* [5] both use a strategy that alternates segmentation and recognition. An initial pass of segmentation is performed and the resulting segments are passed to a recognition module. If a digit is not recognized it is split using a specific segmentation algorithm and recognition is attempted on the resulting pieces. If neither piece is recognized, segmentation is assumed to be incorrect and the digit is split using a different algorithm. If one or both pieces are recognized, any unrecognized pieces remaining are segmented using the same algorithm. The set of segmentation algorithms employed are multiple variations of the "drop falling" algorithm (see Section 2.2).

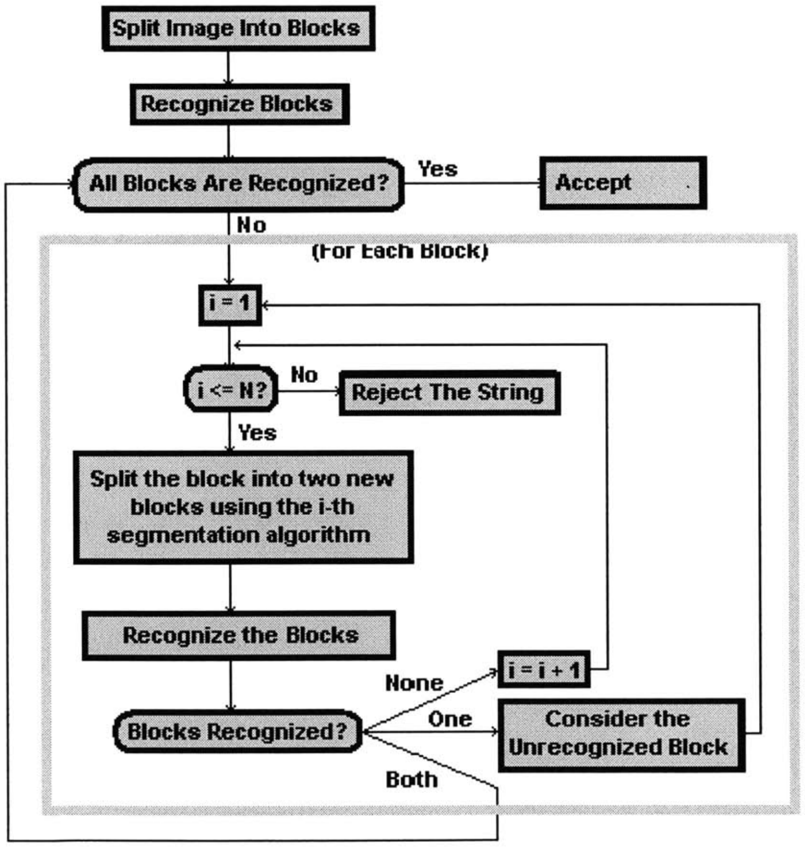


Figure 2.1: Iterated Segmentation and Recognition.

This approach is simple and straightforward but it has a number of drawbacks. The strategy is inefficient. Recognition is attempted before any segmentation is done. Since recognition is a computationally-intensive process, using other methods to detect connected digits can save processing time. Another problem is that no attempt is made to correct fragmented digits. Fragments are treated just like any other digit and will often lead to rejection. Finally, the segmentation algorithms are applied in a fixed order. It may be possible to exploit the fact that some algorithms offer better results in specific situations.

Congedo *et al* [4] reports that 91% of all digits were segmented properly in a test set consisting of U.S. zip codes. The results vary between 53% and 89% when using a single segmentation algorithm to separate digits. The percentage of digits that require separation using the segmentation algorithms is not reported in thier paper.

2.1.2 One Step Segmentation Reconsideration

Blumenstein and Verma [6] use a strategy that generates a number of paths and uses neural networks to select the correct paths. All paths are generated simultaneously using conventional heuristic methods and, a neural network selects the best ones. The system presented in Lee *et al* [7] uses a similar strategy in which a set of paths is generated and the combination of paths that result in the highest confidences in the recognition module is selected.

Both strategies require a large number of paths to be considered and may be computationally expensive. Also, no new paths are generated in later stages so the system does not take advantage of possible feedback from the recognition module. If the correct path was not generated in the original set, the system fails. Even if the path is slightly off, confidence values will go down dramatically and the correct path will not be selected.

Blumenstein and Verma [6] tested their system on samples of handwritten text scanned in from various sources. They report a correct segmentation rate of 86%. Lee et al [7] reports a rate of 90% correct segmentation but the tests were restricted to machine printed alphanumeric characters [7].

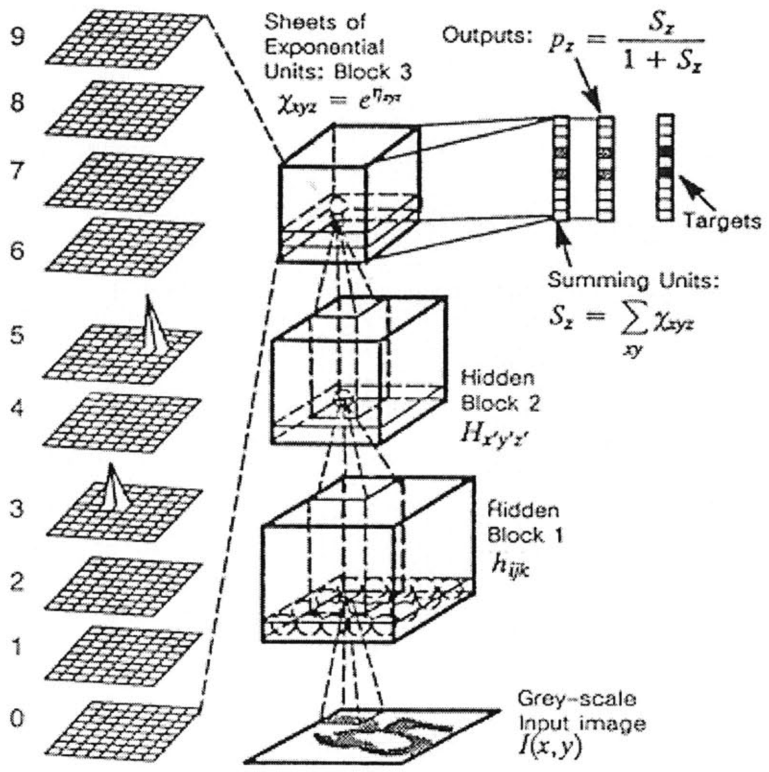
2.1.3 Recognition Based Methods

A simple example of a recognition based method is the Centered Object Integrated Segmentation and Recognition System (COISR) described in [17]. The system uses a sliding window that moves incrementally across a string of text. A neural network attempts to recognize characters centered in the window. Ideally each character will be recognized as it gets to the center of the window so explicit segmentation is not required. Even the characters used to train the network are unsegmented; the network is trained to recognize characters despite background noise.

Although this method performs segmentation and recognition in a single step, it is a computationally expensive process. Each incremental movement of the window must be analyzed by the neural network, which typically requires thousands of computations per instance. Furthermore, many of these instances result in non-recognition (i.e. window is between characters) or redundant recognition of the same character. It is possible to increase the rate at which the window slides across the text. This can eliminate some redundancy and reduce the amount of necessary computation but it is possible to "skip over" characters if the slide rate is too fast.

An alternate method for reducing computational expense in such a system is presented by Martin *et al* [9]. Instead of sliding across text, the window mimics the motion of the human eye and moves in bursts called *saccades*. The window moves forward *in ballistic saccades* and if necessary, jumps backward in smaller jumps called *corrective saccades*. These movements are controlled by a neural network, which learns to jump from character to character as well as recognizing individual characters.

Keeler and Rumelhart [18] describe another recognition-based approach, known as Self-Organizing Integrated Segmentation and Recognition(SOISR.) Instead of using a sliding window, the system searches all locations at once using a set of neural network "sheets". There is one sheet for every possible character and the sheet covers the entire image. The sheet searches the image for a particular character and activates in the areas where it thinks it has found the character. The example in Figure 2.2 shows a connected "3" and "5." If the system works properly, the sheet for '3' should activate where the '3' is found and the sheet for '5' should activate where the '5' is found. All other sheets should remain inactive.



9
8
7
6
5
4
3
2
1
0
Sheets of Exponential Units: Block 3
$\chi_{xyz} = e^{\eta_{xyz}}$
Outputs: $p_z = \frac{S_z}{1 + S_z}$
Targets
Summing Units:
$S_z = \sum_{xy} \chi_{xyz}$
Hidden Block 2
$H_{x'y'z'}$
Hidden Block 1
h_{ijk}
Grey-scale Input image
$I(x,y)$

Figure 2.2: Self Organizing Integrated Segmentation and Recognition (SOISR). Each self-organized neural network sheet is activated over the area of the image where its targeted character lies. Figure reproduced from [17].

The SOISR approach can recognize characters regardless of position. Although training the networks requires manual segmentation of the characters, no explicit segmentation is required once the networks are trained. Unlike COISR, it recognizes characters in a single pass and is therefore more efficient. However, recognition based approaches are intrinsically computationally intensive and are difficult to implement where execution time is a concern.

Both Martin et al [9] and Keeler and Rumelheart tested their systems on a database provided by the National Institute of Standards and Technology. Both papers report similar results, which vary between 90-95% accuracy with error rates between 4% and

8%. Depending on the number of connected digits, the average rejection rates varied greatly between 15% and 45%. Both systems produce exceptionally high recognition rates but low speed remains a major drawback.

2.2 Segmentation Algorithms

Structural approaches have dominated research in the field of character segmentation because most recognition-based approaches were beyond the limits of available computational resources. All structural approaches depend on a segmentation algorithm to break up the image into individual characters. As a result, considerable research has focused on the development of effective segmentation algorithms.

2.2.1 Basic methods

The most simple and straightforward segmentation algorithm is a vertical scan. The algorithm binarizes the image into black and white pixels and simply looks for unbroken columns of white pixels. This works well for machine-printed characters or handwritten characters in which a prescribed amount of whitespace is guaranteed. Unfortunately, normal handwritten characters are often slanted at different angles so vertical scans or scans at any other set angle will fail.

A more robust technique is to isolate regions of connected black pixels. This method separates the black pixels into sets in which each black pixel is adjacent to another black pixel in the set. This method works very well for digits that are not overlapped, touching or disjoint. However, two characters that are overlapped or touching will be treated as a single mass of pixels, and a character that is disjoint (that is, not all pixels are adjacent) will be treated as two or more characters. While it is straightforward to combine two disjoint segments, it is a much more difficult problem to separate connected digits. All the algorithms presented in this section use isolation of connected pixels as the first step in processing and then attempt to separate connected digits.

2.2.2 Hit and Deflect Strategy

The Hit and Deflect Strategy (HDS) attempts to find an accurate path along which to separate connected characters. The algorithm starts at an appropriate point in the image and move toward it. When it 'hits' the black pixels in the image it 'deflects' and changes its direction of motion. The algorithm follows a set of rules that maximizes the chances that it will hit and deflect its way to an accurate path.

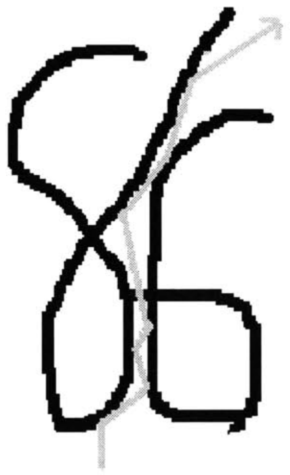


Figure 2.3: An HDS Split

One example of a hit and deflect Strategy was used by Shridhar and Badredlin [19] to segment handwritten words. Their system first tried to segment using a vertical scan along the center of the region of pixels. If the scan resulted in an unbroken column the region was segmented along that path. If the scan resulted in 2 or more color transitions (i.e., cutting through 2 or more lines), the algorithm assumed the characters were slanted and reverted to the HDS rules. The algorithm started at a peak at the bottom contour of the characters and attempted to move upward. It followed a set of rules designed to limit the number of cuts along the segmentation path. When the algorithm reaches the top of the image, the path is complete and segmentation is done.

2.2.3 Drop Fall Methods

Conceptually, a Drop Fall algorithm simulates an object, such as a marble or a drop of water, falling from above the characters and sliding downward along its contours. The basic algorithm is described by Congedo *et al* [4]. The object falls downward until it gets stuck in a "well," at which point it cuts through the contour and continues downward. Like Hit and Deflect strategies, the algorithm first selects a starting point and then follows a set of rules. In fact, Drop Fall can be considered a special case of HDS.

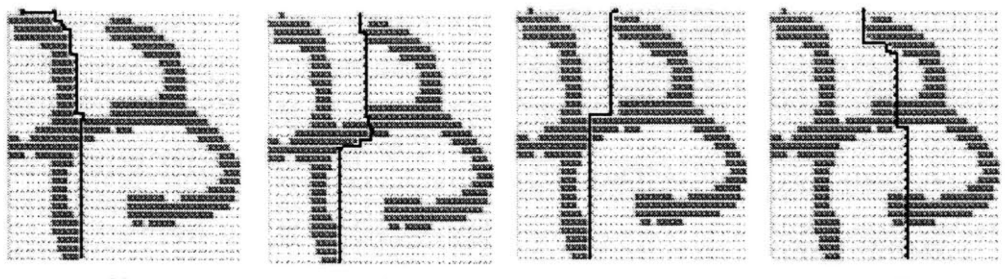


Figure 2.4: Drop falling algorithm (a) top left (b) bottom left (c) top right (d) bottom right

Drop Fall algorithms can be oriented in one of four directions, top-left, top right, bottom-left and bottom-right. Each version starts of in a specific corner of the image (as indicated by the name) and proceeds to move towards the opposite corner. Variants that start at the bottom of the image "fall" upward. They are equivalent to inverting the image about the horizontal axis and performing drop fall from the top of the inverted image.

2.2.4 Structural Analysis

Another class of segmentation algorithms uses structural properties of the characters to determine possible segmentation paths. These algorithms analyze the contours of connected digits and use properties of the contour slope or the presence of local maxima and minima to segment the digits. Abrupt changes in contour slope and pairs of relatively nearby maxima and minima often indicate points through which the character should be cut. Both [20] and [21] present examples of structural segmentation methods.

Strathy et al [21] presents an algorithm designed to find the start and endpoint of pen strokes. The algorithm seeks terminal points of both normal strokes and strokes that are overlapped or touching other characters. This is done by identifying points that are either corner points, local minima, local maxima, or exit point generated by extending a contour through a concave corner. These points are referred to as Significant Contour Points (SCPs). The algorithm then selects two points from the set of SCPs and cuts the segment along the line joining the two points. Segments are selected using the criteria listed below.

- **Preference to local minima-maxima pair**
- **Corner points are preferred**
- **Proximity of the points to each other**
- **Sharpness of the concavity**
- **Horizontal distance between a minima and maxima**
- **Distance of maxima/ minima to next local maxima/minima**
- **Proximity to the left side of the image**

Strathy *et al* reports accuracy between 48% to 98% using a set of US Zip codes as the testing set. Their system exploits the fact that zip codes have a fixed number of digits to improve accuracy. Also, zip codes are typically better spaced than other types of handwritten material such as prose or the courtesy amount on a check.

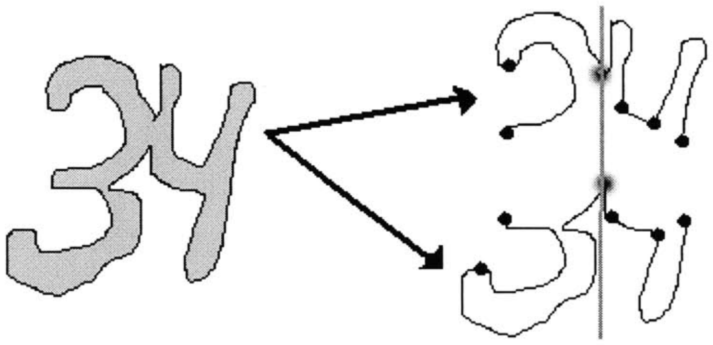


Figure 2.5: Contour Analysis Algorithm

Min-max algorithms constitute a class of structural analysis algorithms that focus on minima and maxima pairs [5, 22]. These algorithms divide the connected digits into upper and lower contours and search for minima on the upper contour and maxima on the lower contour. The algorithm selects a suitable pair of pints and cuts the region through these two points. The choice of segmentation points is based on the distance between the points, proximity to the center of the image, and the number of color transitions (cuts) along the path. The Winbank system uses such an algorithm in addition to Hybrid Drop Fall.

2.3 Winbank

This thesis is based on modifications to the Winbank Character Recognition System developed at Sloan School of Management at MIT. A great deal of research work has been done on various aspects of the system. This section describes the work most relevant to the segmentation module.

Sparks [3] presents a linear segmentation and recognition architecture using a segmentation analysis module for verification. The system normalizes a numeral string to a standard size and then performs segmentation using contour analysis and a Hit and Deflect Strategy. The results are modified using a "Segmentation Critic", which compares the segments with stored structural primitives [3, 14] the Segmentation Critic can veto classification of large segments as digits or force segmentation to be undone. The analysis is performed using a decision tree method based on stored structural rules and by template matching against stored structural rules. Although segmentation is considered the system fundamentally consists of linear segmentation and recognition because segments are not reconsidered after recognition.

Khan [13] presents an algorithm, called Hybrid Drop Fall, which combines all four variants of the basic Drop Fall algorithm with heuristic analysis. Although a single orientation will tend to make systematic errors, the other orientations will generally not make the same error. By using all four orientations, there is an increased probability of correct segmentation because one of the four is likely to have produced the right results.

Chapter Four describes an improvement to the Hybrid Drop Fall algorithm that allows it to segment properly in frequently problematic cases. Further discussion of Drop Fall and Hybrid Drop Fall algorithms can also be found in Chapter Four.

Dey [1] presents a modified feedback architecture that reconsiders segmentation using the results of recognition. This architecture is based on the method presented by Congedo [4] and uses recognition to take the place of the segmentation critic. The system passed the entire string repeatedly through stages of segmentation and recognition using a feedback mechanism. In the segmentation stage all segments that require segmentation are separated and then passed to the recognition stage. Segmentation is done using multiple algorithms including HDF. The best result is chosen as the proper segmentation. Also, fragments are combined with their nearest neighbors.

The work described in the present document adds a new segmentation algorithm and a new method of path selection to the system presented by Dey [1]. A more detailed discussion of the feedback architecture is provided in Chapter Three. Testing and accuracy results for this system are included in Chapter Six.

3. The Winbank System

Winbank [1,2,3,13] is a segmentation based character recognition system for the automated reading of hand written checks. It is a sophisticated system designed to meet the rigorous accuracy and performance requirements of commercial verification of check values. A general overview of the system is necessary to better understand the improvements presented in Chapters Four and Five. More detailed discussions of the segmentation module is presented in those chapters.

The Winbank system operates in four stages: Courtesy Amount Block (CAB) location, Segmentation, Recognition, and Post-Processing. In addition, there is a feedback mechanism that repeats segmentation and recognition processes as necessary to improve accuracy. The architecturee of the Winbank system is summarized by Figure 3.1. The program takes in a check image, finds the CAB, segments the digits and then reads their values. As a final step the system verifies that the value read is reasonable for a check.

