(13)

# Character Segmentation Heuristics for Check Amount Verification

by

Salman Amin Khan

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

June 1998



Author ..................................................................
Department of Electrical Engineering and Computer Science
May 26, 1998

Certified by ..............................................................
Dr. Amar Gupta
Thesis Supervisor

Accepted by ..............................................................
Arthur C. Smith
Chairman, Department Committee on Graduate Thesis

## Character Segmentation Heuristics for Check Amount Verification


by

Salman Amin Khan

Submitted to the Department of Electrical Engineering and Computer Science



May 26, 1998

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



# ABSTRACT

Automatic check amount verification involves three basic steps: 1) finding the courtesy amount (the numerical amount) on the check, 2) segmenting that amount into its individual characters, and 3)Using a recognition engine to recognize the individual characters and, thus, recognizing the check's amount. This thesis will focus on improving the segmentation process. In particular, it will focus on the classical approach of segmentation based on charcter features alone (i.e. not aided by recognition). A segmentation workbench is developed for the purpose of studying current and potential techniques for numerical digit segmentation. The problem of segmenting overlapping or touching characters is studied in detail. Also, a hybrid heuristic involving a combination and variations of the "drop-falling" algorithm is developed which has promising results. This algorithm is then incorporated into a complete optical character recognition system.


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

## Acknowledgments

I would like to thank Dr. Amar Gupta for the time that he put in helping me flesh out the research done in this thesis.

I would like to thank my mother, Gulnar Khan, and my sister, Farah Khan, for all the help and support they have given me throughout my life.

# Table of Contents

# Chapter 1:Introduction

People have always tried to develop machines which could do the work of a human being. The reason is obvious since for most of history, man has been very successful in using the machines developed to reduce the amount of physical labor needed to do many tasks. With the advent of the computer, it became a possibility that machines could also reduce the amount of mental labor needed for many tasks. Over the past fifty or so years, with the development of computers ranging from ones capable of becoming the world chess champion to ones capable of understanding speech, it has come to seem as though there is no human mental faculty which is beyond the ability of machines.

There is one human ability, however, which has eluded the prowess of machines.--namely, that of recognizing a human's handwriting. Of course, much progress has been made in the way of computer handwriting recognition, but few of us believe that a computer will ever be able to read a human's handwriting as good as a human. Even so, it doesn't hurt to try to develop technology which can approach the recognition ability of humans. After all, even less-than-perfect handwriting recognition has its uses.

## 1.1 Overview of handwriting recognition

Handwriting recognition is an extremely large field which can be divided into two general sub-fields: on-line and off-line recognition. On-line recognition is the process of recognizing the characters as they are being written. Examples of systems that employ on-line recognition include the Apple Newton and the Palm Pilot. On the other hand, off-line recognition is recognizing the characters after they have been written. Off-line recognition systems' potential uses lie in fields such as document processing, mail direction, and check verification. Due to the lack of stroke and timing information, off-line recognition is considered a much harder problem than on-line recognition.

## 1.2 The problem of automatic check amount verification

One particular important application of off-line handwriting recognition is check amount recognition. This subfield is very important because of its potential for reducing the overhead in processing checks. 66 billion checks are written annually in the United States alone and it is believed that this number will continue to grow well into the next century [12]. The amounts written on each of these 66 billion checks has to be encoded (often several times) by hand, resulting in a non-trivial expense on the part of banks and the Federal government. Drastically reducing the number of checks which have to be read by hand using automated handwriting recognition could potentially save billions of dollars in the United States alone.

Many systems are already in existence which attempt to tackle the problem of check amount recognition, but none are perfect. Depending on whom you ask, the best check amount recognition engines today have about 50%-85% recognition rates [1, 3, 11]. This means that these systems can provide, with a reasonable level of confidence, the numerical amounts written on 50%-70% of all checks written. Even more, of the 50%-70% of the checks which are recognized, a certain portion are recognized incorrectly (currently .1% for the best systems). Even though an error rate as small as one in a thousand checks may seem minor, it could be very significant in the real world if, for example, a $1,000.00 check is misread to be $100,000!