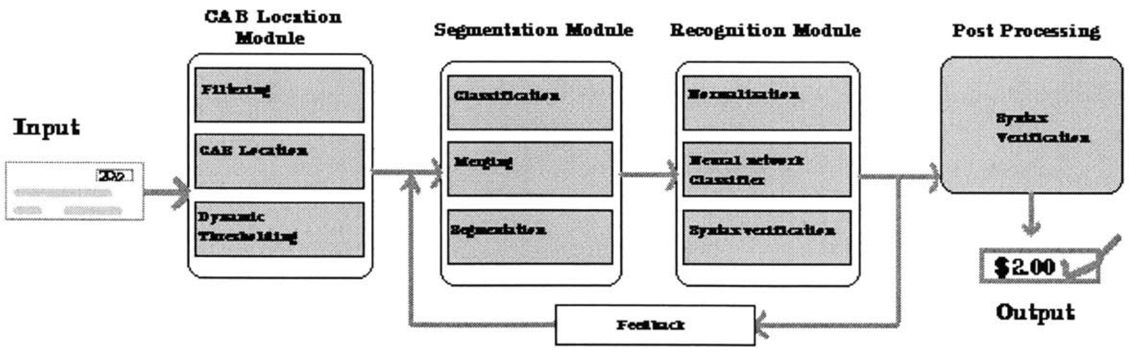


Figure 3.1: Winbank System Overview

3.1 CAB location and Binarization

The program begins with a grayscale image of a paper check. The first stages of the process are binarization and Courtesy Amount Block (CAB) location. The image is first filtered using a median filter to reduce noise. The system then attempts to find the courtesy amount block using a set of heuristics designed to locate the delimiters of the CAB. Delimiters may vary across countries and localities. For example, American checks have a full box delimiting the CAB, but Brazilian checks have just two vertical lines on either line. The CAB finding routine starts at prescribed location (in the upper right portion of the check in most cases) and scans outward looking for horizontal lines and vertical lines. The algorithm tries to find an enclosing box or at least part of a box (i.e., two vertical bars or two horizontal bars.)

Once the CAB is found, the contents of the CAB are binarized using dynamic thresholding to separate the background of the check from the writing [1]. In dynamic thresholding, a histogram of pixel intensities is generated and a threshold value between the two highest peaks (which should correspond to background and writing) in the histogram is chosen. All pixels above this value are considered black while all other pixels are white. Ideally, this produces a good representation of the numbers present in the check [2].

3.2 Segmentation

This image is then segmented. Connected regions of pixels (called segments) are located and organized into "Blocks." Each block contains a set of pixels, the location of the segment in the original image, and a classification for the segment. Segments are classified as digit, fragment, multiple, punctuation, nonsense, or horizontal, based on the height, width, weight (number of pixels), and location in the original image. Segments classified as nonsense or horizontal are most likely noise or extraneous information irrelevant to the value of the check. Segments classified as punctuation are either commas or periods and are noted in the post processing stage. Segments classified as fragments are most likely parts of a digit that were not connected to the rest of the region. These segments are merged with nearby segments. Finally, segments classified as "multiple" are most likely two or more digits that are touching and therefore seen as one connected region; such segments must be separated further into separate regions.

Winbank generates a list of paths along which to segment connected digits. Nine paths are generated using three different segmentation algorithms. The first path is generated using a Min-Max Contour Analysis algorithm similar to the one described in [6]. Four more paths, corresponding to the top-right, top-left, bottom-right, and bottom-left orientations are generated using the Hybrid Drop Fall algorithm described in Section 2.2.3. Finally, a third set of four paths is generated using the Extended Drop Fall algorithm, which will be presented in Chapter 4.

Once the paths are generated, they are ranked using a set of structural heuristics. The system tries to find a path with exactly one cut near the center of the image. Paths with a single cut are preferred to those with multiple cuts, and shorter cuts are preferred to longer cuts. The rankings also take into account the aspect ratio of the resulting segments and a long list of other characteristics.

Finally, the image is separated along each path and recognition is attempted on the resulting segments. The correct path is determined based on the number of recognizable digits. Ideally both will be recognized, and the segmentation process is assumed to be correct. If multiple paths produce the same number of recognizable segments, preference is given to the path with the higher rank. If none of the paths result in segments that are not recognized, it is assumed that these segments need further segmentation. Path selection based on recognition is a new addition to the Winbank system and will be discussed in greater detail in Chapter 4.

3.3 Recognition

The segmentation process should result in a set of segments representing hand written characters. These segments are then normalized and recognized individually. Normalization is a process by which the segment is adjusted to correct slants, thinned to a one-pixel wide skeleton, and rethickened to a standard width. This process eliminates superficial variances and emphasizes the most important structural aspects of the character.

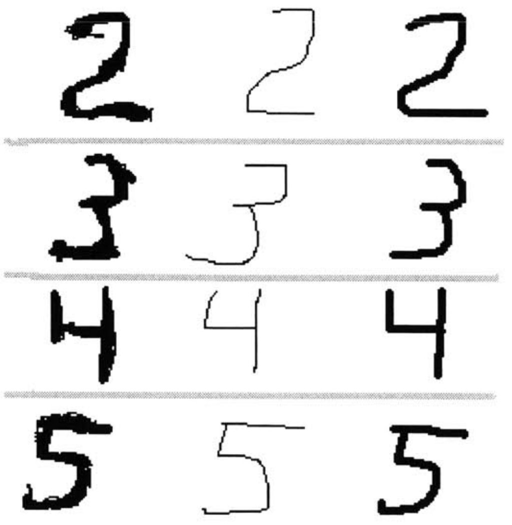


Figure 3.2: Thinning and Rethickening

The normalized character is then classified as one of a set of possible characters using a neural net classifier. The classifier must determine the identity of the digit with a high degree of confidence; otherwise the segment is rejected. To increase accuracy, four different networks are employed and an arbiter is used to make the final decision. In addition the classification is verified using structural heuristics and may also be rejected at this stage. Rejected segments could be ambiguous characters, or characters distorted by noise or overlapping. They may also be the result of improper segmentation.

3.4 Feedback

Rejection caused by improper segmentation is corrected by feedback. Rejected segments are segmented and recognized again with relaxed heuristic criteria for classifying a segment as multiple digits. The feedback loop is iterated until all segments are recognized or until a predefined maximum number of iterations have occurred. The more times a segment is rejected the more likely it is to be classified as multiple digits and separated, thereby giving the system more chances to correct segmentation errors. If errors still remain after the maximum number of iterations have passed, the check is rejected and must be recognized manually.

3.5 Post Processing

As a final processing step, recognized digits and punctuation are analyzed to determine whether the value is a valid amount for a check. Winbank will accept the value "1.00" but not "1,23,45" because the later shows punctuation that is unlikely on a check in Brazil (and the U.S.) Post processing verifies the recognized amount against a set of rules to eliminate obviously incorrect results.

4. Segmentation with Extended Drop Fall

The segmentation module of the Winbank system has been modified to include a new segmentation algorithm that eliminates a number of weaknesses in the previous version of the system. The algorithm, referred to as Extended Drop Fall (EDF), is a variant of the Drop Fall (DF) method and is based on the Hybrid Drop Fall segmentation algorithm presented by Khan [13]. It is designed to improve segmentation on the frequently occurring case of connected zeros, which is often segmented incorrectly, as well as on other cases that are peculiar to Brazilian checks.

Because checks are frequently written for round values such a "\$100" or "\$21.00" two successive zeros are common. The two zeros are frequently connected or slightly overlapped. In fact, a random sampling of over 400 checks shows that approximately 80-85% of successive zeros are connected. Furthermore, 45-55% of all connected digits consists of a pair of zeros. The previous version of the Winbank system used an HDF algorithm to perform segmentation. Although the system was accurate in many case it showed relatively poor performance on connected zeros. The HDF algorithm was able to perform accurate segmentation for only 25% of the connected zeros in test cases. Considering the frequency of connected zeros it is important to improve segmentation in such cases. The Extended Drop Fall Segmentation algorithm doubles the number of connected zeros segmented properly without compromising accuracy in other cases.

Connected zeros are problematic because of the structural properties of the joint between the digits. The joint between connected zeros tends to be longer than the width of a pen-stroke and slanted at an angle. For most other cases the joint is no longer than the width of a stroke and can be separated by cutting down a straight vertical line at the joint. Drop fall algorithms mimic the motions of a falling object that falls from above the digits, rolls along the contour of the digits and cuts through the contour when it can fall no further. The object follows a set of movement rules to determine the segmentation paths. These rules follow the implicit assumption that cuts through a contour should be short and straight through the contours.

When segmenting connected zeros these rules tend to cut into the loop of one of the zeros. Once the path cuts into the loop it will also cut out leaving an open arc on one side of the path, and a digit with a small piece of the broken zero on the other. This is a particularly distressing problem because it leads to false recognition. The recognition module is sophisticated enough to recognize zeros despite the extra fragment, however an open arc can look like a number of other digits. An open arch on the right hand side can look very much like a 7and in some cases it can be mistaken for a 2. Similarly an open fragment on the left side can look like a six. Because of this error a frequently occurring string “00” can be mistaken for “07”, “02”, or “06.”

This type of problem is not unique to zeros a long junction near a loop can cause problems for other looping digits. Similar problems can occur with digits such as “9”,”8”,”6”, and “2,” depending on the writer and the nearby digits. In order to avoid these problems, the algorithm needs to make a long cut through a diagonal joint. However, in the general case, a long cut through a segment is a bad one.

In order to avoid such problems, segmentation paths need to cut along the junction and not enter the loop. Drop Fall algorithms move downward on a binarized image by seeking white pixels in order to avoid the contour. When a cut is made into a black pixel, the algorithm continues to seek white and minimizes the length of a cut. In the general case a short cut is more likely to be correct; however, for connected zero and other special cases, the cut should be extended. Extended Drop Fall will seek black pixels rather than white once a cut has been initiated (i.e., it is currently on a black pixel). This results in longer cuts that follow the interior of the contour and are more likely to be correct for connected zeros.

In order to use the Extended Drop Fall algorithm effectively, we must first identify cases in which it is appropriate. In the general case, the EDF algorithm produces a path that is less likely to be the correct one. EDF is only useful in cases where other Drop Fall algorithms tend to fail. The system needs to be able to choose between different paths generated using different segmentation algorithms. The HDF algorithm generates a set of paths, using standard drop fall from four different orientations. The best segmentation path is selected based on a heuristic analysis of each path. The EDF algorithm also generates multiple paths using the four different orientations of the DF algorithm, but it cannot rely on heuristic analysis to select the correct path because the algorithm is designed for exceptional cases. Instead, all paths are evaluated using recognition. The characters are segmented along all the paths and we look for a path that results in two recognizable segments. If more than one path generates two different sets of recognizable digits, we fall back to heuristic rankings. If none of the paths resulted in two recognized digits, then one needs to pick the best path with one recognized digit. The recognized digits and the identity of the segmentation algorithm that generated them can also be useful. For example, two recognizable zeros that were produced by EDF are more likely to be correct than a "0" and a "7" produced using the standard algorithm, because the EDF algorithm is especially designed for this case.

Recognition based path evaluation is an improvement in its own right. It can improve accuracy even without the EDF algorithm because it does not rely on heuristics. The fact that recognition was successful is a strong indicator that segmentation was correct. When the Extended Drop Fall algorithm is added, this method of evaluating paths becomes an essential part of the segmentation module.

4.1 Drop Fall

Drop Fall is a simple algorithm that offers remarkably good results. The Hybrid Drop Fall concept augments the algorithm by attempting DF in four different orientations. The Extended Drop Fall algorithm developed in this paper is based on the HDF algorithm and operates in essentially the same way. A detailed explanation of standard DF and HDF is necessary in order to explain the differences in the EDF algorithm. Unless otherwise stated, all references in this section refer to the standard DF algorithm using the *top-left* orientation. The basic algorithm mimics a "marble" that falls from a starting point located at the top left corner of the image and has a tendency to fall to the right if obstructed. Variants using other orientations will be discussed in Section 4.1.2.

4.1.1 Start Point

One of the most important aspects of Drop Fall is the point at which the algorithm begins to construct a path. The starting point should be above the image and toward the left side because the simulated marble falls downward and to the right. If the starting point is selected too far to the left, the marble can slide off the left side of the first character and produce no segmentation. Similarly if the starting point is chosen too far to the right, the marble can slide off the right side of the second character (see Figure 4.1).

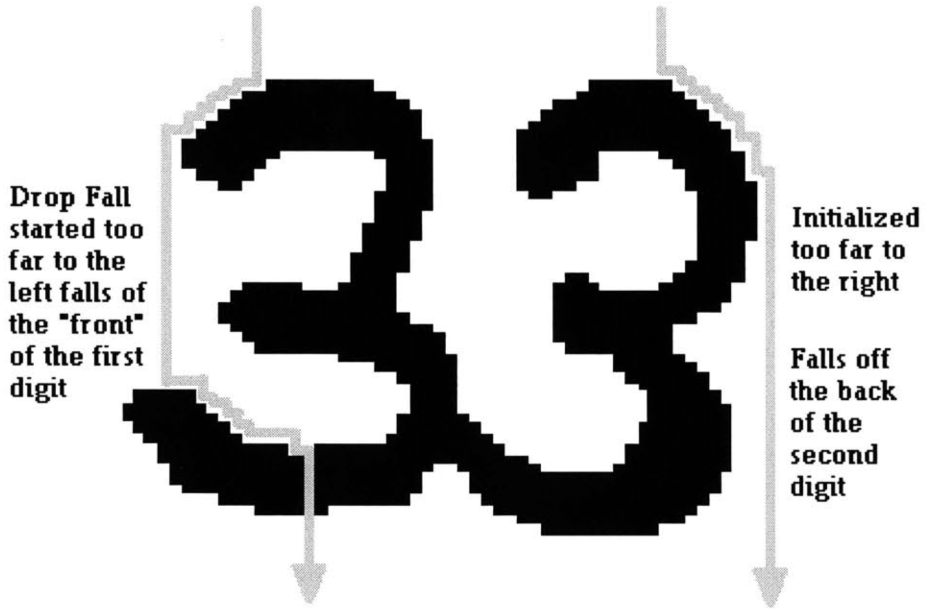

Drop Fall started too far to the left falls of the "front" of the first digit
Initialized too far to the right
Falls off the back of the second digit

Figure 4.1: Bad starting points for Drop Fall Segmentation

The starting point can be any position between the connected numbers and above the connection (area to be cut). The system assumes that an appropriate starting point will have black pixels on either side of it. Furthermore, the point should be close to the top of the image. The system scans the image row by row, starting at the top, and works its way down until it finds a black pixel with white space immediately to the right and another black pixel in the same row, but further to the right (see Figure 4.2). The white pixel immediately to the right of the left-hand boundary is used as the starting point.

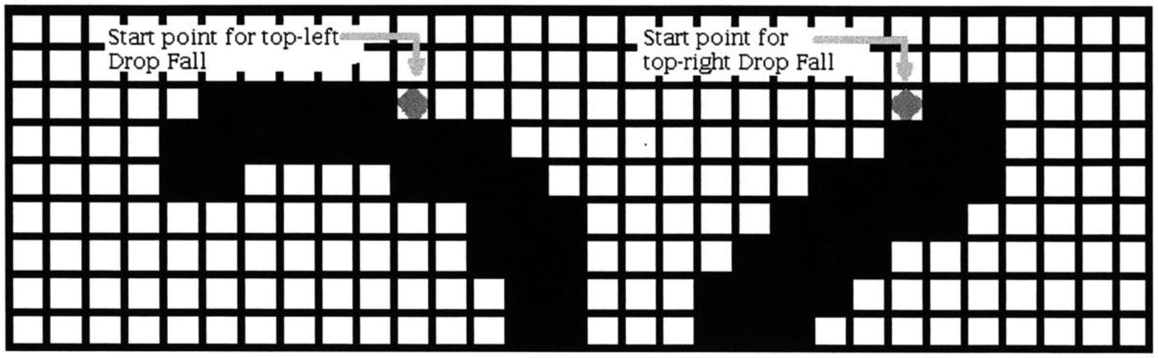

Start point for top-left Drop Fall
Start point for top-right Drop Fall

Figure 4.2: Selecting a good starting point for Drop Fall Segmentation

4.1.2 Movement Rules

After selecting an appropriate start point, the algorithm "drops the marble." The algorithm follows a set of rules that advances the marble one pixel at a time until the bottom of the image is reached. The algorithm considers only five adjacent pixels: the three pixels below the current pixel and the pixels to the left and right. Upward moves are not considered, because the rules are meant to mimic a falling motion. (The marble does not bounce back up.) Characters are treated as solid objects, the marble does not cut through the boundaries (i.e., enter a black pixel) unless all other options are precluded. The movement rules are depicted graphically in Figure 4.3.

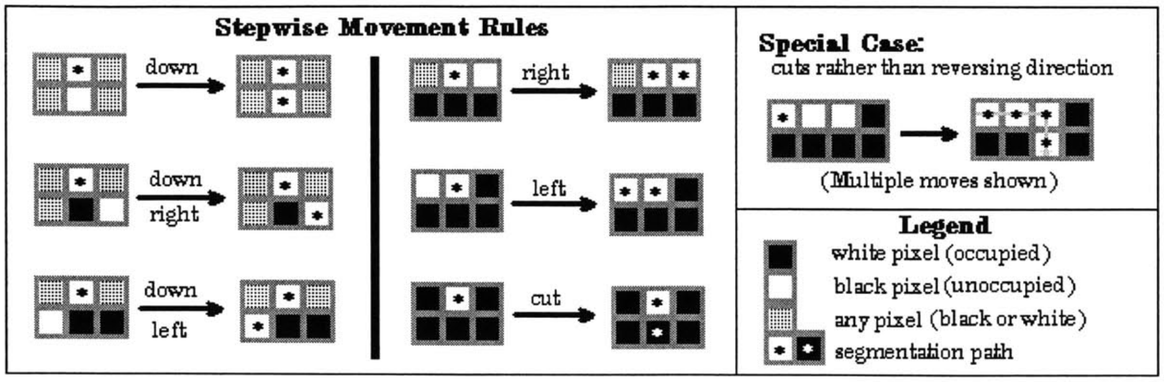


Figure 4.3: Stepwise movement of the Drop Fall Algorithm

The marble will always move directly downward if that pixel is empty (white). If that pixel is occupied then the marble will move down and to the right. If both the pixels directly downward and down and right are occupied the marble will fall down and to the left. If all three pixels at the lower level are occupied the marble move directly to the right. If all pixels except for the one directly left of the marble are occupied then the marble will move directly left. Finally if all pixels around the current point are occupied the marble will do move downward, thereby cutting the contour.

Pseudo Code for Movement Rules

```
IF (WHITE (down))              THEN {GO (down)            }
IF (WHITE (downRightt))        THEN {GO (downRightt)  }
IF (WHITE (downLeft))   T      HEN {GO (downLeft)       }
IF (WHITE (right))             THEN {GO (right)            }
IF (WHITE (left))              THEN {GO (left) }
ELSE                           GO (down);
```

There is an additional restriction that the marble cannot move back to a position it has already occupied. If the above rules specify a location that has already been occupied, the marble will cut downward instead. This prevents the algorithm from getting stuck in a cycle when it reaches a flat plane. If this rule were left out the marble could move right across a flat section and run into a pixel directly to the right, and then enter a cycle. At this point the rules specify movement to the left, but that puts it back where it started so the pixel would just move back and forth on an infinite basis.

4.1.3 Orientation

The Drop Fall algorithm described above corresponds to the top-left orientation. Three more orientations are possible and all four are used in the HDF algorithm. The *term top-left* refers to the fact that the algorithm in this orientation starts at the top of the image and toward the left side. Drop fall with the other three variants is similar except that they do not start near the top left. Two of the orientation start at the bottom of the image and "fall" upward.

Top Right Drop Fall: This variant is identical to the top-left orientation except that it starts at the top of the image toward the right side. The marble still falls downward but favors the left side instead of the right. Conceptually, the algorithm uses the same movement rules on an image that is "flipped" (rotated 180^o) along the vertical axis.

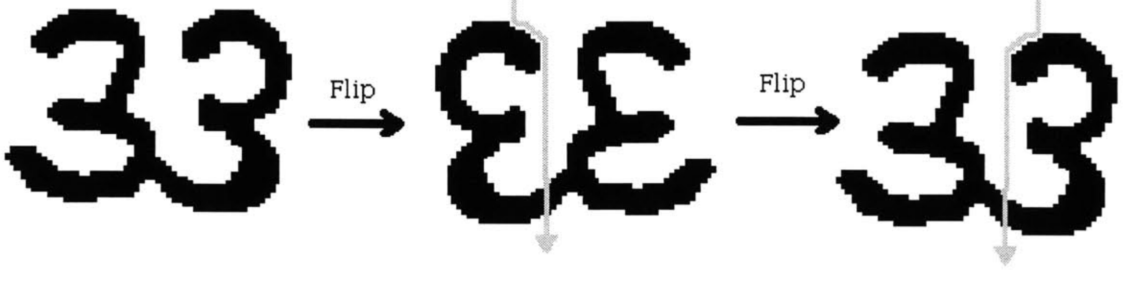


Figure 4.4: Top-Right Drop Fall

More specifically this flipping can be described by the following image transform:

(1) P'(X,Y)= P(w-X, Y)

In this expression, **P** is the array of pixels and **P(X,Y)** refers to the pixel in at the Xth column of the Yth row and **w** is the width of the image. The segmentation path is found by flipping the image, performing the DF algorithm described in the previous section, and then flipping the resulting path.