Any system which attempts to recognize the amount written on a check will have to tackle three fundamental problems (which will be described in further detail later on): (1) finding the courtesy amount block (CAB) on the check; (2) segmenting the digits, commas, and decimals in the CAB; and (3) preprocessing and recognition of the individual characters in the CAB. The first issue has been addressed in other research [2] and, thus, will not be dealt with in this thesis. The segmentation and recognition steps are what this thesis will focus on.

### 1.3 Segmentation Overview

This thesis will focus on the problem of segmentation in particular. Segmentation, or separating out the individual characters which constitute a written character string, is a straightforward process when the characters are well-spaced and written with some degree of neatness (Figure 1). In that ideal situation, one just has to pick out each connected component. The problem becomes much more difficult (Figure 2) when characters are touching, overlap, or disjointed.

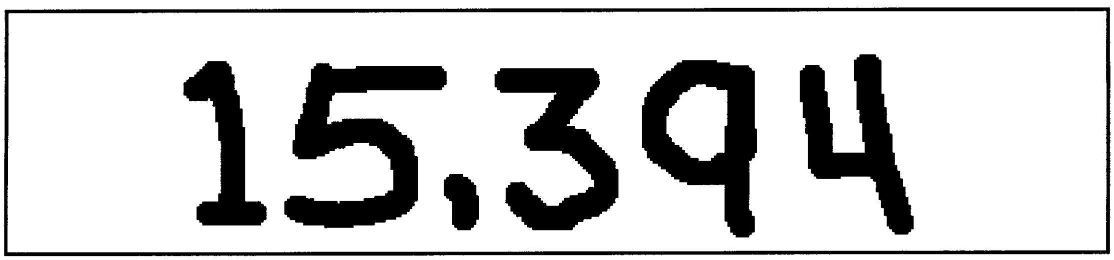


**Figure1: Example of well written numerical amount**

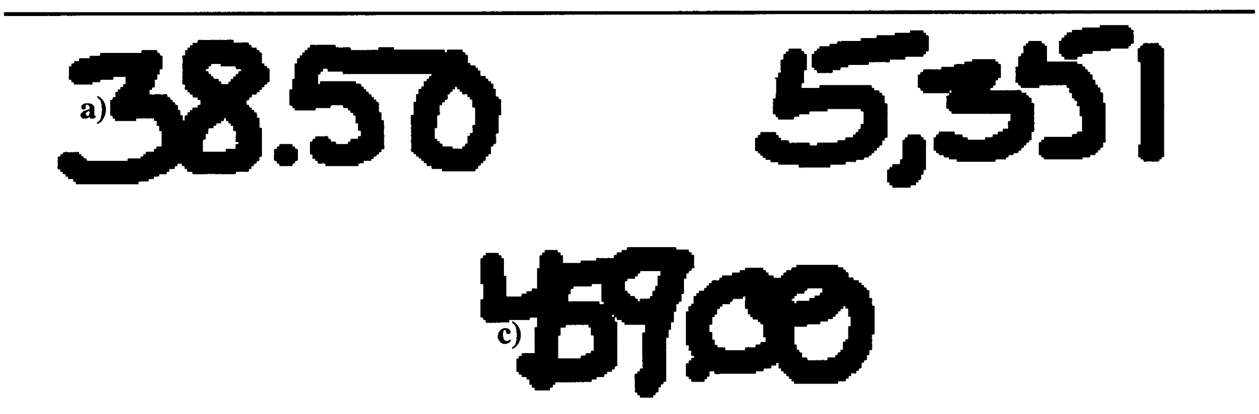


**Figure 2: Examples of a) touching b) disjointed and c) overlapping numbers**

There are several steps commonly employed today for segmenting an image of a numerical string. The first step in segmenting any string of symbols is to identify the connected components in the image of the string. Connected pixels are usually grouped together using some sort of a search algorithm (which will be described in Chapter 5). Once the connected components are found, they then have to be classified according to whether they represent part of a character, a whole character, or two or more characters. This is commonly done using a variety of heuristics which make use of the components' aspect ratios and positions. If a connected component is classified as being made up of more that one character, then another 'splitting' heuristic is used. For example, a common algorithm used today is the min-max algorithm which separates the connected characters by splitting the connected region from a minimum on the upper contour to a maximum on the lower contour (Figure 3). Similar splitting heuristics have also been employed by Dimauro et al.[1].