Bottom-Right: This variant is similar to drop fall in the top-left orientation except that the image starts out at the bottom of the image and moves upwards. Conceptually this corresponds to flipping the image 180o degrees about the horizontal axis and performing top-left Drop Fall.

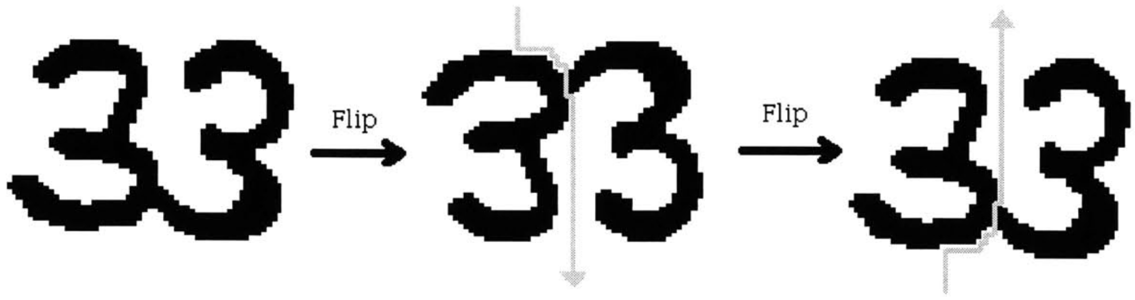


Figure 4.5: Bottom-Left Drop Fall

More rigorously, the necessary transformation is described as follows:

(2) P'(X, Y)=P(X, h-Y)

Here **h** is the height of the image and the rest of the parameters are the same as the above. The segmentation path is obtained by flipping the image, obtaining the segmentation path using standard drop fall, and then flipping that path.

Bottom-Right: This variant of Drop Fall segmentation is similar to the other three except that it starts at the bottom-right corner. Like Bottom-Left Drop Fall this algorithm falls upwards, but the marble tends to move to the Left as in Top-Right Drop Fall. Conceptually, this algorithm is equivalent to performing standard DF on an image that has been flipped along both the horizontal and vertical axes.

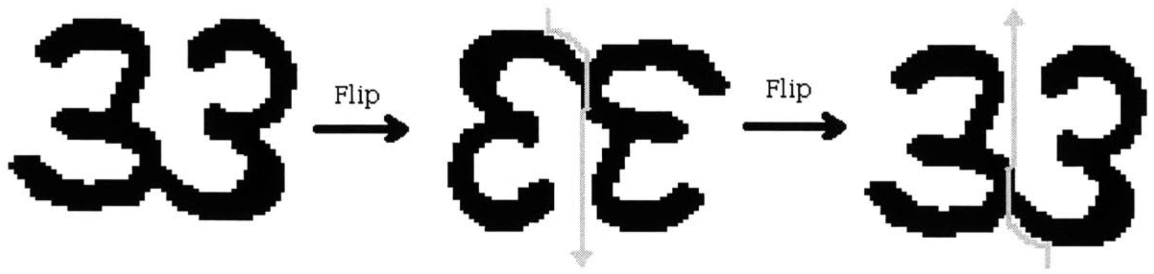


Figure 4.6: Bottom-Right Drop Fall

The necessary transform is described as follows:

$$\text{(3)} \qquad \mathbf{P'(X,Y)=P(w\text{-}X,\ h\text{-}Y)}$$

As with the Bottom-Left and top-right orientations flipping the image, performing standard DF, and then flipping the resulting path will produce the correct path.

In practice, the HDF algorithm changes orientation by redefining directions rather than flipping the image. For example, when performing top-right drop fall the implementation redefines left to be a greater column number rather than a lower one. The code for the movement rules remains the same. A similar adjustment is done for orientations that "fall" upward. Finally, the HDF algorithm selects the best path by using the path evaluation heuristics described in Section 4.3.1.

4.2 Extended Drop Fall

In order to correctly segment connected zeros and other problematic cases, the Extended Drop Fall approach will extend cuts diagonally through a contour. The algorithm considers the pixel it is on, as well as the five pixels surrounding it from the sides and below. If the current pixel is white, the algorithm follows the movement rules specified in Section 4.1.2. However, if the current pixel is black it follows the rules described below:

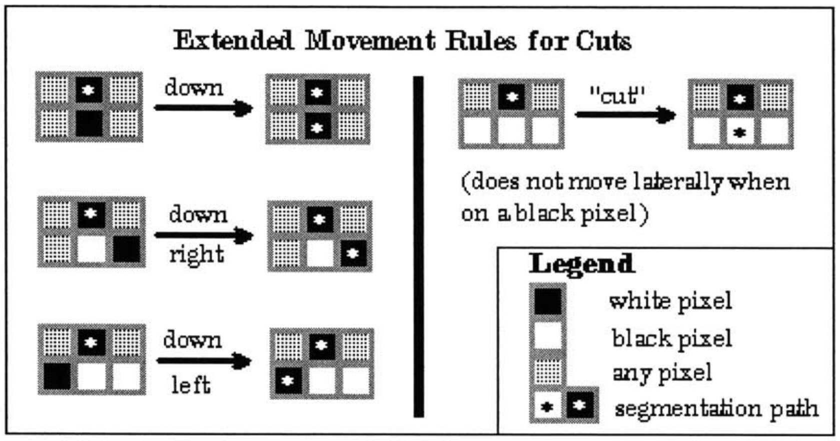


Figure 4.7: Additional movement rules for Extended Drop Fall

The standard algorithm attempts to move the marble onto white pixels at all times. EDF will seek black pixels when it is already on a black pixel. When these rules are in effect, the "marble" will fall directly downward if there is a black pixel below. It will fall down and to the left or right if there is no black pixel directly below (preference to the right.)

```
Extended Drop Fall Movement Rules:
Pseudocode

IF (ONWHITE)   THEN
{
        IF (WHITE(down))          THEN { GO(down)          }
        IF(WHITE(downRight))      THEN { GO(downRight)     }
        IF(WHITE(downLeft))       THEN { GO(downLeft)      }
        IF (WHITE(right))         THEN { GO(right)         }
        IF (WHITE(left))          THEN {GO(left) }
        ELSE                              {GO(down)        }
}
ELSE
{
        IF(BLACK(down))           THEN {GO(down)           }
        IF(BLACK(downRt))         THEN {GO(downRt)         }
        IF(BLACK(downLt))         THEN {GO(downLt)         }
        ELSE                              {GO(down)        }
}
```

However, the pixel will always move downward; unlike the "white seeking" rules that apply when the marble is on white, the "black seeking" rules do not move directly left or right. The intent is to extend a diagonal cut, not to follow the contour. If all three pixel below the current pixel are white, then the marble moves directly down and terminates the cut.

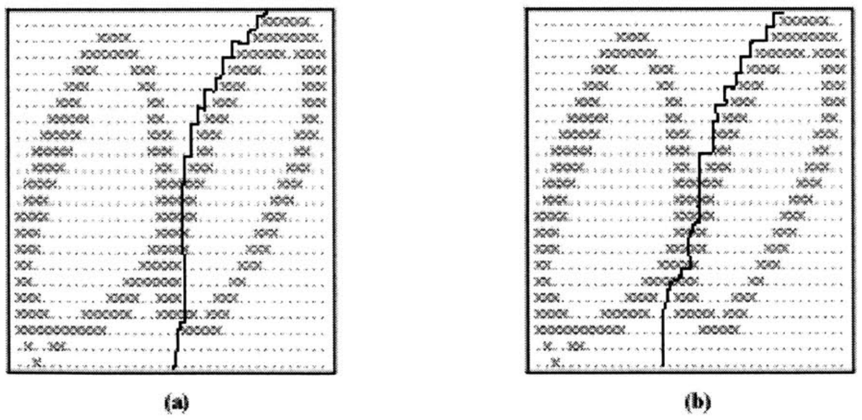


Figure 4.8: Segmenting double zeros (a) best results with Hybrid Drop Fall (b) Extended Drop Fall

Figure 4.8 illustrates the difference between the two algorithms. Figure 4.8a shows the best result using hybrid drop fall (top-right orientation in this case.) The other three orientations produced similar results. The Extend Drop Fall approach correctly segments the digits by following the diagonal junction all the way through (depicted in Figure 4.8b).

4.3 Recognition Based Path Evaluation

The goal of segmentation is to segment characters along the right segmentation path. When multiple paths are possible, there must be a mechanism for selecting the best one. The Winbank system now uses a recognition-based approach to select the path rather than rely on heuristic analysis. Nine paths are generated using various algorithms: one using contour analysis, four from Hybrid Drop Fall, and four more using Extended Drop Fall. The paths are first ranked heuristically using the same criteria as in the previous Winbank implementation. Characters are then separated along each path and recognition is attempted on all sets of digits. The heuristic rankings are still used as tiebreakers.

Because some recognition is now performed in the segmentation module, the process need not be repeated in the Recognition module. Winbank stores the recognized values of the digits along with the separated image. After generating paths, the system separates the character along each path and evaluates the results. Each evaluation is done using a fresh copy of the original segment. When a path is selected, the copy of the segment corresponding to this path and the new segments generated from it will replace the original. Because the replacement segments have already been recognized, the recognition module will not attempt to recognize on a redundant basis. This saving in computational time recovers some of the performance penalty involved in recognizing the results of each path. Furthermore, since recognized segments are most likely to be correct, it saves extra passes through the Feedback loop.

4.3.1 Heuristic Path Evaluation

Paths are ranked heuristically using structural features of the path and the image. The segmentation algorithm that generated the path is also considered. Features can be either positive or negative, so a point based ranking issued. Paths are awarded points for each positive feature and points are subtracted for negative features. If more than one path has the same value, the two are ranked in the order they were evaluated.

The most important criteria is the number of cuts made by the segmentation path. In most cases, only one cut is necessary. If more cuts are made, it is likely that the path is cutting through a digit. The path value is unchanged if there is no cut or a single cut. Two points are subtracted for every cut beyond the first. A cut is identified using transitions from white to black and vice versa. A white to black transition signals the start of a cut and a black to white transition indicated termination of the cut. If the space surrounding an image is assumed to be whitespace, so a black pixel in the first or last row will also be considered a color change. Finally, it is possible to cut a diagonal line without a color transition if the marble is moving perpendicular to the diagonal. For example, if there is a pixel immediately downward and immediately to the right of the reference point and the path moves down and right to a white pixel, it has “squeezed” through these pixels and cut the diagonal line.

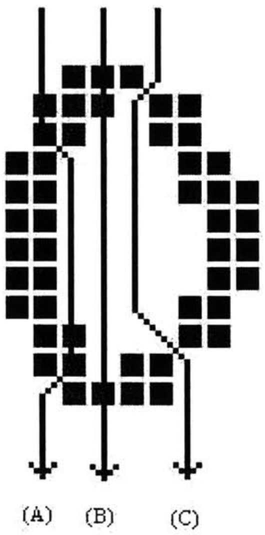


Figure 4.9: Detection of cuts.

Each of these paths cuts through two lines. (A) black-white transitions indicate cuts (B) black pixels at the very top and bottom rows of the image are also treated as transitions. (C) cuts that “squeeze through” diagonally without color transition

The junction between two characters is typically characterized by two corners, which are normally convex corners. In some cases the junction can be flat, but a cut through a flat region normally means a stroke is being broken incorrectly. A convex corner is very unlikely. One point is awarded for each concave corner at a cut point. One point is subtracted for every cut through a flat area and three points are subtracted in the event of a convex corner.

The length of the cut is another important criteria. This length is the distance from the initial cut point to the terminating cut point. For multiple cuts, the cut length is measured from the start of the initial point of the first cut to the terminating point of the last cut.

Two points are subtracted from the score if this cut length is above an empirically determined threshold.

Finally, the path evaluation score reflect characteristics of the segmentation algorithm used. Drop Fall from the top-left and top-right orientations tend to make errors in the lower half of the image, while bottom-right and bottom-left make errors in the top half. One point is subtracted if the algorithm made cuts where it may cause errors. The order in which paths are listed can also make a difference in the case of identical scores. If one algorithm offers empirically better results than another, it should be listed ahead of it.

4.3.2 Path Selection Rules

The appropriate segmentation is done using the results of recognition. Obviously the more blocks recognized the better. But when multiple of paths all result in the same number of recognized block tiebreakers are necessary. The best path is chosen using the following set of rules.

Path Selection Rules

1. If any of the EDF paths results in two recognized zeros, select the highest ranked of these paths.
2. Otherwise, if there are paths that produce two recognized digits select the highest ranked of these paths.
3. Otherwise, if there are sets that produced one recognizable digit, select the highest ranked of these paths.
4. Otherwise, select the highest ranked path in the set

5. Other Improvements to the Winbank System

The main focus of this thesis is on the Extended Drop Fall algorithm as well as the recognition-based path selection mechanism necessary to utilize the algorithm correctly. However, work has been done to improve other aspects of the Winbank system, as well. The mechanism for reconnecting fragmented digits has been improved. Segments identified as character fragments were previously merged to nearby segments only during the first iteration of the feedback mechanism. The system has been improved to undo merges and remerge fragments with different digits when merging fails.

The recognition module has also received some attention. The neural networks were trained using an iterative approach in which mistakes made in testing are included in the training set in each successive iteration. The scope of the slant correction algorithm, used in normalizing segments before recognition, has been reduced to increase speed and eliminate some types of recognition errors.

5.1 Merging

When a segment is identified as a fragment of a character, the system attempts to merge it with a neighboring segment. A set of four neighboring segments is considered and the best one is chosen based on criteria such as proximity, overlap and whether or not the fragment has previously been merged to the segment. Segments that have been previously merged will not be merged again because a correct merge is assumed to result in recognition. Vertical overlap of segments takes precedence over proximity, because although these factors tend to go hand in hand, the former can be more important in some cases. A five with a disjointed top stroke is a common example of such a case. The top stroke is often closer to another character in proximity but, the stroke starts directly above the rest of the digit, which makes vertical overlap a much better indicator. If a suitable segment is found, the fragment is merged with that neighbor for consideration by the recognizer. If unrecognized, the two merged segments will be separated on the next iteration of the feedback loop and a new option for merging will be attempted.

On each pass through the feedback loop, a fragment can have a different set of neighbors for consideration. Segments can be separated or merged so that the neighbor segments may not be the same on every iteration. Furthermore, the fragment could potentially be placed in a different position in the "list" of segments. The set of neighbor segments considered depends on the internal representation used to handle the check value. A description of this representation is provided in Section 5.1.1, before presenting the details of the implementation of the merging algorithm in Section 5.1.2.

5.1.1 Representation of the Courtesy Amount Block

The Winbank system repeatedly performs separation and merging of character segments. These operations need to be done efficiently and, if necessary, undone efficiently as well. To simplify these operations, the system uses a data structure with attributes of both a linked list and a binary tree. Conceptually, the value of a check is made up of a string of characters, which suggests a list structure or dynamic array. However, the feedback mechanism requires that separation and merging need to be undone. This operation is performed by backtracking on a tree-like structure. The representation chosen is a compromise between the two requirements.

Initially, the image is separated into regions of connected pixels. Each of these regions is placed in a block; blocks are connected together as a simple linked list. At this stage, blocks do not necessarily hold individual digits. The region may be a multiple digit in which case separation is necessary or it may be a fragment that needs to be merged. When separation is performed, the newly formed segments are stored in blocks that branch out from the original. Much of the information for the original segment is kept intact so that the operation can easily be undone. Similarly when a merge is done, a new block replaces the two merged blocks. However, the original blocks will branch out from this new block so that the merging may be undone as necessary by flattening the tree.

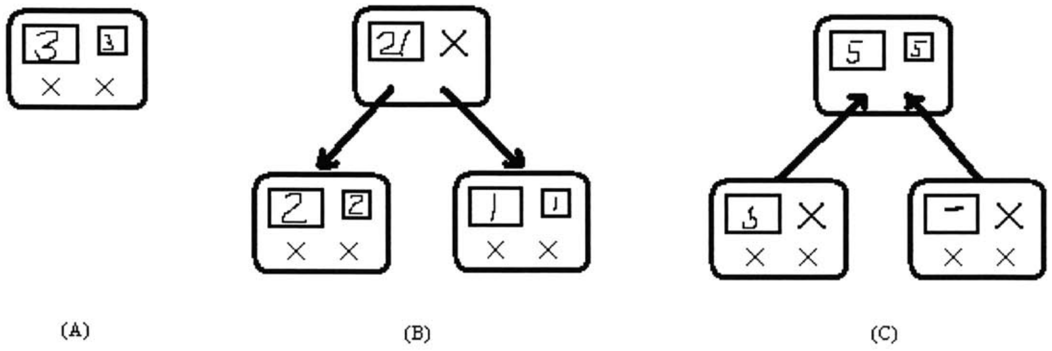


Figure 5.1: Internal Representation of Digits
Segments are stored in blocks (a) a simple block (b) separated blocks (c) two merged blocks

Each block can contain a segment, a normalized segment and two pointers to other blocks. The normalized segments are required by the recognition module and are created just before recognition. Normalizing the segment causes some loss of information so the original segment is kept for future merging or separation. If a segment is not recognized, the normalized segment is deleted. If the segment is recognized, the original is deleted because it is no longer needed. Similarly, when segments are merged or separated, the original blocks are kept but the normalized segments and unnecessary information are deleted.

Each block is given a block type to keep track of the type of segments it represents. A block can be classified as UNKNOWN, HOLD, MERGE, or SEPARATE. Blocks containing a single character are classified as UNKNOWN until they are recognized. A block of this type could be separated or merged in future iterations. Once a block is recognized, it is classified as HOLD. This prevents further attempts to separate or merge this block. Blocks that have been separated are classified as SEPARATE. This indicates that the branches that the system should look to the branches of this block for further processing. Finally a new block created by merging two older blocks is classified as MERGE. This indicates that its component segments are available should merging need to be undone.

Because of such merging and separation, the original linked list becomes a list of tree structures. In order to reduce complexity, an iterator is used to traverse the structure as if it were a list. Externally the data appears to be a string of segments organized as a linked list, but backtracking is simple because of the internal representation scheme used in this architecture.

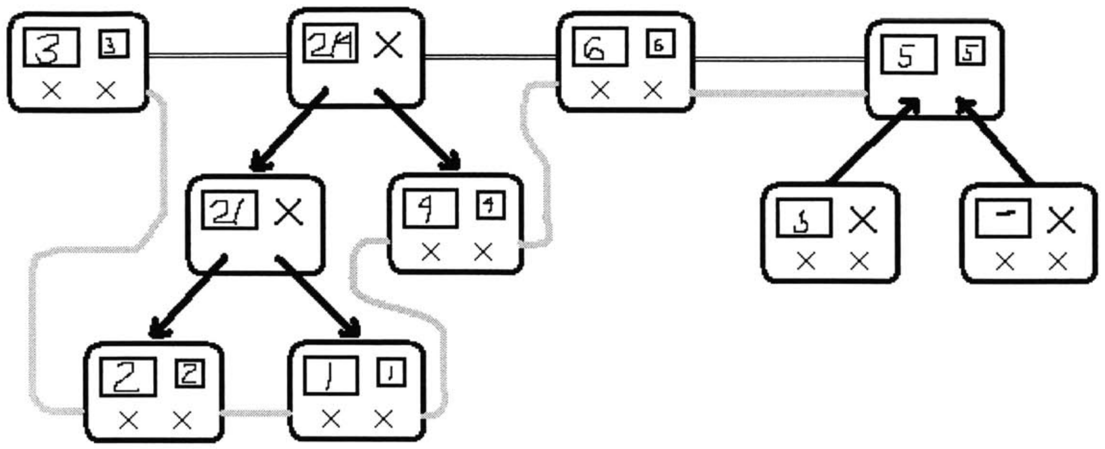


Figure 5.2: The Block Iterator

This representation is simplified when segments are recognized. If both child components of a segmented bock are recognized, there is no further need for the original box so the tree is flattened. Similarly if a merged block is recognized, its component blocks are deleted since there is no further need for them.

5.1.2 Re-Merging Implementation

In order to redo merging, it is important to identify blocks to which a fragment has already been merged. Otherwise, merging could be done repeatedly on the same block and result in failure. Because, separation and merging can change the data structure at each iteration of the feedback loop, the system must do more than just remember the relative position of previously merged blocks. Instead, each block is given a unique identification number (the block ID) and fragments keep a history of blocks to which they have already been merged. Note that separation and merging operations create new blocks, which have unique ID numbers, so a fragment can be merged with these blocks even if merging with their component segments has already been attempted.

Unmerged fragments will always be placed at the top level linked list of the tree-list structure. When merging is done, the system looks at the two list elements preceding the fragment and two elements following it in the linked list. Each list element can be a single block or a compound tree structure formed by previous separation or merging. Only one block from each list element is considered. If it is a single block or a merged block, the choice is obvious. If it is a separated block then one of the leaf nodes in the tree is selected for consideration. This selection process ignores blocks that have already been merged with the segment so other blocks in the tree may be considered in future iterations.

The fragment is merged with the best match, based on the criteria described above. If none of the candidate blocks is close enough to the fragment (maximum distance threshold) then the fragment is deleted. Otherwise it is merged with the best candidate. The new merged block replaces the block to which the segment was merged. The fragment and the segment blocks are relocated to the branches of this new block. If the resulting segment is not recognized, then the merge is undone in the next iteration of the feedback loop. The fragment is placed at the top level linked list again. It is placed immediately before or after the list entry to which it was joined, depending on what side of the segment it was joined to.

5.2 Iterative Training

A somewhat different training method was employed to improve the results of the recognition module. The method is based on an algorithm presented by Rowley [n1] for the purpose of face detection. The neural networks in Rowley's system were trained iteratively in a series of stages. At each training stage, errors from the previous stage were included in the training set for the current stage. In this way, common errors are reduced because the network. Rowley's method for training face detection networks is summarized below:

Iterative Training

1. Train the network using standard training set.
2. Test the resulting network on a different set, including randomly generated images as negative examples.
3. Identify cases where it makes an error, and retrain using these cases in the training set.
4. Repeat steps 2 and 3 until the desired accuracy is achieved

This algorithm was implemented with some modifications in the Winbank system. The existing networks were retrained using properly segmented characters that were not recognized. The method improved accuracy results drastically; however this is misleading because the testing set now contains characters used in training and is somewhat small in size. The networks could have "memorized" these cases rather than developing accuracy in the general case. The system was then tested on a completely distinct set of checks. In the original set the results went from 24 checks read correctly and 49 incorrect to 50 correct and 20 incorrect. The second set showed improvement from 7 correct and 36 incorrect to 13 correct and 18 incorrect. The second set showsimprovement similar to the original testing set. The improvements were significant but not as drastic as the improvement in the earlier scenario. Iterative training does indeed improve the accuracy of the system, although the results may be exaggerated when using the original testing set.

5.3 Slant correction

Winbank uses slant correction to normalize digits and improve the likely hood of recognition. However slant correction is a computationally expensive process; the entire image is copied over repeatedly with incremental changes in rotation. The proccess is time consuming and requires significant resources. Furthermore, it can cause incorrect recognition by overcorrecting digits. A common error is that connected zeros are read as heavily slanted eights.

The degree to which the image is slant corrected has been reduced. The new value is 45° on either side. This modification has been shown empirically to decrease errors while maintaining the rate of correct recognition. Reducing slant correction decreases overall error by 0.3% while the false positive rate is improved by 0.9%. This modification also improves the speed of the system because the number of caparisons needed is reduced. Test results show an average improvement of 6.1% in time. These results are explained in more detail in Section 6.2.

6. Performance Evaluation

The Winbank system was tested using several different techniques to determine the accuracy and speed of the system. This thesis focuses on the segmentation module and, specifically, segmentation algorithms such as Extended Drop Fall (EDF) and Hybrid Drop Fall (HDF). One set of tests was designed specifically to determine the performance of these algorithms. The scope of these tests is limited to those cases in which segments were properly identified as multiple digits and separation was attempted. Errors in CAB location, merging and recognition are not considered in these tests. The results of this set of tests are presented in Section 6.1.

The second set of tests, presented in Section 6.2, evaluates the accuracy and speed of the entire system. The ultimate goal of the system is to read checks rather than simply segment numbers. The EDF algorithm is implemented to increase overall performance via the segmentation module. Chapter 5 describes other modifications employed for the same purpose. In order to determine overall performance, the system was tested on several sets of check images taken from a real life application (i.e. images of checks that were actually issued). The tests were performed using Winbank in a number of different configurations, so that the results indicate the incremental improvement of each of the modifications as well as their cumulative effect. Because several of these modifications involve additional computation, the speed of the system is also tested. The results of these tests are presented in Section 6.2.

6.1 The Segmentation Algorithm

In order to gauge the accuracy of a segmentation algorithm, we must consider performance on a per segment basis, rather than on a per-check basis. Both the HDF algorithm used in the previous implementation of Winbank and the new EDF algorithm were tested using a set of 145 checks containing 910 digits. The segmentation algorithm was used to separate digits in 162 cases. The EDF algorithm was found to have consistently better results than HDF. The results are presented in Table 6.1.

	Hybrid Drop Fall			**Extended Drop Fall**		
	Total	Correct	%	Total	Correct	%
Overall	162	72	44.4%	162	94	58.0%
Connected Zeros	86	22	25.6%	86	44	51.2%
Other Cases	76	50	65.8%	76	50	65.8%

Table 6.1: Accuracy of Segmentation Algorithms

Note that over half (53%) the cases requiring segmentation involved connected zeros. This is typical of bank checks and is the key factor in the increased accuracy from HDF to EDF. The HDF algorithm properly segments only about a quarter of the connected zeros (25.6%). The EDF algorithm is specifically designed for this case and doubles the number of accurate separations. Furthermore performance on cases other than connected zeros is identical. However, performance on individual cases is not identical. EDF segments some cases that HDF does not, and vice versa.

In practice, the overall performance of the system is higher than the 44.4% and 58% listed in the tables because the recognizer is robust enough to handle digits that were not perfectly segmented. Only cases in which no flaws could be detected on manually inspection were considered correct. Although recognition by the system is a more practical indicator, it depends on the performance of the recognition module, which can bias the results. Using "perfect" segmentation as the criteria allows us to focus on the performance of just the segmentation algorithms.

6.2 Overall System Performance

The overall performance of the system is measured by examining accuracy per check, rather than per digit. The system was tested on several sets of authentic checks obtained from a Brazilian bank (because Winbank is optimized for syntax and grammar conventions used in Brazil.) The primary testing set contains 320 paper checks that were scanned in by hand using a flatbed scanner and stored as bitmapped images. In addition to the paper checks there are several sets, of approximately 300 images each, stored in a compressed JPEG format. The secondary sets are of relatively low quality; they contain an unusually high level of background noise and many segments are difficult to separate and recognize. The majority of the results discussed in this section use the primary set. The secondary sets are used to verify the results, particularly when using iteratively trained neural networks.

Table 6.2 summarizes test results that compare the most recent version of the program with the pre-existing Winbank system. The pre-existing ("Unmodified") system is compared with the most recent version of Winbank which implements all the modifications described in this thesis. The iterative training method described in Section 5.2 is treated separately because these results require further analysis to decide whether or not they represent the true accuracy of the system. (See the discussion in section 5.3). In order to differentiate the two cases the proceeding discussion "trained" will refers to the use of iterative training and untrained refers to the use of a standard neural network in the proceeding discussion.

	Untrained				Trained			
	Reject	Read	Error	False pos.	Reject	Read	Error	False pos.
Winbank Unmodified	76.9%	7.6%	15.5%	67.1%	77.9%	15.8%	6.3%	28.6%
Winbank Modified	78.1%	10.0%	11.9%	54.3%	73.0%	19.5%	7.5%	27.9%

Table 6.2: Overall Accuracy of the System

Table 6.2 lists the four most significant statistics for each test: the reject rate, the read rate, the error rate, and the false positive rate. Rejects rate indicates the percentage of checks that could not be recognized by the system; such checks will need to be read by a human operator. The read rate refers to the percentage of checks that were recognized correctly. Finally, the error rate is the percentage of all checks that were recognized incorrectly and the false positive rate is the percentage of recognized checks that were read incorrectly.

The modifications resulted in a marked improvement for both trained and untrained versions of the system. The untrained system rejects more cases but correctly reads a greater percentage of the checks and makes errors on a smaller percentage, when the modifications are included. As a result, the false positive rate is significantly lower at 54.3%. The trained version shows a 1.2% increase in the overall error rate when the modifications are implemented. However, this increase is outweighed by greater improvements in the correct-read rate, the false positive rate and the reject rate. The overall change in performance from the untrained, unmodified version of Winbank to the trained, modified version is considerable. The read rate has more than doubled, the error rate is less than half as much, and the false positive rate falls from 67.1% to 27.9%.

The results presented in Table 6.2 shows the cumulative effects of all the modifications performed on the system. It is useful to note the incremental improvement for each modification employed. Table 6.3 lists test results using independent implementation of each modification, so that the impact of each change can be considered individually. All tests were performed using the same set of 320 checks that was used in Table 6.2. The same set of iteratively trained neural networks was used for each test.

Reduce Slant	RBPS	EDF	Redo Merges	Reject Rate	Correct Read Rate	Overall Error Rate	False Positive Rate
				78.9%	13.2%	7.9%	37.3%
X				78.9%	13.2%	7.6%	36.4%
	X			83.0%	11.0%	6.0%	35.2%
		X		78.2%	13.6%	8.2%	37.7%
			X	78.5%	13.6%	7.9%	36.8%
	X	X		80.5%	12.6%	6.9%	35.5%
X	X	X		80.5%	13.5%	6.0%	30.6%

Table 6.3: Incremental impact of each modification to the system

EDF and RBPS indicate implementation of Extended Drop Fall and the Recognition based path selection, respectively, as described in Chapter 4. “Redo Merging” indicates that merges are reconsidered at every iteration of the feedback loop (see Section 5.1.) Finally, “Reduce Slant” indicates that the scope of slant correction has been narrowed down to eliminate rare false positives (see Section 5.3.)

Another important consideration is the speed of the system. Several of the mechanisms employed in the system are considered computationally expensive processes. Neural networks inherently entail many mathematical calculations. Similarly, slant correction makes numerous copies of the same image in order to minimize the aspect ration. The recognition based path selection method, used with EDF segmentation, forces the system to perform recognition, for up to eight cases, every time segmentation is attempted. This has a significant impact on the speed of the system, however, it still performs within previously established performance boundaries.

A human operator typically takes 5 to 10 seconds on average to process a check [13]. An automated system should operate well within these limits. The goal of this project has been to keep processing time below one second per check.

	Average time (sec. / check)	**Maximum time** (sec)	**Minimum time** (sec)
Unmodified System	0.33	0.80	0.11
Modified System	0.60	1.88	0.15
Optimal System	0.31	0.75	0.08

Table 6.4: Processing latency in the Winbank System

Three different versions of Winbank are presented in Table 6.4. Unmodified refers to the previously implemented system. The Modified configuration includes all modifications described in this paper including, EDF, RBPS, reduced slant correction, redoing merging, and use of iteratively trained networks. The Optimal configuration is designed to be most efficient in speed. It is simply the unmodified configuration with reduced slant correction, because this is the only modification that decreases time needed to process each check.

The Modified version is considerably slower than both the Unmodified and Optimal configurations. On average it takes twice as long to process a check. However, the average processing time of 0.6 seconds per check is still well within acceptable performance limits for the system. Although these modifications leave less flexibility for future modifications, the associated improvement in accuracy are certainly worth the added latency, especially since latency is still at acceptable levels.

7. Possible Improvements

Winbank is a sophisticated system with relatively good results in terms of performance and accuracy. However, further improvement may be possible with additional modifications. This section describes a set of modifications that should be considered for future research and further development of the Winbank system.

As noted in Chapter 2, segmentation is made easier if the system has *a priori* knowledge of exactly how many characters are present in the image. Although this is generally not possible with bank checks, the system may be able verify the number of segments distinguished using contextual information such as punctuation. Grammatical rules for American checks dictate that commas are used to separate three digits at a time and that only two digits follow a decimal point. In Brazil, the functions of the comma and the period are reversed, but the grammatical rules are similar. In either case the placement of punctuation in the image can be used to adjust the likelihood of segmentation. The Winbank system is somewhat restrictive in the early iterations of the feedback loop to prevent unnecessary segmentation. Punctuation can be used to segment the characters more aggressively. Segmentation criteria should be relaxed when punctuation suggests there should be more digits than detected and more restrictive if this is not the case. Other information, such as delimiters or horizontal lines, which are currently discarded, might also be used in a similar manner.

Another possible improvement is to discard similar paths. In general, it is difficult to determine whether or not two paths are similar. Even small deviations could make the difference between correct segmentation and failure. However, paths that make the same cuts are identical for purposes of segmentation. Checking paths for identical cuts and eliminating these duplicate paths could improve the performance of the system.

It may also be useful to extend connected pixel extraction beyond the boundaries of the CAB. Some digits tend to hang down below the lower border of the CAB. This is particularly common for digits with a long straight lower stroke such as “4”,”7” and “9.” By extending pixel extraction, these digits are not truncated and are more likely to be recognized. Similarly, segments that extend well outside the CAB may not be part of the numerical value of the check. Writing from the Legal Amount Block (where the value of the check is written out in text) frequently extends into the CAB. These fragments are often treated as punctuation or unrecognized digits by the segmentation module. By eliminating all fragments that extend beyond a threshold distance outside of the CAB, segmentation errors can be reduced.

Also, improvements can be made in the way Winbank deals with segments composed of more than two digits connected together. When a segment containing three or four characters is separated, the original segment is separated in two and the resulting segments are separated further as necessary. The result is a tree structure with the first separation as the root and further separations branching downward. Currently, separation lower in the tree is reconsidered more often than earlier separation.

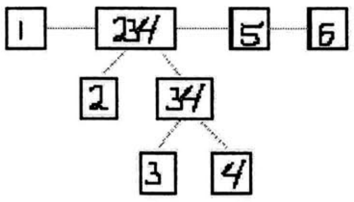


Figure 7.1: Multiple segmentations for a tree structure

Figure 7.1 illustrates an example. The "234" block is separated first into the "2" bock and the "34" block. The "34" block is then further separated. If the segments are not recognized, the first separation will not be reconsidered until all possibilities for the "34" block are exhausted. It might be more productive to reconsider the first separation, especially if the "2" block is not recognized.

A number of improvements can be made in the recognition module. The iterative training method described in Chapter 5 should be explored further. Iterative training should be done more rigorously, using more incorrectly recognized segments and more rounds of training. The networks should be trained using a larger training set and a number of distinct sets for testing the results. The results presented in this thesis reflect a limited use of iterative training, but they suggest that considerable improvements in accuracy can be achieved using this method.

The recognition module could also be improved by allowing a larger set of possible outputs. There are a number of symbols that the system is not currently trained to recognize. For example, the dollar sign ("$") was not included in the set of possible classifications because the system was designed for Brazilian checks in which "#' is a common delimiter. The delimiters "RS" and "R$" are also common in Brazilian check but the system does not recognize them. The current system will simply reject checks with these characters. Adding these cases will improve the accuracy and the versatility of the system. However, this requires a change to the network topology, so it may not be possible to reproduce current accuracy levels with the new characters included.

In addition to an expanded character set, it may be useful to have the recognizer find special cases such as double zeros or connected digits. Because segmentation is conservative on the first pass, connected digits are routinely passed to the recognition module several times before they are segmented. If the recognizer could indicate "connected digit" rather than simply "not recognized" the segmentation module can then force segmentation on the second pass regardless of the aspect ratio. Connected zeros are so common that it would be useful to recognize this case separately from connected digits. Once a region is identified as two connected zeros, there is no need for further segmentation because the value is already known.

Another possible improvement involves CAB location. Although CAB location works well in the majority of cases, the system will occasionally miss part of the CAB. The borders of the CAB are located by looking for straight horizontal or vertical lines. The straight stroke of the digits such as 1,9 and 7 can be "straight enough" to be mistaken for a vertical CAB border. In these cases any number digits to the left of the straight stroke will not be found. In theory, this can also happen with horizontal lines but, it is very unlikely that the system will find a straight horizontal stroke long enough to be mistaken for a border.

A possible solution to this type of CAB location problem is to consider a number of candidate lines for borders. Currently, the system begins at a hard coded starting point inside the CAB and scans outward looking for straight vertical and horizontal lines. When a straight line is found, the system stops scanning in that direction. An improved system could continue scanning and identify all possible lines or, at least, all possible vertical lines. The most likely CAB borders can then be determined using heuristic analysis with such factors as the aspect ratio of the CAB, the thickness of the lines detected, and the distance from other elements. Most checks will have CABs of roughly the same aspect ration. Brazilian checks typically have only thick vertical lines to delimit the CAB. There will be some whitespace between the handwritten characters and the CAB borders. The system can use all these properties to determine the CAB location in cases of uncertainty.

8. Conclusion

The modified segmentation module described in this paper is designed to overcome some of the weaknesses of heuristic analysis. The system also pursues many more possible options in order to select the correct division of pixels into characters. The performance of the module is much improved on several cases that were frequently segmented incorrectly using earlier techniques. Such cases were consistently segmented improperly because heuristic analysis techniques rate the correct segmentation paths lower than other options. The modified system works better because the reliance on heuristics has been reduced. The new system also unmerges and remerges to correct previous errors in merging. In addition improvements to the recognition module were included to allow better performance of the overall system.

The Extended Drop Fall algorithm, described in Chapter 4, works well for cases in which the correct path should make a long cut through the connected characters. It works particularly well for connected zeros, a case that occurs frequently and is routinely separated incorrectly. In order to utilize this algorithm, the segmentation module also incorporated recognition based path selection. The Extended Drop Fall approach adds a set of paths to the list of possible separations. However, heuristic analysis would rank these paths very low because short cuts are better than long cuts in the general case. By using a recognition-based approach, the system can determine when these new paths are better than the normal set because, ideally, only the correct path will result in recognition of both parts of the image.

Recognition based path selection is an improvement in its own right. It can improve accuracy results even without Extended Drop Fall. The new module shows definite improvement over the previous system in terms of accuracy. Although the module is significantly slower than the previous version, it still runs well within performance limitations, so the improvement in accuracy certainly out weighs the cost.

In the previous implementation of the segmentation module, merging was done only on the first pass through the system. This reduced complexity while still attempting to correct fragmented characters. The improved module can do merging on every pass through the loop. Like the new segmentation algorithm, merging now attempts more cases and catches a number of common errors.

These improvements are combined with a new training method that improves the accuracy of the recognition module. The result is a significant improvement in the overall performance of the system. Although, reading of all types of handwritten material with the Winbank system is still far from perfect, one has come closer to the goal of imitating human ability to read characters!

References

[1] Dey, S. (1999) *Adding Feedback to Improve Segmentation and Recognition of Handwritten Numerals.* Masters Thesis, Massachusetts Institute of Technology.

[2] Sinha, A. (1999) *An Improved Recognition Module for the Identification of handwritten Digits.* Masters Thesis, Massachusetts Institute of Technology.

[3] Sparks, P., Nagendraprasad, M.V., Gupta, A. (1992) " An Algorithm for Segmenting Handwritten Numeral Strings." *Second International Conference on Automation, Robotics, and Computer Vision.* Singapore, Sept. 16-18, 1.1.1 - 1.1.4.

[4] Congedo, G., Dimauro, G., Impedovo, S., Pirlo, G. (1995) "Segmentation of Numeric Strings." *Proceedings of the Third International Conference on Document Analysis and Recognition, Vol. II* 1038-1041.