The heuristics tend to be effective in general, but still fail on a significant portion of cases with many connected or overlapping digits. In fact, the inability of current systems to properly segment these character strings has been the limiting factor in building better recognizers. Since the recognition process is doomed to fail without correct segmentation, it is vitally important for the segmentation process to become more robust if automatic check verification is ever to be commercially feasible. The research described in this thesis attempts to extend the heuristic set in order to enhance segmentation performance.

In order to facilitate the heuristic experimentation, a graphical workbench called the Segmentor was developed. This tool enables the user to quickly change test data and modify heuristics. This tool is used extensively in experimenting with the variations of 'drop-falling' type algorithms.

# Chapter 2: Fundamental Segmentaion Issues

Segmentation is the process of separating out the individual digits which make up a number. Although it is a straightforward task when dealing with typewritten or well written numbers, it can be quite difficult when the digits are handwritten. The main reason for this is that handwritten characters often overlap and, in some cases, may be disjointed. Also, the wide variations in handwriting styles make it very difficult to make generalizations for making segmentation heuristics. In this chapter, we will explore the basic processes common to any system dedicated to handwritten numeral segmentation and the assumptions on which they are based.

## 2.1 Thresholding of the numeral image

Before any segmentation or processing on a numerical image can be done, it is very useful to convert the image from grayscale or color to a black and white bitmap. This is a very simple process which entails just scanning the image and changing each pixel value to either black or white depending on whether they are above or below a given threshold. The threshold chosen should be a function of the intensity range of the pixels in the image.

The threshold-determining function can be made arbitrarily complex to optimize the conversion process, but setting the threshold to the average of the minimum and maximum threshold values is usually sufficient. This heuristic will break down only when the average of the minimum and maximum pixel values is either less than many of the pixels meant to represent black or greater than many of the pixels meant to represent white. This could happen if, say, most of the pixels values fall in the 150-255 range, but there is one pixel with a value of zero. Since this is unlikely to happen (devices that scan the images are very unlikely to produce images with random pixel values so far from the range of most of the other pixels), it is usually not necessary to worry about these special cases. Alternate heuristics which could take care of the above-mentioned problem (but have their own cases on which they fail) include basing the threshold on the weighted average of all the pixels or determining white or black pixels based on changes in contrast (for example, if two adjacent pixel values differ from each other significantly, than set the lighter one to white and the darker one to black.

### 2.2 Extracting connected components

The first step in actually segmenting any numeral image is to extract out each of the connected components. In order to do so, it is necessary to find a black pixel and to then use a search algorithm to find all of the pixels which are connected to it. Once all of those pixels are recognized, they should be grouped together into one connected component. Then another, as yet not-looked-at, pixel needs to be found on which the process of searching for all of the connected components is repeated. The overall process is repeated again and again until every black pixel in the image is classified as being part of a connected component.

We say that two pixels are connected to each other if they are adjacent. Since the characters which are being segmented are at least several pixels wide, it is unusual for two pixels which are only diagonal with respect to each other (they share no other adjacent neighbors) to be part of the same character. This is why, in searching for connected pixels to a given pixel, it only necessary to take into consideration the pixels above, below, to the right, and to the left of that pixel.

If the digits in the numeral image are well separated when they are written, then each connected component would represent a whole character in the numeral string and the task of segmentation would be done. Unfortunately, this is seldom the case. If any of the digits are connected or disjointed, then any of the connected components could represent a single character, more than one character, or part of a character. How we classify the connected components into one of these three categories is the subject of the next section.

### 2.3 Classification of connected components

Most segmentation methods today classify the connected components using information on their sizes and positions. The ratio of a connected component's bounding box height to its width is the most common indicator of what fraction of characters the component is composed of. If the width is greater than a certain multiple of the height, then it is very unlikely that the component is only one character. More likely, it is either more than one character or part of a character that is a horizontal line.

Although using the width-to-height ratio is useful for the more standard cases, it is not difficult to find cases on which it breaks down. For example, very often the numbers 2,3,5,or7 are written so that they are unusually wide. Moreover, a 1 and 7 could be connected together and the resulting connected components would have a height to width ratio identical to the 7 alone. Because of this, it is a nontrivial problem to robustly determine how many characters a connected component is composed of.