[5] Dimauro, G., Impedovo, S., Pirlo, G., Salzo, A. (1997) "Automatic Bankcheck Processing: A New Engineered System." *International Journal of Pattern Recognition and Artificial Intelligence* 11 (4) 467-504.

[6] Blumenstein, M. and Verma, S. (1998) "A Neural Based Segmentation and Recognition Technique for Handwritten Words". *IEEE International Conference on Neural Networks* Vol. 3, 1738-1742.

[7] Lee, S.-W., Lee, D.-J., Park, H.-S. (1996) "A New Methodology for Gray-Scale Character Segmentation and Recognition." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16 (10) 1045-1050.

[8] Hong, T., Hull, J., Srihari, S. (1996) "A Unified Approach Towards Text Recognition." *Proceedings of SPIE: The International Society for Optical Engineering* Vol. 2660 27-36.

[9] Martin, G., Mosfeq, R., Pittman, J. (1993) "Integrated Segmentation and Recognition Through Exhaustive Scans or Learned Saccadic Jumps." *International Journal of Pattern Recognition and Artificial Intelligence* 7 (4) 831-847.

[10] Feliberti, V. and Gupta, A. (1991) *A New Algorithm For Slant Correction of Handwritten Characters*, Masters Thesis, Massachusetts Institute of Technology.

[11] Nagendraprasad, M.V., Wang, P.S.P., Gupta, A. (1993) "Algorithms for Thinning and Rethickening Binary Digital Patterns." *Digital Signal Processing* 3, 97-102.

[12] Sparks, P. (1992) *A Hybrid Method for Segmenting Numeric Character Strings.*

[13] Khan, S. (1998) *Character Segmentation Heuristics for Check Amount Verification.* Masters Thesis, Massachusetts Institute of Technology.

[14] Nagendraprasad, M., Sparks, P., Gupta, A. (1993) "A Heuristic Multi-Stage Algorithm for Segmenting Simply Connected Handwritten Numerals." *The Journal of Knowledge Engineering & Technology* 6 (4) 16-26.

[15] Zhao, B., Su, H.,Xia S. (1997) "A New Method of Segmenting Unconstrained Handwritten Numeral Strings." *Proceedings of the Fourth International Conference on Document Analysis and Recognition.* (ICDAR '97) Vol. II 524-527

[16] The Green Sheet, Inc. *United States Check Study:Results of 1997 Research,* (Pentaluma, CA Published Report,1998)

[17] Martin, G. (1993) "Centered-Object Integrated Segmentation and Recognition of Overlapping Handprinted Characters." *Neural Computation* 5, 419-429.

[18] Keeler, J., Rumelhart, D. (1992) "A Self Organizing Integrated Segmentation and Recognition Neural Network." *Proceedings of SPIE: The International Society for Optical Engineering* Vol. 1710 744-755.

[19] Shridar, M. and Badrelin, A. "Recognition of Isolated and Simply Connected Handwritten Numerals." Pattern Recognition. 19(1). 1986.

[20] R. Fenrich and K. Krishnamoorthy.(1990) "Segmenting diverse quality handwritten digit strings in near real-time." *Proceedings of the 4th Advanced Technology Conf.* 523-580

[21] N.W. Strathy, C. Y. Suen and A. Kryzak. "Segmentation of handwritten digits using contour features". *Second Int. Conf. On Document Analysis and Recognition.* Montreal, August 14-16, 1995, pp.1038-1041

[22] R. G. Casey, E. Lecolinet.(1996) "A Survey of Methods and Strategies in Character Segmentation." IEEE transactions on pattern analysis and machine intelligence. 18(7), pp.690

[23] Rowley, H. *A Trainable View-Based Object Detection System.* Thesis Proposal, Carnegie Mellon University. 1996

Appendix A: Segmentation Code

This section contains the source code for the segmentation module. Only the major functions are reproduced here. Higher level functions are presented firsty then the more detailed functions. The reader should be able to follow the progression of the code. Descriptions of all functions and datastructures can be found in the header files, which are listed in Appendix B.

```
// Function:     SegmentAndRecognize
//
// Purpose:              Iterates through segmentation and recognition until nothing else
//                               can be done.
//
// Arguments:    [in] image                 B/W image to be segmented and recognized
//
// Returns:              NULL if an error occurred.  Otherwise returns a set of processed
//                               segments.
// Notes:
//
CABValue* SegmentAndRecognize(BWImage* image, DDDNet* nn1, PosNet* nn2, DDDNet* nn3,
PosNet* nn4,
                                                        StructProcessor* sp, Arbiter* arb)
{
        CABValue* CAB = new CABValue;
        if (!CAB) return NULL;

        // Separate out all of the connected regions in the image and classify them.
        if (CAB->CollectConnectedRegions(image) == SEG_ERROR) {
                delete CAB;
                return NULL;
        }

        // Write the initial CAB to the log
        WriteLog(0,
"*******************************************************************************
********");
        WriteLog(0, "INITIAL CONNECTED REGIONS");
        WriteLog(0,
"*******************************************************************************
********");
        CAB->Log(0);
        char itermessage[100];     // A handy variable for logging
        strcpy(itermessage, "Iteration: ");

        // Remove all "isolated" characters, which are probably side effects of bad CAB location
        CAB->RemoveIsolatedRegions(image);


        // Iterate through segmentation and recognition until everything is recognized
        // or until we go through enough times.
        int iteration_counter = 0;
        while (!CAB->DoneRecognizing() && iteration_counter < MAX_ITERATIONS) {
                iteration_counter++;

                // Adjust classifications to catch mistakes.  This is the meat
```

```
			// of the feedback system.
			CAB->AdjustClassifications(iteration_counter);

			// Write the current CAB to the log
			WriteLog(0,
"*****************************************************************************************
********");
			sprintf(itermessage+11, "%d", iteration_counter);
			WriteLog(0, itermessage);
			WriteLog(0,
"*****************************************************************************************
********");
			CAB->Log(0);

			// Evaluate all unrecognized segments and make changes as necessary.
			CAB->DoSegmentation(nn1, nn2, nn3, nn4, sp, arb);

			// Normalize segment sizes, thicknesses, slants, etc.
			CAB->PrepareForRecognition();

			// Attempt to recognize all unrecognized segments
			for (Segment* seg = CAB->IterateSegments(true, true);
				seg != NULL;
				seg = CAB->IterateSegments(false, true)) {

				// We filter out segments which are classified "NONSENSE" and "HORIZ"
				// because they are most likely the result of a bad segmentation.
//				if (!seg->ClassIs(SC_NONSENSE)  && !seg->ClassIs(SC_HORIZ)) {
				if (seg->ClassIs(SC_NONSENSE)  || seg->ClassIs(SC_HORIZ)
					|| seg->ClassIs(SC_FRAGMENT) || seg-
>ClassIs(SC_FRAGMENT_ERROR)) {
					seg->digit = NOT_RECOGNIZED;
				} else {
					if (!g_isGUI) {
						Segment* sseg=RecognizeSeg(nn1, nn2, nn3, nn4, sp, arb,
seg);
						//if(sseg==NULL){WriteLog(0,"RecognizeSeg returned
null");}
						//else{WriteLog(0,"RegonizeSeg valid");}
					}
					else{
					if (g_gatherSegs) {
						HumanRecognize(seg);	// TEST RECOGNIZER
					} else {
						RecognizeSeg(nn1, nn2, nn3, nn4, sp, arb, seg);
						seg->LogConfidences(3);
					}
					}
				}
			}

			// Get rid of old data we don't need anymore.
			CAB->CleanupAfterRecognition();
		}

		// Finalize everything, recognized or not
```

```
	CAB->CleanupWhenDone();

	// Write the final CAB to the log
	WriteLog(0,
"*****************************************************************************
********");
	WriteLog(0, "FINAL RESULT");
	WriteLog(0,
"*****************************************************************************
********");
	CAB->Log(0);

	return CAB;
}
```

```
// Function:	DoSegmentation
//
// Purpose:		Performs any necessary separations and merges.
// Arguments:

void CABValue::DoSegmentation(DDDNet* nn1, PosNet* nn2,DDDNet* nn3, PosNet* nn4,
StructProcessor* sp, Arbiter* arb)
{
	Segment* seg = NULL;
	bool first;
	Block* block = NULL;
	bool changed;

	//Undo Merges. Each time through undo unsucessfull merges and merge to other candidates
	if(g_redoMerging){UndoMerging();}

	// Each time through this loop we look at the list of regions and make
	// one change to get rid of one character classified as fragmented or
	// multiple.  (This is because once a change has been made we should start
	// over looping from the beginning.)  We keep looping until no more changes
	// are made.
  /**/
	// MERGING LOOP
	do {
		changed = false;

		// Attempt to paste a fragment to its most reasonable neighbor.
		for (block = head; (!changed) && (block != NULL); block = block->next) {
			first = true;
			while (!changed && ((seg = block->IterateSegments(first)) != NULL)) {
				first = false;
//				if (seg->ClassIs(SC_FRAGMENT) || seg->ClassIs(SC_HORIZ) || seg-
>ClassIs(SC_PUNCT)) {
				if (seg->ClassIs(SC_FRAGMENT) || seg->ClassIs(SC_HORIZ)) {
					WriteLog(0, "MERGE");
					if (Merge(block, seg) == SEG_OK) {
						WriteLog(0, "MERGE SUCCEEDED");
						changed = true;
					} else if (seg->ClassIs(SC_FRAGMENT)) {
						WriteLog(0, "MERGE FAILED");
						seg->SetClass(SC_FRAGMENT_ERROR);
```

```
                                        } else {
                                                // FIXLATER: SHOULD DO SOMETHING HERE
to keep HORIZ and PUNCT
                                                // segments from being merged (unsuccessfully) more
than once...
                                                WriteLog(0, "MERGE FAILED -- horiz or punct");
                                        }
                                }
                        }
                }

                // Do nothing for other cases.

        } while (changed);
/**/

        // SEPARATION LOOP
        do {
                changed = false;

                // Process all multiple-character regions
                for (block = head; (!changed) && (block != NULL); block = block->next) {
                        // Attempt to separate multiple characters.
                        first = true;
                        while (!changed && ((seg = block->IterateSegments(first)) != NULL)) {
                                first = false;
                                if (seg->ClassIs(SC_MULTIPLE)) {
                                        WriteLog(0, "SEPARATE");
                                        if (Separate(nn1, nn2, nn3, nn4, sp, arb, block, seg) ==
SEG_OK) {
                                                WriteLog(0, "SEPARATE SUCCEEDED");
                                                changed = true;
                                        } else {
                                                WriteLog(0, "SEPARATE FAILED");
                                                seg->SetClass(SC_MULTIPLE_ERROR);
                                        }
                                }
                        }
                }

        } while (changed);

}

// Function:    (CABValue::)Separate
//
// Purpose:             Separate segment seg within block into two different parts.
//                              This function is really just a wrapper for Block::Separate,
//                              with the addition that it maintains the head and tail pointers.
//
// Arguments:   [in]  block             Block containing segment to be separated
//                              [in]  seg               Segment to be separated
//                              [in]  nn1,nn2,nn3 Neural net recognizers
//                              [in]  sp  A structural Processor
//                              [in]  arb The arbiter to decide between networks
//
```

```
// Returns:                    SEG_ERROR if an error occurred or if separation was
//                                     unsuccessful.
//                                 SEG_OK if separation was successful.
//
// Notes:
//
Status CABValue::Separate(DDDNet* nn1, PosNet* nn2,DDDNet* nn3, PosNet* nn4, StructProcessor*
sp, Arbiter* arb, Block* block, Segment* seg)
{
        // Perform the separation
        if (block->Separate(nn1, nn2, nn3, nn4, sp, arb, seg, vmid) == SEG_ERROR)
                return SEG_ERROR;

        // Reset iterator for safety
        iter_loc = NULL;

        return SEG_OK;
}


// Function:        (Block::)Separate
//
// Purpose:                    This is the real meat of the segmentation. Separate two (or
//                                 more) joined characters into two distinct connected regions, so
//                                 that one is a single, complete character. Replace the old segment
//                                 in RegionList with two separate segments. Uses vmid to
//                                 critique segmentation done.
//
//                                 Maintains the classification for each segment.
//                                 Makes no changes to image.
//
// Arguments:       [in] s             Segment being separated
//                                 [in] vmid          Vertical midline of image
//                                 [in] nn1,nn2,nn3 Neural net recognizers
//                                 [in] sp   A structural Processor
//                                 [in] arb  The arbiter to decide between networks
//
// Returns:                    SEG_ERROR if an error occurred or if separation was
//                                     unsuccessful.
//                                 SEG_OK if separation was successful.
//
// Notes:
//
Status Block::Separate(DDDNet* nn1, PosNet* nn2,DDDNet* nn3, PosNet* nn4, StructProcessor* sp,
Arbiter* arb, Segment* s, int vmid)
{
        // Find the block that s came from
        if (s != seg) {
                if ((type != BT_SEPARATE) || (!bList1) || (!bList2))
                        return SEG_ERROR;
                Status result = bList1->Separate(nn1, nn2, nn3, nn4, sp, arb, s, vmid);
                if (result == SEG_OK)
                        return result;
                return (bList2->Separate(nn1, nn2, nn3, nn4, sp, arb, s, vmid));
        }
```

```
	Path* path;

	// Get the paths.
	if (!paths) {
		// First perform contour analysis.
		path = GetContourPath(vmid);
		if (path) AddPath(path);
		else WriteLog(0, "Countour Failed");
		if (path) LogPath(3);  // REMOVE LATER

		// Then do a hybrid drop-fall analysis.
		for (int dir = 0; dir < 4; dir++) {
			path = GetDropFallPath(dir, vmid, SEG_DROP_FALL_NORMAL_CASE);
			if (path) AddPath(path);
			if (path) LogPath(3);  // REMOVE LATER
		}

		// Then do an Extended drop-fall analysis.
		if (g_useEDF){
			for ( dir = 0; dir < 4; dir++) {
				path = GetDropFallPath(dir, vmid,
SEG_DROP_FALL_SPECIAL_CASE);
				if (path) AddPath(path);
				if (path) LogPath(3);  // REMOVE LATER
		}

		}
		// Rank the paths.
		RankPaths(vmid);
	}

	//char* rr= new char[100];
	//sprintf(rr,"+++%d,%d",g_segmentByRecognizing, numpaths);
	//WriteLog(0,rr);


	bool success=false;
	if (g_segmentByRecognizing){


		//Get the best path (segments that will be recognized)
		Path* bestPath=SegmentByRecognizing(nn1, nn2, nn3, nn4, sp, arb, seg);
		if (bestPath){
			success = (SeparateRegionAlongPath(bestPath) == SEG_OK);
		}

		//free memeory alocated for paths
		while ((numpaths > 0) && !success) {
			numpaths--;
			delete paths[numpaths];
			paths[numpaths] = NULL;
		}

	}
	else{
```

```
//		LogPath(3);

		// Divide along the best path.  (the best path is the highest-numbered one)
		//bool success = false;
		while ((numpaths > 0) && !success) {
			numpaths--;
			success = (SeparateRegionAlongPath(paths[numpaths]) == SEG_OK);
			delete paths[numpaths];
			paths[numpaths] = NULL;
		}
	}///matches if(g_SegmentBYRecognizing)


	// If we had no successful paths, flag an error
	if ((numpaths == 0) && !success) {
		return SEG_ERROR;
	}

	// Otherwise, separation was successful, so set the proper classifications
	type = BT_SEPARATE;
	if (success) {
		// Just one block in each list here
		bList1->seg->Classify(vmid);
		bList2->seg->Classify(vmid);
	}

	WriteLog(0, "RESULT:");
	bList1->Log(6);
	bList2->Log(6);

	// Reset iteration for safety
	iter_loc = NULL;

	return SEG_OK;
}

// Function:		(Block::)SegmentByRecognizing
//
// Purpose:			Selects one of several segmentation paths by attempting
//					to recognize the two resulting segments. Assumes paths have already
//	been generated and ranked. Returns the firsty path that recognizes
//					both segments. If no such path exists returns the firs path that
//					results in recognition of one segment. If no sergments are
//					recognized for any path the first path is returned
//
//
//					Maintains the classification for each segment.
//					Makes no changes to image or segment.
//
// Arguments:	[in]  seg		Segment being separated
//					[in]  nn1,nn2,nn3 Neural net recognizers
//					[in]  sp   A structural Processor
//					[in]  arb  The arbiter to decide between networks
//
// Returns:			NULL if an error occurred or if separation was
//						unsuccessful. The most sucessful Path* otherwise
```

```
//
// Notes:              Allocates an array of Blocks so that the Cabvlue is unaffected by
//                                  function. cleans up afterward. Assumes Paths have already been
//                                  generated and ranked

Path* Block::SegmentByRecognizing(DDDNet* nn1, PosNet* nn2,DDDNet* nn3, PosNet* nn4,
StructProcessor* sp, Arbiter* arb, Segment* seg){

        if (!paths||numpaths==0) return NULL;//sanity check
        //Allocate a new array of blocks used to test the paths
        Block** blocks= new Block*[numpaths];
        if (!blocks){return NULL;}

        WriteLog(0,"######################################################################
###");
        //create each block with the current segment
        //seperate along diferent paths, and recognize
        int oneRecognized=-1;
        int twoRecognized=-1;
        bool edfPath;
        for(int i=0;i<numpaths;i++){


                edfPath=(paths[i]->type==PT_EDF_TOPLEFT)|| //are we using an EDF path
                        (paths[i]->type==PT_EDF_TOPRIGHT)||
                        (paths[i]->type==PT_EDF_BOTTOMLEFT)||
                        (paths[i]->type==PT_EDF_BOTTOMRIGHT);

                if(twoRecognized==-1||edfPath){

                        blocks[i]=new Block(seg, true);
                        Segment* seg1;
                        Segment* seg2;

                        if(blocks[i]->SeparateRegionAlongPath(paths[i])==SEG_OK){

                                char* ss= new char[100];
                                blocks[i]->type=BT_SEPARATE;
                                blocks[i]->PrepareForRecognition();
                                seg1=blocks[i]->bList1->seg_norm;
                                seg2=blocks[i]->bList2->seg_norm;

                                if (!seg1 || !seg2){continue;}

                                sprintf (ss,"Hello %d -> %d, %d", seg1->digit, numpaths, i);

                                WriteLog(0,ss);
                                RecognizeSeg (nn1, nn2, nn3, nn4, sp, arb,seg1);

                                RecognizeSeg (nn1, nn2, nn3, nn4, sp, arb,seg2);
                                //if(sseg==NULL){WriteLog(0,"Recognize seg returned null");}

                                blocks[i]->Log(3);
                                //blocks[i]->Log(5);
                                if(seg1->digit !=NOT_RECOGNIZED && seg2->digit
!=NOT_RECOGNIZED){
```

```
                                        //Both recognized
                                        WriteLog(0,"-->Both recognized");
                                        if (edfPath && seg1->digit==ZERO && seg2-
>digit==ZERO) {

                                                return paths[i];
                                        }else{
                                                if (twoRecognized==-1){twoRecognized=i;}

                                        }
                                }

                                if ((oneRecognized==-1) &&
                                        (seg1->digit !=NOT_RECOGNIZED ||seg2->digit
!=NOT_RECOGNIZED)){

                                        //record the first path for which one segment is recognized

                                        WriteLog(0,"-->One recognized");
                                        oneRecognized=i;
                                }

                        }//matches if (paths[i]-type....

                }//matches for (int i=0...

                //cout<<"\n";

        }
        WriteLog(0,"#######################################################################
###");
        if (twoRecognized !=-1){ return paths[twoRecognized];}
        if (oneRecognized !=-1){ return paths[oneRecognized];}
        else {return paths[0];}

}
```

```
Path* ConstructPath(Segment* region, Point start, int direction, bool special_case)
{
        // Set up the new path
        Path* path = new Path(region->height);
        if (!path) {
                // Out of memory
                return NULL;
        }

        // Assign it the correct type
        if (special_case){
                switch (direction) {
                case SEG_FALL_TOPLEFT:
                        path->type = PT_EDF_TOPLEFT;
                        break;
                case SEG_FALL_TOPRIGHT:
                        path->type = PT_EDF_TOPRIGHT;
                        break;
```

```
		case SEG_FALL_BOTTOMLEFT:
			path->type = PT_EDF_BOTTOMLEFT;
			break;
		case SEG_FALL_BOTTOMRIGHT:
			path->type = PT_EDF_BOTTOMRIGHT;
			break;
		default:
			path->type = PT_UNKNOWN;
		}
	}else{
		switch (direction) {
		case SEG_FALL_TOPLEFT:
			path->type = PT_DROP_TOPLEFT;
			break;
		case SEG_FALL_TOPRIGHT:
			path->type = PT_DROP_TOPRIGHT;
			break;
		case SEG_FALL_BOTTOMLEFT:
			path->type = PT_DROP_BOTTOMLEFT;
			break;
		case SEG_FALL_BOTTOMRIGHT:
			path->type = PT_DROP_BOTTOMRIGHT;
			break;
		default:
			path->type = PT_UNKNOWN;
		}
	}

	// Now find the drop-fall path which divides the character into two pieces.
	// Whatever is on the path or falls to the left of the path is in the first piece;
	// anything to the right is in the second piece.  If there is filled-in space above
	// (or below, depending on drop-fall direction) the start point, move around it.

	Point current;

	// Start with a straight line above and below the start point.
	for (current.row = 0; current.row < region->height; (current.row)++)
		path->AddPoint(current.row, start.col);

	// Perform the drop-fall.
	current.row = start.row;
	current.col = start.col;
	Point prev;
	if ((direction == SEG_FALL_TOPLEFT) || (direction == SEG_FALL_TOPRIGHT)) {
		// Do one move at a time until we reach the bottom.
		while ((current.col >= region->bbox.left) && (current.col <= region->bbox.right) &&
			  (current.row < region->height - 1)) {
			prev.row = current.row;
			prev.col = current.col;
			if (DropFallOneMove(region, &current, direction, special_case) ==
SEG_ERROR) {
				delete path;
				return NULL;
			}

			// Special case to make sure we don't get stuck in "flat" minima
```

```
            if (prev.row == current.row) {
                if (direction == SEG_FALL_TOPLEFT) {
                    if ((current.col + 1 <= region->bbox.right) &&
                        (region->pixel[current.row][current.col + 1] == 1)) {
                        // We have already been at this corner; cut
downwards.
                        (current.row)++;
                    } else  if (current.col == region->bbox.right) {
                        // We pushed all the way to the side of the segment --
make a cut
                        (current.row)++;
                    }
                } else if (direction == SEG_FALL_TOPRIGHT) {
                    if ((current.col - 1 >= region->bbox.left) &&
                        (region->pixel[current.row][current.col - 1] == 1)) {
                        // We have already been at this corner; cut
downwards.
                        (current.row)++;
                    } else  if (current.col == region->bbox.left) {
                        // We pushed all the way to the side of the segment --
make a cut
                        (current.row)++;
                    }
                }
            }

            // If we moved, record in the path
            if (prev.row < current.row) {
                Point* p = path->GetPoint(current.row);
                p->col = current.col;
            }
        }

        // If we didn't reach the bottom, an error occurred.
        if (current.row < region->height - 1) {
            delete path;
            return NULL;
        }

    } else if ((direction == SEG_FALL_BOTTOMLEFT) || (direction ==
SEG_FALL_BOTTOMRIGHT)) {