Using a connected component's width-to-height ratio and position is much more useful in identifying whether or not the component is a separated segment of a larger character. The most common case of this happening is when the top stroke of a five is separated from the rest of it. In this case the top stroke, which is an almost completely horizontal line, would have a very high width to height ratio (much higher than even a connected component composed of many characters). Not only that, but the height of the component will be significantly less than the heights of the other connected components in the image and its bounding box will be contained in the upper half of the image. Based on these assumptions, it is not difficult to find the upper stroke of a five (the only characters that it could conceivably be confused with are the commas and decimal points, but those are located in the lower half of the image and would not have as high of a width to height ratio--this is described in more detail in Chapter 5).

As we can see, robustly classifying the components is not an easy thing to do. More complicated heuristics based on features other than the aspect ratios and positioning of the components will have to be developed in order to more accurately classify them. These heuristics could potentially make use of min-max points on the upper and lower contours and come type of complexity measure of the pixels making up the component, in conjunction with the aspect ratio and positioning, to make a better guess at properly classifying the component.

# Chapter3: Current Segmentation Techniques

Many different methods have been developed over the past 50 years for the purpose of separating two connected characters [1, 3, 4, 5, 6, 7, 8]. Despite the wide variety in the methods developed, they can be divided into two groups: 1) those that are based only on character features and b) those that make use of some type of pre-recognition process. Since the focus of this thesis is on developing a heuristic which falls under the first of the two categories, we will restrict ourselves to surveying feature based techniques only in this chapter.

Once a connected component is classified as being composed of more than one character, all that is left to be done is to determine how many characters the component is composed of and how to optimally separate the characters. Most of these components will be composed of two characters, but in unusual cases they can be composed of three or more. Heuristics similar to the ones employed in Section 2.3 can be used to determine just how many characters make up the component. As mentioned in that section, these heuristics are not perfect and can lead to misclassification in special cases. To simply the discussion, we will focus on the case where there are two connected characters involved and then extend the heuristics developed for that problem to the case of three or more characters.

The algorithms to be discussed in this chapter can be classified under the categories of "Min-Max", "Hit and Deflect", "Drop Fall", and "Structural Feature" based segmentation. Collectively, they constitute the major types of nonrecognition-based segmentation algorithms being used today. Drop fall based algorithms will be studied in further detail in Chapter 4 as they provide the basis for the hybrid drop fall algorithm developed as part of this thesis.

### 3.1 Hit and Deflect algorithms

Hit and Deflect algorithms attempt to find an optimal path for cutting a connected component by literally hitting and deflecting their way through the connected component. Casey *et al.* [6] define Hit and Deflect algorithms to be those "able to compute a curved segmentation path by iteratively moving a fixed point." Most implementations of this type of algorithm involve a set of rules which dictate in which direction to continue construction of the cutting path based on the surroundings of the current path pixel. The rules could be designed so as to mimic other segmentation heuristics such as min-max or drop fall (to be described in the next section) methods.

One of the earliest examples of employing a hit and deflect strategy into a segmentation system was done by Shridar and Badredlin [7] in 1986. They used Hit and Deflect in an attempt to segment handwritten words. Their system first tried to separate out characters by a performing a vertical scan near the middle of an image. If the scan resulted in zero or two black-to-white or white-to-black transitions, then it was assumed that the characters were either not connected or simply connected (connected at only one point). In this case, the scan line would provide a suitable segmentation path. On the other hand, if scan line crossed more than two transitions, the assumption was made that the characters are slanted. This is when a Hit and Deflect-type algorithm was used. Their algorithm starts from a peak in the lower contour of a connected component. It then attempts to move upwards according to a series of rules. These rules are designed to minimize the number of cuts which have to be made though the component. The algorithm would finish once it reaches the top of the image.

### 3.2 Drop Fall algorithms

Drop fall algorithms attempt to build a segmentation path by mimicking an object falling or rolling in between the two characters which make up a connected component (Figure 3). This class of algorithm was first proposed by Congedo *et al.* in [10]. There are four primary types of drop-fall algorithms which differ on the direction and the starting point of the drop fall. These are top-left (or left-descending), top-right (or right-descending), bottom-left (or left ascending), and bottom-right (or right ascending). Although similar in techniques, each of these variations has their own strong and weak point which will be explored further in Chapter 4. Since the implementation of a drop fall algorithm usually involves a set of rules which dictate the next pixel in the path of the fall which are based on the current position, drop fall algorithms could be seen as a specific type of Hit and Deflect algorithm. In the next chapters, we will explore how combining variations of the drop fall algorithm could lead to a potentially more powerful segmentation technique.

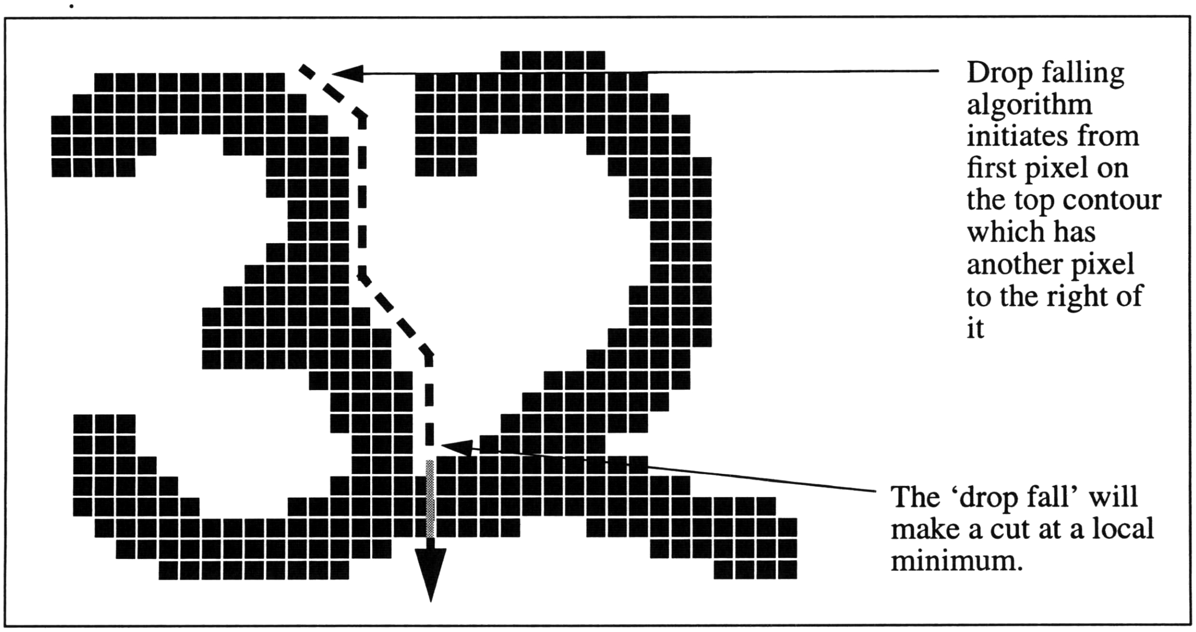


**Figure 3: Example of the path of a 'drop-falling' algorithm**

### 3.3 Structural feature-based segmentation

Another major type of nonrecognition based segmentation heuristic involves basing the segmentation of a connected component on structural features in the component itself. These could be points at which the slope of the contour changes dramatically or extrema within the component. Examples of systems which use structural feature-based segmentation are given in [8] and [9].