        // Do one move at a time until we reach the top.
        while ((current.col >= region->bbox.left) && (current.col <= region->bbox.right) &&
             (current.row > 0)) {
            prev.row = current.row;
            prev.col = current.col;
            if (DropFallOneMove(region, &current, direction, special_case) ==
SEG_ERROR) {
                delete path;
                return NULL;
            }

            // Special case to make sure we don't get stuck in "flat" maxima
            if (prev.row == current.row) {
                if (direction == SEG_FALL_BOTTOMLEFT) {
                    if (current.col == region->bbox.right) {
```

```
                                        // We pushed all the way to the side of the segment --
make a cut
                                        (current.row)--;
                                } else if ((current.col + 1 <= region->bbox.right) &&
                                            (region->pixel[current.row][current.col +
1] == 1)) {
                                        // We have already been at this corner; cut upwards.
                                        (current.row)--;
                                }
                        } else if (direction == SEG_FALL_BOTTOMRIGHT) {
                                if (current.col == region->bbox.left) {
                                        // We pushed all the way to the side of the segment --
make a cut
                                        (current.row)--;
                                } else if ((current.col - 1 >= region->bbox.left) &&
                                            (region->pixel[current.row][current.col -
1] == 1)) {
                                        // We have already been at this corner; cut upwards.
                                        (current.row)--;
                                }
                        }
                }

                // If we moved, record in the path
                if (prev.row > current.row) {
                        Point* p = path->GetPoint(current.row);
                        p->col = current.col;
                }
        }

        // If we didn't reach the top, an error occurred.
        if (current.row > 0) {
                delete path;
                return NULL;
        }
} else {
        // Invalid direction
        delete path;
        return NULL;
}

// Adjust the path score based on algorithm-specific criteria
AdjustPathScore(path, region);

return path;
}


// Function:    DropFallOneMove
//
// Purpose:             Move from the current point to the next point in the region according to
//                      the drop-fall algorithm for this direction.
//
//                      This function embodies the rules of the drop-falling algorithms.
//
// Arguments:   [in]    region          Region moving through
//                                  [in/out] current    Location to begin drop-fall
//                                  [in]     direction  Direction of drop-fall (SEG_FALL_TOPLEFT,
//
        SEG_FALL_TOPRIGHT, SEG_FALL_BOTTOMLEFT,
//
        SEG_FALL_BOTTOMRIGHT)
//
// Returns:                 SEG_ERROR if an error occurred or a new point could not be found.
//                                          (current may be modified in this case)
//                                  SEG_OK if the move was successful.
// Notes:
//
Status DropFallOneMove(Segment* region, Point* current, int direction, bool special_case)
{
        // Define the array of neighbors
        Point cell[8] = {
                -1, 0,
                -1, 1,
                 0, 1,      // 701
                 1, 1,      // 6 2
                 1, 0,      // 543
                 1,-1,
                 0,-1,
                -1,-1
        };

        // This chunk of code might seem confusing, but its purpose is to "reverse"
        // all directions so that we can always think of our drop-fall as coming from
        // the top left.  This simplifies the code a lot (no dumb replication) and
        // makes it easier to comprehend (once you get past this following chunk).
        // Previously some people have implemented drop-falls other than the top left
        // variety by flipping the array being scanned.  Here we leave the array alone,
        // but we pretend that left is right, or that up is down, or both.

        int right, downRt, down, downLt, left;        // directions -- indices into the cell array
        bool rowles, colgrt, colles;                  // tests to see if the row & col are within bounds
        switch (direction) {
        case SEG_FALL_TOPLEFT:
                right = 2; downRt = 3; down = 4; downLt = 5; left = 6;
                colgrt = (current->col > 0);
                rowles = (current->row + 1 < region->height);
                colles = (current->col + 1 < region->width);
                break;
        case SEG_FALL_TOPRIGHT:
                // left & right are switched
                right = 6; downRt = 5; down = 4; downLt = 3; left = 2;
                colgrt = (current->col + 1 < region->width);
                rowles = (current->row + 1 < region->height);
                colles = (current->col > 0);
                break;
        case SEG_FALL_BOTTOMLEFT:
                // up & down are switched
                right = 2; downRt = 1; down = 0; downLt = 7; left = 6;
                colgrt = (current->col > 0);
                rowles = (current->row > 0);
                colles = (current->col + 1 < region->width);
                break;
        case SEG_FALL_BOTTOMRIGHT:
                // everything is switched
                right = 6; downRt = 7; down = 0; downLt = 1; left = 2;
                colgrt = (current->col + 1 < region->width);
                rowles = (current->row > 0);
                colles = (current->col > 0);
                break;
        default:
                // bad direction
                return SEG_ERROR;
        }

// Macros to make the code below simpler
#define WHITE(i)        !region->pixel[current->row + cell[i].row][current->col + cell[i].col]
#define GO(i)           current->row += cell[i].row; current->col += cell[i].col; return SEG_OK

#define BLACK(i)        region->pixel[current->row + cell[i].row][current->col + cell[i].col]
#define ONBLACK                 region->pixel[current->row][current->col]

        // Lookup table of drop-falling rules for single steps.
        // See BWImage::FindNextContourPixel() for explanations of abbreviations
        // and for beautiful ASCII art.

        if (!special_case||!ONBLACK){
                ///This is the standard drop fall
                //special cases are handled in the else clause

                // Signal if we can't go any farther down.
                if (!rowles)
                        return SEG_ERROR;

                // First look down
                if (WHITE(down))                        {       GO(down);       }

                // Down & right
                if (colles && WHITE(downRt))    {       GO(downRt);     }

                // Down & left
                if (colgrt && WHITE(downLt))    {       GO(downLt);     }

                // Directly right
                if (colles && WHITE(right))             {       GO(right);      }

                // Directly left
                if (colgrt && WHITE(left))              {       GO(left);       }

                // Otherwise we are stuck in a hole - cut downward!
                GO(down);
        }
        else{
                //Special case:
                // If we are on a black pixel we are making a cut
                // extend the cut by staying in black
                // This is a bad idea in genertal but it may help in casess such as connected zeros
```

```
			// Signal if we can't go any farther down.
			if (!rowles)
				return SEG_ERROR;

			// First look down
			if (BLACK(down))				{		GO(down);		}

			// Down & right
			if (colles && BLACK(downRt))	{		GO(downRt);	}

			// Down & left
			if (colgrt && BLACK(downLt))	{		GO(downLt);	}

			// Do Not go Directly right or left
			// terminate the cut if all spaces below are white
			//if (colles && WHITE(right))		{		GO(right);		}

			//if (colgrt && WHITE(left))		{		GO(left);		}

			// end of cut
			GO(down);

		}
}
```

```
Status CABValue::Merge(Block* block, Segment* seg)
{
	Block* newBlock = block->Merge(seg, vmid);
	if (newBlock == NULL)
		return SEG_ERROR;

	// Adjust the head and tail pointers.
	for (head = newBlock; head->prev; head = head->prev);
	for (tail = newBlock; tail->next; tail = tail->next);

	return SEG_OK;
}


// Function:		(Block::)Merge
//
// Purpose:			Merges the segment (which is within this block) with that of
//					the most logical other block within the list.  Replaces
//					the two blocks with one composite block.
//
// Arguments:	[in]  s		Segment to be merged
//					[in]  vmid		Vertical midline of image
//
// Returns:			NULL if an error occurred or if merge was unsuccessful.
//					Otherwise returns a pointer to the new block.
//
// Author:	Susan Dey	1/19/99
//
// Notes:
//
```

```
Block* Block::Merge(Segment* s, int vmid)
{
	// Make sure s is within this block
	// It ALWAYS should be there... Right?
	/////////////JIBU (additional comments)
	///s==seg because BT_Seperate blocks do not contain fragments in subblock
	///Seperate does not allow fragments broken off. If this is no longer the
	///case then s may be in a subblock and s != seg
	if (s != seg)
		return NULL;

	LogPath(3);

	// Choose a block to merge with.
	// This algorithm assumes that you will want to merge with blocks no
	// farther than 2 away in the list.  It does not look farther away in the
	// list than that.

	//bool usePrev = true;
	//bool usePrev2 = true;
	//bool useNext = true;
	//bool useNext2 = true;

	Block* prev1=NULL;
	Block* prev2=NULL;
	Block* next1=NULL;
	Block* next2=NULL;


	Block* vertOverlap = NULL;
	Block* closest = NULL;
	double minDist;



	//
	// Test 1:  Evaluate based on type.
	//

	// Don't merge with recognized digits
	// or with blocks it's already been merged to
	//if (!prev || (prev->type == BT_HOLD)||AlreadyMerged(prev->GetBlockID()))
	//		usePrev = false;
	//if (!prev || !(prev->prev) || (prev->prev->type == BT_HOLD)||AlreadyMerged(prev->prev-
>GetBlockID()))
	//		usePrev2 = false;
	//if (!next || (next->type == BT_HOLD)||AlreadyMerged(next->GetBlockID()))
	//		useNext = false;
	//if (!next || !(next->next) || (next->next->type == BT_HOLD)||AlreadyMerged(next->next-
>GetBlockID()))
	//		useNext2 = false;

	if (prev && (prev->type != BT_HOLD)&&!AlreadyMerged(prev->GetBlockID()))
		prev1= GetMergeCandidate(prev);
	if (prev && (prev->prev) && (prev->prev->type != BT_HOLD)&& !AlreadyMerged(prev->prev-
>GetBlockID()))
		prev2= GetMergeCandidate(prev->prev);
	if (next && (next->type != BT_HOLD)&& !AlreadyMerged(next->GetBlockID()))
		next1= GetMergeCandidate(next);
	if (next && (next->next) && (next->next->type != BT_HOLD) && !AlreadyMerged(next->next-
>GetBlockID()))
		next2= GetMergeCandidate(next->next);



	// Check to see if we have any options remaining
	//if (!usePrev && !usePrev2 && !useNext && !useNext2)
	//		return NULL;

	// Check to see if we have any options remaining
	if (!prev1 && !prev2 && !next1 && !next2)
		return NULL;


	//
	// Test 2:  Evaluate based on vertical overlap.
	//

	// FIXLATER: Try to incorporate vertical overlap with other classes?

	if (seg->ClassIs(SC_HORIZ)) {

		// Evaluate based on vertical overlap.  Make sure the other regions overlap
		// enough, vertically.  If one does, assign vertOverlap to point to that
		// region.  In doing this we assume that we will assign vertOverlap only once,
		// which is not unreasonable because it would be very unlikely for a horizontal
		// to be near the center of more than one other region.  (And they could be
		// merged on the next time around if necessary.)

		int centerCur = (int) ((seg->pos.left + seg->pos.right) / 2.0);
		int centerOther;

		// This order (prev2, next2, prev, next) is used to bias the results.
		if (prev2) {
			centerOther = (int) ((prev2->seg->pos.left + prev2->seg->pos.right) / 2.0);
			if (abs(centerOther - centerCur) < SEG_MERGE_HORIZ_DIST_MAX)
				vertOverlap = prev2;
		}
		if (next2) {
			centerOther = (int) ((next2->seg->pos.left + next2->seg->pos.right) / 2.0);
			if (abs(centerOther - centerCur) < SEG_MERGE_HORIZ_DIST_MAX)
				vertOverlap = next2;
		}
		if (prev1) {
			centerOther = (int) ((prev1->seg->pos.left + prev1->seg->pos.right) / 2.0);
			if (abs(centerOther - centerCur) < SEG_MERGE_HORIZ_DIST_MAX)
				vertOverlap = prev1;
		}
		if (next1) {
			centerOther = (int) ((next1->seg->pos.left + next1->seg->pos.right) / 2.0);
			if (abs(centerOther - centerCur) < SEG_MERGE_HORIZ_DIST_MAX)
				vertOverlap = next1;
		}
```

```
        }


        //
        // Test 3:  Evaluate based on distance between regions.
        //

        // First get the distances.
        double distP, distP2, distN, distN2;
        distP = distP2 = distN = distN2 = 0;
        if (prev1) distP = prev1->seg->FindClosestDistance(seg);
        if (prev2) distP2 = prev2->seg->FindClosestDistance(seg);
        if (next1) distN = seg->FindClosestDistance(next1->seg);
        if (next2) distN2 = seg->FindClosestDistance(next2->seg);

        // Then choose the closest segment to merge with.
        minDist = distP + distP2 + distN + distN2;
        if (prev2) {        // prev2 first to bias towards other options in a tie
                minDist = distP2;
                closest = prev2;
        }
        if (next2 && (distN2 <= minDist)) {        // next2 first to bias towards other options in a tie
                minDist = distN2;
                closest = next2;
        }
        if (prev1 && (distP <= minDist)) {
                minDist = distP;
                closest = prev1;
        }
        if (next1 && (distN <= minDist)) {
                minDist = distN;
                closest = next1;
        }


        //
        // Test 4:  A bunch of hacks
        //

        // Follow up on our vertical overlap test for horizontals.
        if (seg->ClassIs(SC_HORIZ)) {
                // Allow horizontals to merge with overlapped regions that are farther away
                if (vertOverlap && (closest == vertOverlap) && (minDist < 2 *
SEG_MERGE_DIST_MAX))
                        minDist = SEG_MERGE_DIST_MAX;

                // If we overlap with something, we merge with it or nothing.
                if (vertOverlap && (closest != vertOverlap))
//                      return NULL;
                        closest = vertOverlap;

                // If we don't overlap, we'd better be darned close
                if (!vertOverlap && (minDist > 0.6 * SEG_MERGE_DIST_MAX))
                        return NULL;
        }
```

```
	// Prevent bad merging of symbols
	if (seg->ClassIs(SC_PUNCT) && (minDist > 0.6 * SEG_MERGE_DIST_MAX))
		return NULL;


	//
	// Now merge with the best block
	//

	// Make sure it's not too far away
	if (minDist > SEG_MERGE_DIST_MAX)
		return NULL;

	// If we were unsuccessful finding a block to merge with, quit now.
	if (closest == NULL)
		return NULL;

	closest->LogPath(3);

	// Merge the blocks
	Block* merge = NULL;
	if ((closest != next) || (next && (closest != next->next)))

		merge = new Block(closest, this);
	else
		merge = new Block(this, closest);
	if (!merge) {
		// Out of memory
		return NULL;
	}
	AddToMergeHistory(closest->GetBlockID());
	merge->LogPath(3);

	// Evaluate the new merged region
	if (closest->seg->ClassIs(SC_HORIZ) && seg->ClassIs(SC_HORIZ))
		// Special case merging two horizontals - an equal sign
		merge->seg->SetClass(SC_EQUALS);
	else
		merge->seg->Classify(vmid);

	// Make sure we are not creating a multi-character.
	if (merge->seg->ClassIs(SC_MULTIPLE)) {
		// Merging was bad -- restore the original state
		merge->Flatten_KeepBlocks();
		delete merge;
		return NULL;
	}

	// Reset iterator for safety
	iter_loc = NULL;

	return merge;
}

// Function:	(Block::)GetMergeCandidate
//
```

```
// Purpose:			Takes a block and retrns the subblock (if any) that are suitable for
//				merging with a fragment
// Arguments:	[in]  block			Block to check for subblocks
//
// Returns:			NULL if an error occurred or if merge was unsuccessful.
//				Otherwise returns a pointer to the new block.
//
// Author:	Susan Dey   1/19/99
//
// Notes:
//
Block* Block::GetMergeCandidate(Block* block){
	if (!g_redoMerging){return block;}
	if (!block){return NULL;}//exit if an empty block was chosen

	Block* temp;
	if (block->type==BT_SEPARATE){
		//FIXLATER: searches the tree left to right
		// change this so it goes right to left if the segment is to the right
		temp=GetMergeCandidate(block->bList1);
		if (temp) {return temp;}
		temp=GetMergeCandidate(block->bList2);
		if (temp) {return temp;}
	}

	if (AlreadyMerged(block->GetBlockID())){ return NULL;}
	if (block->type==BT_HOLD){return NULL;}

	//Default case, BT_UNKNOWN, BT_MERGE
	return block;
}

// Function:		(CABValue::)Undomerge
//
// Purpose:			Goes through the entire data structure and undoes Merges
//				merged blocks that were recognized are ignored.Works recursivly
//				sublists.
//
// Arguments:	none
//
// Returns:			void.
//
// Author:	Jibu Punnoose   10/06/99
//
// Notes:
//
void CABValue::UndoMerging(){
	bool changed;
	Block* current;
	do{
		changed=false;
		current=head;

		while (!changed){
			if (current==head){
```

```
                                        //adjusts head if it is Merge block
                                        head=current->Unmerge(current,&changed);
                                        //in case  it placed a fragment in front of current
                                        while (head->prev){head=head->prev;}

                                } //flatten and fix the head
                                else{
                                        current=current->Unmerge(current, &changed);//just flatten head does
not need fixing
                                }

                                if (current-> next){
                                        current=current->next;//advance pointer
                                }else{ return;} //end of list,we're done
                        }
        } while (changed);
        //BT_Hold and BT_UNKNOWn are ignored

}


// Function:        (Block::)Unmerge
//
// Purpose:                 undoes merged blocks.Places the fragment at the nearest
//                                  in the top level linked list. the other psegment replaces the
//                                  MERGE block in the tree-list structure
// Arguments:       [in]  block                 Block to check for subblocks
//
// Returns:                 void

Block* Block::Unmerge(Block* root, bool* changed){

        if (*changed){ return this;} //error should not have enterd the function

        Block* result=NULL;
        //If it is a merge block do Unmerging
        if (type==BT_MERGE){
                //AT Top level
                if (root==this){ //top level just flaten the list
                        result=Flatten_KeepBlocks();
                        return result;
                }
                //not at top level
                Block* temp=Flatten_KeepBlocks ();
                if (temp->seg->ClassIs(SC_FRAGMENT)){

                //remove and place in front of the root node
                        if (temp->next){
                                result = temp->next;//fix the pointer from the parent
                                temp->next->prev =temp->prev;// if for some reason
                                        //we are in the middle of a list
                        }
                        //place in front of root node
                        if (root->prev){root->prev->next= temp;}
                        temp->next=root;
                        temp->prev= root->prev;
                root->prev= temp;
            }
            else if (temp->next && temp->next->seg->ClassIs(SC_FRAGMENT)){
                //Place behind the rot node
                result= temp;
                temp=temp->next;
                if (temp->prev){ temp->prev->next=temp->next;}

                if (root->next){root->next->prev=temp;}
                temp->prev=root;
                temp->next=root->next;
                root->next=temp;
            }
            *changed=true;
            return result;
        }
        //If it's a seperate block there may be a merge in the subblocks
        // iterate through them! stop if a change is made!
        else if (type==BT_SEPARATE){
            bool done=false;
            Block* current=NULL;
            if (bList1){current=bList1;}
            else if (bList2){
                current=bList2;
                done=true;
            }
            else {return this;}

            while (!(*changed)){
                //if first element may need to update Blist (if first element is a merge block)
                if(current==bList1){bList1=current->Unmerge (root,changed);}
                else {current->Unmerge(root, changed);}//otherwise just call Unmerge

                if (current->next) {current=current->next;}//advance pointer
                else if (bList2 && !done){
                    current=bList2;
                    done=true;
                }
                else{return this;}
            }

            return this;
        }
        return this;
}
```

Appendix B: Header Files

All header files for the Winbank system are reproduced here. They are presented in alphabetical order by file name.

```
// Arbiter.h: interface for the Arbiter class.
// Arbitrates between different recognizers and maintains an
// outfile to measure results if given an actual answer.
//
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_ARBITER_H__65B11026_7CB9_11D2_8AC4_00609702ADF5__INCLUDED_)
#define AFX_ARBITER_H__65B11026_7CB9_11D2_8AC4_00609702ADF5__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#include "Segment.h"

class Arbiter
{
public:
	Arbiter(int num);
	Arbiter(char* filename, int num);
	virtual ~Arbiter();

	//functions
	void arbitrate(Segment* s);
	void inputAnswer (int d);
	void reset();
	int getValue();

	//member variables
	int digit;

private:
	//member variables
	int numNets;			//the number of Nets being used

	double* confidence;
	double CorrectConfAvg;
	double IncConfAvg;
	int ncount;				//count of number in test set
	int *count;				//array of the number of each possible value/integer, indexed
by digit
	int numCorrect;			//count of total correct
	int *correct;			//array of number correct for each possible value/integer
	int FalsePos;			//count of total False Pos
	int *fpos;				//number of false positives for each integer
	int **resultmap;  //maps actual value vs. value guessed
	//can calculate NoRec from this data
	//outfile
	FILE* outfile;

};

#endif // !defined(AFX_ARBITER_H__65B11026_7CB9_11D2_8AC4_00609702ADF5__INCLUDED_)
```

```
// Block.h: interface for the Block class.
//
//////////////////////////////////////////////////////////////////////
// Must #include before Block.h:

#if !defined(AFX_BLOCK_H__57020CC1_A71E_11D2_AB8A_00105A155AE1__INCLUDED_)
#define AFX_BLOCK_H__57020CC1_A71E_11D2_AB8A_00105A155AE1__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#include "Segment.h"
#include "Path.h"


#include "PosNet.h"
#include "DDDNet.h"
#include "MeshNet.h"
#include "Arbiter.h"
#include "StructProcessor.h"


//Global variable
//used to give each new block a unique id number
//increments when a new block is created
//int g_NewBlockIDNum= 0;

typedef enum {
	BT_SEPARATE, BT_MERGE, BT_HOLD, BT_UNKNOWN
} BlockType;


// On a high level, a block is a doubly-linked list element with information about
// several segments inside it.
class Block
{
public:
	// Constructors/Destructors
	Block();
	Block(Segment* s);
	Block (Segment* s, bool copy);		//clean copy
	Block(Block* b1, Block* b2);		// Merging
	virtual ~Block();

	// Functions
	Segment* GetSeg(bool use_norm = false);
	Segment* IterateSegments(bool start, bool use_norm = false, bool use_completed = false);
	void AddPath(Path* p);
```

```
        Status Separate(DDDNet* nn1, PosNet* nn2,DDDNet* nn3, PosNet* nn4, StructProcessor* sp,
Arbiter* arb, Segment* s, int vmid);
        Path* SegmentByRecognizing(DDDNet* nn1, PosNet* nn2,DDDNet* nn3, PosNet* nn4,
StructProcessor* sp, Arbiter* arb, Segment* seg);
        Block* Merge(Segment* s, int vmid);
        Block* Block::UndoMerging(Block* head,Block* root, bool &changed); //undo sublists
        Block* Unmerge(Block* root, bool * changed); //undo individual blocks
        bool IsIsolated(bool look_right);
        void PrepareForRecognition();
        Block* CleanupAfterRecognition();
        Block* ForceFlatten(bool markAsDone);
        bool DoneRecognizing();  // used for intermediate tests during feedback
        bool IsRecognized();     // used for intermediate tests during feedback
        bool IsBeingTested();    // used for intermediate tests during feedback
        void Log(int indent);
        void LogPath(int indent);

        int GetBlockID();//accessor function

        // DEBUG PRINT -- REMOVE LATER
        void PrintPathToFile();

        // Accessor Functions
        bool IsFragment();       // REMOVE THIS HACK!!

        // Data
        Block* next;
        Block* prev;

private:
        // Private Functions
        Path* GetDropFallPath(int dir, int vmid, bool special_case);
        Path* GetContourPath(int vmid);
        void RankPaths(int vmid);
        Status SeparateRegionAlongPath(Path* path);          // Separation
        Status SeparateRegionAlongPath(Path* path, int overlap);
        Block* Flatten_KeepSegment();
        Block* Flatten_KeepBlocks();
        void DeleteBlockList(Block* list);

        int GetUniqueID();  //used to retrieve and increment the global variable
        bool AlreadyMerged(int id);
        void AddToMergeHistory(int id);
        Block* GetMergeCandidate(Block* block);

        // Private Data
        // On a low level, a block is 2 Seg pointers, 2 Block pointers, and a set of paths.
        BlockType type;
        Segment* seg;
        Segment* seg_norm;
        Block* bList1;
        Block* bList2;
        Path** paths;
        int numpaths, paths_alloc;

        double iter_loc;   // used to keep track of iteration
        int block_id;               //the unique id number for this block
        int* mergeHistory;
        int mergeHistorySize;
};

#endif // !defined(AFX_BLOCK_H__57020CC1_A71E_11D2_AB8A_00105A155AE1__INCLUDED_)
```

```
// BWImage.h: interface for the BWImage class.
//
//////////////////////////////////////////////////////////////////////
// Must #include before BWImage.h:
//  Winbank.h

#if !defined(AFX_BWIMAGE_H__C4620598_4F06_11D2_AB7A_00105A145DC0__INCLUDED_)
#define AFX_BWIMAGE_H__C4620598_4F06_11D2_AB7A_00105A145DC0__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

class BWImage
{
public:
        // Constructors/Destructors
        BWImage();
        BWImage(int h, int w);
        BWImage(char* filename, int h, int w);
        BWImage(BIT** pix, int h, int w);
        BWImage(BWImage* img);          //copy constructor
        virtual ~BWImage();

        // Functions
        void AddPixel(int row, int col);
        void RemovePixel(int row, int col, bool adjustBBox = false);
        Status GetConnectedRegion(char regionNum, BWImage* region, Dimension* pos);

        // DEBUG PRINT -- REMOVE LATER
        void PrintToFile(char* filename);

   // Accessor Functions
        int Width() { return width; }
        int Height() { return height; }
        int Weight() { return weight; }
        int BBoxL() { return bbox.left; }
        int BBoxR() { return bbox.right; }
        int BBoxT() { return bbox.top; }
        int BBoxB() { return bbox.bottom; }
   char operator() (const int& row, const int& col) { return pixel[row][col]; }
   char operator() (const Point& p) { return pixel[p.row][p.col]; }
        int FindVerticalMidline();
        int CornerType(Point* p, int dir);
        Status FindNextContourPixel(ClockDir dir, Point* prev, Point* cur);

        // Public normalization routines
        BWImage* Normalize();
        BWImage* NormalizeSize();
        BWImage* TruncateSize();
        BWImage* NormalizeSlant();
        BWImage* NormalizeThickness();
        BWImage* BWImage::CenterWidth();
        Status ScaleUp(int newHeight, int newWidth,
                                    bool fillHeight = true, bool fillWidth = true, double maxScaleDiff =
3);
        Status ScaleDown(int newHeight, int newWidth,
                                          bool fillHeight = true, bool fillWidth = true, double
maxScaleDiff = 3);
        int NeedToScaleWidth();
        void ScaleWidth(int newHeight, int newWidth);

protected:
        void ComputeBBox();
        int ComputeWeight();

public:  // make this protected later...
        // Data
        BIT** pixel;        // 2d array of 1s & 0s representing image
        int width, height;
        Dimension bbox; // used to keep track of bounding box of writing within image
        int weight;

private:
        // Private segmentation functions
        void Fill(int row, int col, char regionNum, Dimension* boundary);

        // Private normalization routines
        BWImage* ThickenLines();
        BWImage* ThinLines();
        bool ContourCheck(int row, int col);
        bool DeleteCheck(int row, int col);

};


BIT** AllocatePixels(const int height, const int width);
void DeallocatePixels(BIT** *pixel, int height);


#endif //
!defined(AFX_BWIMAGE_H__C4620598_4F06_11D2_AB7A_00105A145DC0__INCLUDED_)
```

```
// BWPixel.h: BWPixel Class Declaration
//
//////////////////////////////////////////////////////////////////////

#if !defined _BWPIXEL_HH
#define _BWPIXEL_HH

#include "Pixel.h"

class BWPixel : public Pixel
{
public:

        // constructors
        BWPixel();
        BWPixel(int _height, int _width);

        // destructor
        virtual ~BWPixel();

        // accessors
        virtual unsigned char r(int row, int col);
        virtual unsigned char g(int row, int col);
        virtual unsigned char b(int row, int col);

        virtual void setR(int row, int col, unsigned char val);
        virtual void setG(int row, int col, unsigned char val);
        virtual void setB(int row, int col, unsigned char val);

        virtual void AllocatePixel(int _height, int _width);
        virtual void DeallocatePixel();
        virtual void LoadSignal(TUCharSignal& signal);

private:
        BWPIXEL** pix;
};

#endif
```

```
// CABValue.h: interface for the CABValue class.
//
//////////////////////////////////////////////////////////////////////
// Must #include before CABValue.h:
//   Winbank.h

#if !defined(AFX_CABVALUE_H__C462059A_4F06_11D2_AB7A_00105A145DC0__INCLUDED_)
#define AFX_CABVALUE_H__C462059A_4F06_11D2_AB7A_00105A145DC0__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#include "Block.h"
#include "PosNet.h"
#include "DDDNet.h"
#include "MeshNet.h"
#include "Arbiter.h"
#include "StructProcessor.h"


class CABValue
{
public:
        // FIXLATER: Remove this!
        friend class NeuralNet;

        // Constructors/Destructors
        CABValue();
        virtual ~CABValue();

        // Functions
        Segment* IterateSegments(bool start, bool use_norm = false, bool use_completed = false);
        void OutputSegments(char* filename);
        void OutputSegmentsToOpenFile(ofstream& file);
        char* ComputeValue();
        void Log(int indent);

        // Functions for gathering connected regions
        Status CollectConnectedRegions(BWImage* image);
        Status AddSegment(Segment* seg);
        void RemoveIsolatedRegions(BWImage* image);
        void ReclassifySegments(BWImage* image);

        // Segmentation Functions
        void DoSegmentation(DDDNet* nn1, PosNet* nn2,DDDNet* nn3, PosNet* nn4,
StructProcessor* sp, Arbiter* arb);
        Status Separate(DDDNet* nn1, PosNet* nn2,DDDNet* nn3, PosNet* nn4, StructProcessor* sp,
Arbiter* arb, Block* block, Segment* seg);
        Status Merge(Block* block, Segment* seg);
        void CABValue::UndoMerging();

        // Feedback Functions
        void PrepareForRecognition();
        void CleanupAfterRecognition();
        void CleanupWhenDone();
        bool DoneRecognizing();
        void AdjustClassifications(int iternum);

        // DEBUG PRINT -- REMOVE LATER
        void PrintToFile(char* filename, bool use_norm = false);
        void PrintToOpenFile(ofstream& file, bool use_norm = false);
        void PrintToFileHoriz(char* filename, bool use_norm = true);
        void PrintToOpenFileHoriz(ofstream& file, bool use_norm = true);

        // Data
        Block* head; // front of list
        Block* tail; // end of list


private:
        // Private Functions
        void DeleteSegments();

        // Private Data
        Block* iter_loc;    // Current block being polled by iterator
        int vmid;                       // Vertical midline of the original image (-1 if not yet
calculated)
        int imageHeight, imageWidth;        // Height and width of the original image
        int* value;  //the recognized value of text

};

#endif //
!defined(AFX_CABVALUE_H__C462059A_4F06_11D2_AB7A_00105A145DC0__INCLUDED_)
```

```
// DDDNet.h: interface for the DDDNet class.
//
//////////////////////////////////////////////////////////////////////
```

```
#if !defined(AFX_DDDNET_H__C150EA43_EC75_11D2_93E2_97C5FA26B430__INCLUDED_)
#define AFX_DDDNET_H__C150EA43_EC75_11D2_93E2_97C5FA26B430__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#include "NeuralNet.h"

class DDDNet : public NeuralNet
{
public:
	DDDNet(int numIN, int numHIDDEN, int numOUT);
	DDDNet(char* filename);
	virtual ~DDDNet();

	int getValue(double arr[]);
	void ExtractFeature(Segment* s);

};

#endif // !defined(AFX_DDDNET_H__C150EA43_EC75_11D2_93E2_97C5FA26B430__INCLUDED_)
```

```
// Feedback.h: Globals for Feedback
//
//////////////////////////////////////////////////////////////////////
// Must #include before Feedback.h:
//   WinBank.h
//   CABValue.h

// Function declarations
CABValue* SegmentAndRecognize(BWImage* image, DDDNet* nn1, PosNet* nn2,DDDNet* nn3,
PosNet* nn4,
						StructProcessor* sp, Arbiter* arb);
```

```
// Image.h: Image Class Declaration
//
//////////////////////////////////////////////////////////////////////
// Must #include before Image.h:
//   WinBank.h

#if !defined _IMAGE_HH_
#define _IMAGE_HH_

#include "Pixel.h"


class Image
{

public :

	// constructors
	Image();
	Image(const char* fileName);

	// Destructor
```

```
        ~Image ();

        // load contents of filename into image
        void open(const char* fileName);
        void close();
        bool is_open() { return (ens ? true : false); };

        // Accessors
        unsigned char r(int row, int col);
        unsigned char g(int row, int col);
        unsigned char b(int row, int col);
        void setR(int row, int col, const unsigned char _r);
        void setG(int row, int col, const unsigned char _b);
        void setB(int row, int col, const unsigned char _g);
        int getWidth(){return hsize;};
        int getHeight(){return vsize;};
        int getBWWidth(){return width;};
        int getBWHeight(){return height;};
        const Dimension& GetCab();

        Status Binarize();
        void CleanUp();

        Pixel* pixel;

        // grrrr... these are exposed so that BWImage and Image don't have to know each other
        BIT** bwPixel;// binarized pixel array
        int width, height;  // height and width of binarized image

private:

        // CAB location and Binarization Functions
        int GetRegionThreshold(Dimension& box);
        Status LocateCAB();
        bool DetectHorizontalLine(int row, Dimension& box);
        bool DetectVerticalLine(int col, Dimension& box);
        Status FindCABTopAndBottom(Dimension& box, Dimension& CAB);
        Status FindCABLeftAndRight(Dimension& box, Dimension& CAB);
        Status FindRowLimits(Dimension& dims, Dimension& CAB);
        void BuildProfiles(Dimension& box);
        void BuildColumnProfile(Dimension& box);
        int FindVerticalLine(Dimension& box, char direction);

        // In the end, there can be only one... filtering function.
        // For now there are three.  FIXLATER
        void MeanFilter(Dimension& box);
        void MedianFilter(Dimension& box);
        void SelectiveMedianFilter(Dimension& box);

        // CAB Location and Binarization Data
        Dimension CAB;          // location of CAB
        int thresh;                     // pixel binarization threshold value
        int* vprof;                     // vertical profile
        int* hprof;                     // horizontal profile
        int* cprof;                     // Column profile of 1s, 0s, -1s
                                                // (v-line, unsure, whitespace)
        // Image Data

        int bitCode;  // stores the image type e.g. 8-bit or 24-bit Image

        void Init(const char* fileName);
        TSignalsEnsemble* ens;
        int hsize;                  // width (horizontal size)
        int vsize;                  // height (vertical size)
        bool isOpen;        // flag indicating whether a file is currently open
};


#endif
```

```
// MeshNet.h: interface for the MeshNet class.
//
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_MESHNET_H__C150EA44_EC75_11D2_93E2_97C5FA26B430__INCLUDED_)
#define AFX_MESHNET_H__C150EA44_EC75_11D2_93E2_97C5FA26B430__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#include "NeuralNet.h"

class MeshNet : public NeuralNet
{
public:
        MeshNet(int numIN, int numHIDDEN, int numOUT);
        MeshNet(char* filename);
        virtual ~MeshNet();

        int getValue(double arr[]);
        void ExtractFeature(Segment* s);


};

#endif //
!defined(AFX_MESHNET_H__C150EA44_EC75_11D2_93E2_97C5FA26B430__INCLUDED_)
```

```
// NeuralNet.h: interface for the NeuralNet class.
//
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_NEURALNET_H__4AC286A0_5723_11D2_93E2_C150380CB199__INCLUDED_)
#define AFX_NEURALNET_H__4AC286A0_5723_11D2_93E2_C150380CB199__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#include "Node.h"

class NeuralNet
```

```
{
public:
        // Friends
        friend class PosNet;
        friend class DDDNet;
        friend class MeshNet;

        //constructors/destructors
        NeuralNet();
        virtual ~NeuralNet();

        //data members
        bool initialized;

        //functions
        void trainer(char* filename);
        void recognize(Segment* s, int array_num);
        void setConfidences(double arr[]);
        double test(char* infile, char* outfile);
        virtual int getValue(double arr[])=0;
        void saveWeights(char* filename);
        float randomweight(unsigned seed);
        void convertSNNS(char* infilename, char* outfilename);
        virtual void ExtractFeature(Segment* s)=0;

private:
        //functions
        int setDigit(char* s);
        void init (FILE* infile); //might have to make this public?????
        void train(int digit, double rate);
        void propogate();

        //data members
        int nInputs;
        int nHidden;
        int nOut;
        int type;

        //arrays for the layers of the network
        double* inputs;
        Node** hiddenNodes;
        Node** outNodes;
};

#endif //
!defined(AFX_NEURALNET_H__4AC286A0_5723_11D2_93E2_C150380CB199__INCLUDED_)
```

```
// Node.h: interface for the Node class.
//
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_NODE_H__4AC286A1_5723_11D2_93E2_C150380CB199__INCLUDED_)
#define AFX_NODE_H__4AC286A1_5723_11D2_93E2_C150380CB199__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
```

```
class Node
{
public:
	Node();
	Node(int num, double b);
	virtual ~Node();

	//data members
	double* weights;
	double* inputs;
	double output;
	double bias;
	double error;  //for training only

	//methods
	void setNumInputs(int num);
	void setOutput();
private:
	int nInputs;
};

#endif // !defined(AFX_NODE_H__4AC286A1_5723_11D2_93E2_C150380CB199__INCLUDED_)
```

```
// Path.h: interface for the Path class.
// This class is overloaded and "secretly" used for Contour as well.
// When Path is used, it should be exclusively used as a Path or as
// a Contour.  Don't mix the two... The storage is different inside!
//
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_PATH_H__6AB658C0_92DD_11D2_AB89_00105A155AE1__INCLUDED_)
#define AFX_PATH_H__6AB658C0_92DD_11D2_AB89_00105A155AE1__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

// PathType used to keep track of what segmentation algorithm was used.
typedef enum {
	PT_DROP_TOPLEFT,
	PT_DROP_TOPRIGHT,
	PT_DROP_BOTTOMLEFT,
	PT_DROP_BOTTOMRIGHT,
	PT_CONTOUR,
	PT_UNKNOWN,
	PT_EDF_TOPLEFT,
	PT_EDF_TOPRIGHT,
	PT_EDF_BOTTOMLEFT,
	PT_EDF_BOTTOMRIGHT,

	// The next two types are used to hide the fact that the Contour class
	// is just Path...  No true paths should have these types!
	CT_TOP,
	CT_BOTTOM

} PathType, ContourType;
```

```
// Constants to keep track of right and left inside Contour
#define CONTOUR_LEFT 0
#define CONTOUR_RIGHT 1

class Path
{
public:
	// Constructors/Destructors
	Path();
	Path(int alloc_len);
	virtual ~Path();

	// Functions
	Status AddPoint(int row, int col);
	Point* GetPoint(int index);
	Point* IteratePoints(bool start);
	int GetLength();
	void Log(int indent);

	// DEBUG PRINT -- REMOVE LATER
	void PrintToFile(char* filename, Segment* s);

	// Contour Functions & Data
	CTOUR(CTOUR& c, int pointsToCopy);
	CTOUR(int num, ContourType t);
	Status AddPoint(Point& p, int end);
	int centerIndex;	// Index of center of point array
	int leftIndex;		// Index of left side of point array
	int rightIndex;		// Index of right side of point array

	// Data
	PathType type;		// Algorithm used to create the path (handy for debugging or maybe
ranking)
	int score;			// Path score used for ranking

private:
	// Data
	int alloc_length;	// Allocated length
	int length;			// Length of actual data
	Point* point;		// Array of points
	int iter_loc;		// Iteration counter
};

#endif // !defined(AFX_PATH_H__6AB658C0_92DD_11D2_AB89_00105A155AE1__INCLUDED_)
// Pixel.h: Pixel Class Declaration
//
//////////////////////////////////////////////////////////////////////

#if !defined _PIXEL_HH_

#define _PIXEL_HH_

#include <zmstall.h>
```

```
class Pixel
{
public:

        // constructors
        Pixel();
        Pixel(int _height, int _width);

        // destructor
        virtual ~Pixel();

        // accessors
        virtual unsigned char r(int row, int col) = 0;
        virtual unsigned char g(int row, int col) = 0;
        virtual unsigned char b(int row, int col) = 0;

        virtual void setR(int row, int col, unsigned char val) = 0;
        virtual void setG(int row, int col, unsigned char val) = 0;
        virtual void setB(int row, int col, unsigned char val) = 0;

        virtual void AllocatePixel(int height, int width) = 0;
        virtual void DeallocatePixel() = 0;

        virtual void LoadSignal(TUCharSignal& signal);
        virtual void LoadSignal(TRGBSignal& signal);

protected :
        int height;
        int width;
};


#endif
```

```
// PosNet.h: interface for the PosNet class.
//
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_POSNET_H__C150EA40_EC75_11D2_93E2_97C5FA26B430__INCLUDED_)
#define AFX_POSNET_H__C150EA40_EC75_11D2_93E2_97C5FA26B430__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#include "NeuralNet.h"

class PosNet : public NeuralNet
{
public:
        PosNet(int numIN, int numHIDDEN, int numOUT);
        PosNet(char* filename);
        virtual ~PosNet();

        int getValue(double arr[]);
        void ExtractFeature(Segment* s);
};

#endif // !defined(AFX_POSNET_H__C150EA40_EC75_11D2_93E2_97C5FA26B430__INCLUDED_)
```

```
// Process.h: processing segment stuff
//
//////////////////////////////////////////////////////////////////////

// Function Declarations

double FirstHitAvg(BWImage* s, int SR, int ER);
double LastHitAvg(BWImage* s, int SR, int ER);
void OutputPixel(BIT** pix, char* temp);
```

```
// Recognize.h: Globals for Segmentation
//
//////////////////////////////////////////////////////////////////////
// Must #include before Segmentation.h:
//   WinBank.h
//   CABValue.h

// Function Declarations

Segment* RecognizeSeg(DDDNet* nn1, PosNet* nn2, DDDNet* nn3, PosNet* nn4,StructProcessor* sp,
Arbiter* arb, Segment* seg);
void TotalTest(char* filename,DDDNet* nn1, PosNet* nn2,DDDNet* nn3, PosNet* nn4, StructProcessor*
sp, Arbiter* arb);
void NormDigitFile(char* infile, char* outfile);
int Train(char* file_loc, char* weightfile);
int setDigit(char*s);
```

```
// RGBPixel.h: RGBPixel Class Declaration
//
//////////////////////////////////////////////////////////////////////

#if !defined _RGBPIXEL_HH
#define _RGBPIXEL_HH_

#include "Pixel.h"

class RGBPixel : public Pixel
{
public:

        // constructors
        RGBPixel();
        RGBPixel(int _height, int _width);

        // destructor
        virtual ~RGBPixel();

        // accessors
        virtual unsigned char r(int row, int col);
        virtual unsigned char g(int row, int col);
        virtual unsigned char b(int row, int col);
        virtual void setR(int row, int col, unsigned char val);
        virtual void setG(int row, int col, unsigned char val);
        virtual void setB(int row, int col, unsigned char val);

        virtual void AllocatePixel(int _height, int _width);
        virtual void DeallocatePixel();
        virtual void LoadSignal(TRGBSignal& signal);

private:
        RGBPIXEL** pix;
};


#endif
```

```
// SegContour.h: Globals for Contour Segmentation
//
//////////////////////////////////////////////////////////////////////
// Must #include before SegContour.h:
//   WinBank.h
//   CABValue.h
//   Segmentation.h

// Function declarations
Block* SeparateUsingContours(BWImage* image, Segment* region);
```

```
// SegDropFall.h: Globals for Drop-Fall Segmentation
//
//////////////////////////////////////////////////////////////////////
// Must #include before SegDropFall.h:
//   WinBank.h
//   CABValue.h
//   BWImage.h
//   Segmentation.h

// Types

// Variables
#define SEG_FALL_TOPLEFT 0
#define SEG_FALL_BOTTOMLEFT 1
#define SEG_FALL_TOPRIGHT 2
#define SEG_FALL_BOTTOMRIGHT 3
#define SEG_UP 0
#define SEG_DOWN 1

// Function declarations
Block* SeparateUsingDropFall(BWImage* image, Segment* region);
```

```
// Segment.h: interface for the Segment class.
//
//////////////////////////////////////////////////////////////////////
// Must #include before Segment.h:
//   WinBank.h


#if !defined(AFX_SEGMENT_H__C4620599_4F06_11D2_AB7A_00105A145DC0__INCLUDED_)
#define AFX_SEGMENT_H__C4620599_4F06_11D2_AB7A_00105A145DC0__INCLUDED_
```

```
#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#include "BWImage.h"

// SegClass: used to classify segments during segmentation
typedef enum {
        SC_UNKNOWN,                             // default: hasn't been classified
        SC_NONSENSE,            // segment should be discarded
        SC_FRAGMENT,                    // segment is an incomplete character
        SC_FRAGMENT_ERROR,              // previously classified incomplete -- merge failed
        SC_MULTIPLE,            // segment is more than one character
        SC_MULTIPLE_ERROR,// previously classified multiple -- separation failed
        SC_PUNCT,                       // segment is a comma or period
        SC_EQUALS,                      // segment is an equal sign
        SC_HORIZ,                       // segment is a horizontal line
        SC_DIGIT                        // segment is a single digit
} SegClass;

class Segment: public BWImage
{
public:
        // Friends
        friend class Block;

        // Constructors/Destructors
        Segment(int h = NORM_Y, int w = NORM_X);
        Segment(BWImage* image);
        Segment(Segment* s);    //copy constructor
        virtual ~Segment();

        // Accessor Functions
        bool ClassIs(const SegClass& c) { return (classification == c); };
        void SetClass(const SegClass& c) { classification = c; };
        void SetPos(Point topLeftCorner);
        void Log(int indent);
        void LogConfidences(int indent);
        int DistFrom(int col); // hack... remove later?

        // Segmentation Functions
        void Classify(int vmid);
        void RelaxClassification(int iternum);
        double FindClosestDistance(Segment* s);
  Segment* Normalize();
  Segment* Truncate();

        // Data
        // height, width, bbox, pixel inherited from BWImage
        double Conf0[NUM_POSSIBLE];
        double Conf1[NUM_POSSIBLE];
        double Conf2[NUM_POSSIBLE];
        double Conf3[NUM_POSSIBLE];
        double ArbConf[NUM_POSSIBLE];
        int digit;                      // value determined by recognizer
        Dimension pos;                  // to keep track of original position of segment within image
```

```
	int num_examinations;	// small hack to allow classification relaxation

private:
	// Private data
	SegClass classification;

	// Private segmentation routines
	Status RankPaths();
};

#endif //
!defined(AFX_SEGMENT_H__C4620599_4F06_11D2_AB7A_00105A145DC0__INCLUDED_)
```

```
// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
//      are changed infrequently
//

#if !defined(AFX_STDAFX_H__FBA443E8_CAB2_11D2_AB8E_00105A155AE1__INCLUDED_)
#define AFX_STDAFX_H__FBA443E8_CAB2_11D2_AB8E_00105A155AE1__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#define VC_EXTRALEAN		// Exclude rarely-used stuff from Windows headers

#include <afxwin.h>		// MFC core and standard components
#include <afxext.h>		// MFC extensions
#include <afxdisp.h>		// MFC OLE automation classes
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>			// MFC support for Windows Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT


//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the previous line.

#endif //
!defined(AFX_STDAFX_H__FBA443E8_CAB2_11D2_AB8E_00105A155AE1__INCLUDED_)
```

```
// StructProcessor.h: interface for the StructProcessor class.
//
//////////////////////////////////////////////////////////////////////

#if
!defined(AFX_STRUCTPROCESSOR_H__A7FC20E3_B134_11D2_93E2_8B899EA9553B__INCLUDE
D_)
#define
AFX_STRUCTPROCESSOR_H__A7FC20E3_B134_11D2_93E2_8B899EA9553B__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

class StructProcessor
{
public:
		StructProcessor();
		virtual ~StructProcessor();
		StructProcessor(char* filename);

		//functions
		int check0();
		int check1();
		int check2();
		int check3();
		int check4();
		int check5();
		int check6();
		int check7();
		int check8();
		int check9();


		void SetSeg(Segment* s);
		Segment* PreProcess(Segment* s);
		void ScaleWidth();
		int PostProcess(Segment* s);
		void inputAnswer(int ans);

		//data members
		double AvgWidth;
		int initguess;		//the guess given to the post processor
		int digit;		//the digit that the post processor will set for the input

private:
		Segment* seg;
		FILE* outfile;
		int numSamples;	//number of samples tested
		int numCorrect;	//number of segments that the structprocessor has correctly evaluated
		int numToWrong;		//number incorrectly evaluated
		int* total;		//total number evaluated for each value
		int** InitRight;//array representing that actions of the structprocesor when the
						//intial guess is correct. dimension = NUM_POSSIBLEx2.
						//the first element is the count of each digit initially guessed
						//the second element is a count of the number that had their
value
						//switched to NR
		int** InitWrong;//array representing that actions of the structprocesor when the
						//intial guess is incorrect.  dimension = NUM_POSSIBLEx2.
						//the first element is the count of each digit initially guessed
						//the second element is a count of the number that had their
value
						//switched to NR
		bool processed;	//value that keeps track of a digit has been handle by the post processor

		//private member functions
		double FirstHitAvg(int SR, int ER);
		double LastHitAvg(int SR, int ER);
		int Num2ndTrans(int SR, int ER);
		Point HoleDetect(int SR, int k);
		Point getBound(int top,int bot);
		bool IsLoopClosedTop(int toprow);
```

```
        bool IsLoopClosedBottom(int botrow);

};

#endif //
!defined(AFX_STRUCTPROCESSOR_H__A7FC20E3_B134_11D2_93E2_8B899EA9553B__INCLUDE
D_)
```

```
// WinBank.h: Globals for WinBank program
//
//////////////////////////////////////////////////////////////////////

// This prevents conflicts from including this file more than once.
#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000


// Athena compatibility
#define ATHENA 0  // Set to 1 for compatibility with Athena, 0 for Windows
#if (ATHENA)
#include "Athena.h"
#else
#include <windows.h>
#endif
#include <fstream.h>
#include <stdio.h>

// This variable indicates whether we're in "gather segment" mode -> human recognizer
extern bool g_gatherSegs;
// This variable indicates what file to put segments into when we're in "gather segment" mode
extern FILE* g_gatherSegsFile;

// WINBANK PROGRAM PARAMETERS

//  ----------------------------------------
// |                  T       | T = CAB_TOP_MARGIN
// |              ---------   |   = index of top row of CAB
// |             H |        |R| R = CAB_RIGHT_MARGIN
// |              ---------   |   = index of rightmost column of CAB
// |                  W       | H = CAB_HEIGHT
// |                          |   = number of rows in CAB
// |                          | W = CAB_WIDTH
// |                          |   = number of columns in CAB
//  ----------------------------------------

extern double CheckSizeFactor;
#define CSF CheckSizeFactor

// CAB Location
#define SEG_CAB_RIGHT_MARGIN (int)(10*CSF)          // Width of right margin >
#define SEG_CAB_TOP_MARGIN (int)(10*CSF)        // Width of top margin      > see diagram
#define SEG_CAB_WIDTH (int)(265*CSF)            // Width of CAB                 > above
#define SEG_CAB_HEIGHT (int)(70*CSF)            // Height of CAB            >
#define SEG_CAB_MIN_HEIGHT (int)(10*CSF)        // Min height of detected CAB
#define SEG_CAB_MIN_WIDTH (int)(10*CSF)             // Min width of detected CAB
```

```
#define SEG_CAB_PADDING (int)(3*CSF)                    // Extra space left when line is
estimated

// Connected region extraction and classification
#define SEG_THRESH_LIGHTNESS 0.8                        // Thresh control: 0->image all
white, 1->black
#define SEG_REGION_DISTANCE_MAX (int)(25*CSF)     // A region spaced away this far is outside
the CAB
#define SEG_VERY_LOW_WEIGHT_THRESH 14*CSF*CSF           // Below this a region is thrown
away
#define SEG_LOW_WEIGHT_THRESH 35*CSF*CSF                // Below this a region is judged
incomplete
#define SEG_ASPECT_MIN 0.25                                     // Below this a region is a
horizontal line
#define SEG_ASPECT_CUT 0.65                                     // Used to classify region
as multi or horiz -- increased on relaxation
#define SEG_ASPECT_MAX 1.54                                     // Maximum limit of
aspect ration relaxation
#define SEG_MAX_LINE_HEIGHT 9*CSF                       // A region taller than this is NOT a
horizontal line
#define SEG_MAX_RELAXATIONS        2                                    // Max # of
classification relaxations allowed

// Merging
#define SEG_MERGE_HORIZ_DIST_MAX 8*CSF                  // Horizontals are merged if their
centers are closer than this
#define SEG_MERGE_DIST_MAX 7*CSF                        // Regions which are farther apart
than this are not merged

// Separation
#define SEG_CONTOUR_FRACTION 1.0                                                //
Fraction of top/bottom contour to look at
#define SEG_CONTOUR_POINT_HORIZONTAL_DISTANCE_LIMIT 3*CSF       // Max horizontal
distance between critical points
#define SEG_CONTOUR_POINT_ABSOLUTE_DISTANCE_LIMIT 5*CSF         // Max distance
between critical points
#define SEG_MIN_CUT_ANGLE 0.2
        // Min angle allowed along a path before score is reduced
#define SEG_MAX_CUT_LENGTH 4*CSF                                                // Max
cut length before the score is reduced
#define SEG_CONTOUR_CENTER_DISTANCE_LIMIT 10                    // Max distance
of critical point from center
#define SEG_MAX_PATHS 5
        // Max number of paths used in feedback
#define SEG_CUT_POINTS -2
        // Path score modifier for a cut
#define SEG_CONCAVE_CORNER_POINTS +1                                            // Path
score mod for concave corner
#define SEG_NONCORNER_POINTS -1
        // Path score mod for cut on straight line
#define SEG_CONVEX_CORNER_POINTS -3
        // Path score mod for convex corner
#define SEG_CUT_ANGLE_POINTS -1
        // Path score mod for low-angle cuts
#define SEG_CUT_LENGTH_POINTS -2 // should be -1?  FIXLATER // Path score mod for long cuts
```

```
#define SEG_DROP_FALL_POINTS -1
        // Drop fall-specific path score mod
#define SEG_DROP_FALL_SPECIAL_CASE true                                  // for
special case drop fall
#define SEG_DROP_FALL_NORMAL_CASE false                                  // for
standard drop fall
// Normalization
#define NORM_Y 16                                  // Normalized Height Dimension
#define NORM_X 16                                  // Normalized Width Dimension
#define SEGMENT_WEIGHT_MINIMUM 10              // Scaling failed if a segment is below this
weight


// Recognition
#define NUM_POSSIBLE 12                            // Number of values output by
recognizer
#define ZERO 0
#define ONE 1
#define TWO 2
#define THREE 3
#define FOUR 4
#define FIVE 5
#define SIX 6
#define SEVEN 7
#define EIGHT 8
#define NINE 9
#define POUND 10                                   // Pound or star symbol
#define CONN_ZEROES 13
#define CONN_DIGITS 12
#define NOT_RECOGNIZED 11
#define RS 14
#define COMMA 15                                   // Comma or period        >>
        These 3 are recognized
#define EQUAL_SIGN 16                          // Equal sign                >> by the
segmentor, not
#define MINUS 17                                   // Minus sign             >>
        by the recognizer.
#define THROWAWAY 20                               // Obsolete?

//for traingin nets
extern bool g_trainmode;

//for segment files
#define WRITEDIGITNUM 1

// Feedback
#define MAX_ITERATIONS 5                       // Max feedback iterations

//use recognition to determine the best segmentation path
extern bool g_segmentByRecognizing;
extern bool g_redoMerging; //redoes merges
extern bool g_useEDF;     //uses Extended Drop Fall


// Input/Output file extensions and printing information
#define PIXEL_EXT ".pix"
```

```
#define VAL_EXT ".txt"
#define LOG_EXT ".log"
extern char** g_Args;
extern char g_val01[];

// Other values
#if (!ATHENA)
#define NULL 0
#endif

extern bool g_isGUI;

// Simple types
#define BIT unsigned char
typedef enum {SEG_OK, SEG_ERROR} Status;
typedef enum {CLOCKWISE, COUNTERCLOCKWISE} ClockDir;
#define CTOUR Path

// Dimension: used during segmentation
// Dimensions taken in pixels with origin at top left corner
struct Dimension {
        int left, right, top, bottom;
};

struct Point {
        int row;
        int col;
};

#define SEG_DEBUG TRUE


// setumo
typedef struct RgbPixel {
  unsigned char r, g, b;
} RGBPIXEL;


#define BWPIXEL unsigned char
```