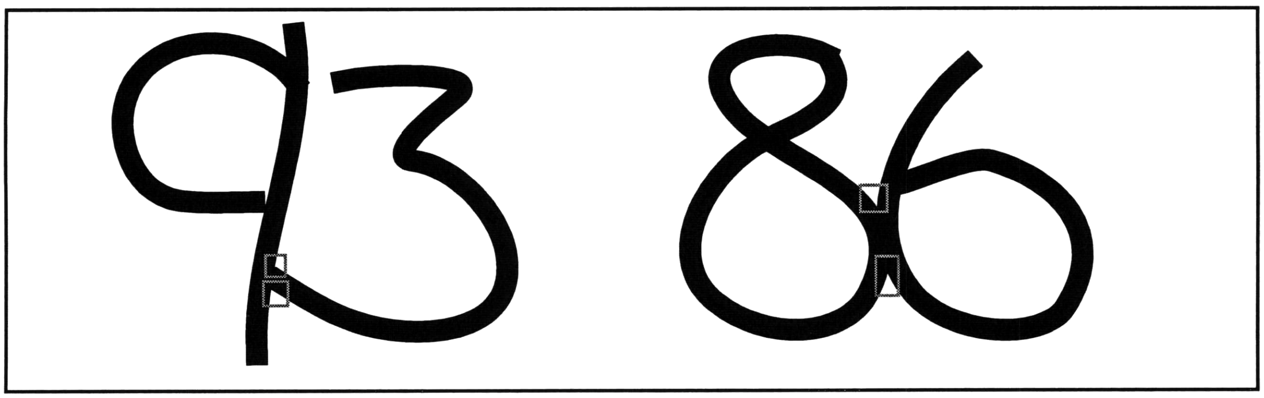

**Figure 4: Structural features which could be entry or exit points of intersecting strokes**

The segmentation method developed by Strathy *et al.* in [9] is a prime example of a relatively successful feature-based algorithm. This method attempts to identify structural features which could signify entry or exit points of overlapped or touching strokes (Figure 4). It takes into consideration the fact that the strokes could be straight or overlapped. The routine starts off by identifying significant contour points (SCP) which are either: 1) corner points, 2) local minima, 3) local maxima, or 4) a potential exit point of a straight stroke created by extending a contour through a concave corner (Figure 5). From this pool of SCPs, the algorithm selects a pair of points which could be the entry and exit points for the best cutting path. The points are chosen based on features such as:

- whether one is a minima and the other is a maxima
- corner points are preferred
- the closer the points are to each other the better
- the sharpness of the concativity at the SCPs
- the degree to which a minima is directly above a maxima
- the distance of a minima/maxima to the next local maxima/minima
- preference for pairs closer to the left side of the image

This method was tested on a small set of USPS zip codes and resulted in accuracies ranging from 48 to 97% [9]. Although these numbers are encouraging, one has to keep in mind that segmenting zip codes is an easier problem then check amounts since zip codes have a constant number of characters and are usually better spaced than check amounts.

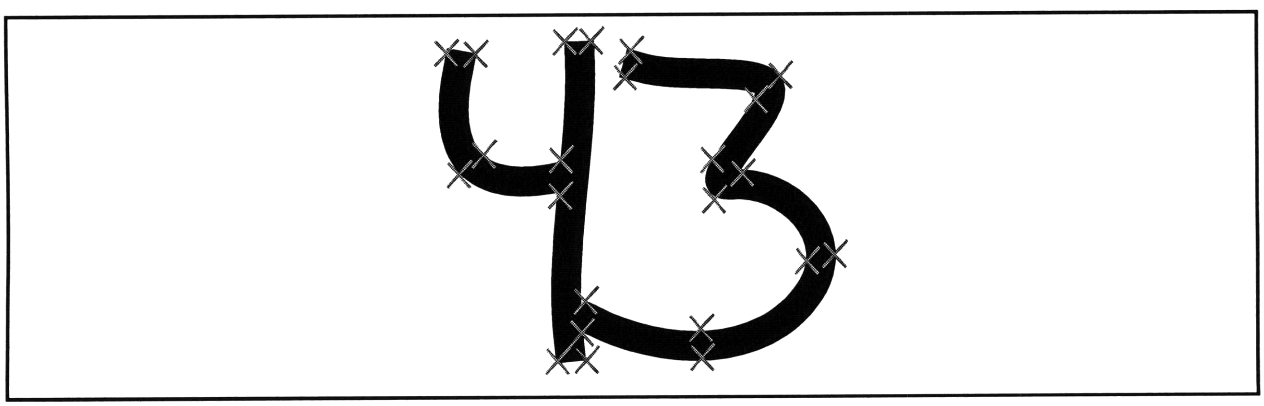

**Figure 5: Significant contour points (SCPs)**

### 3.4 Min-Max algorithms

Min-max algorithms are some of the most common employed in segmentation systems today [1, 5] and can be considered a subclass of structural feature-based methods. They are bases on the premise that most touching or overlapped characters are joined where the distance from the upper contour to the lower contour of the connected component is a minimum. The algorithm finds this minimum by examining the extrema on the upper and lower contours. In particular, it identifies the minima in the upper contour and the maxima in the lower contour. Then, a straight-line path is picked between an upper-contour minima and a lower contour maxima. The decision as to which minima and maxima to connect is usually based on values such as the distance of the cut from the center of the image, the distance between the extrema, and the number of white-black or black-white transitions encountered along the cut.

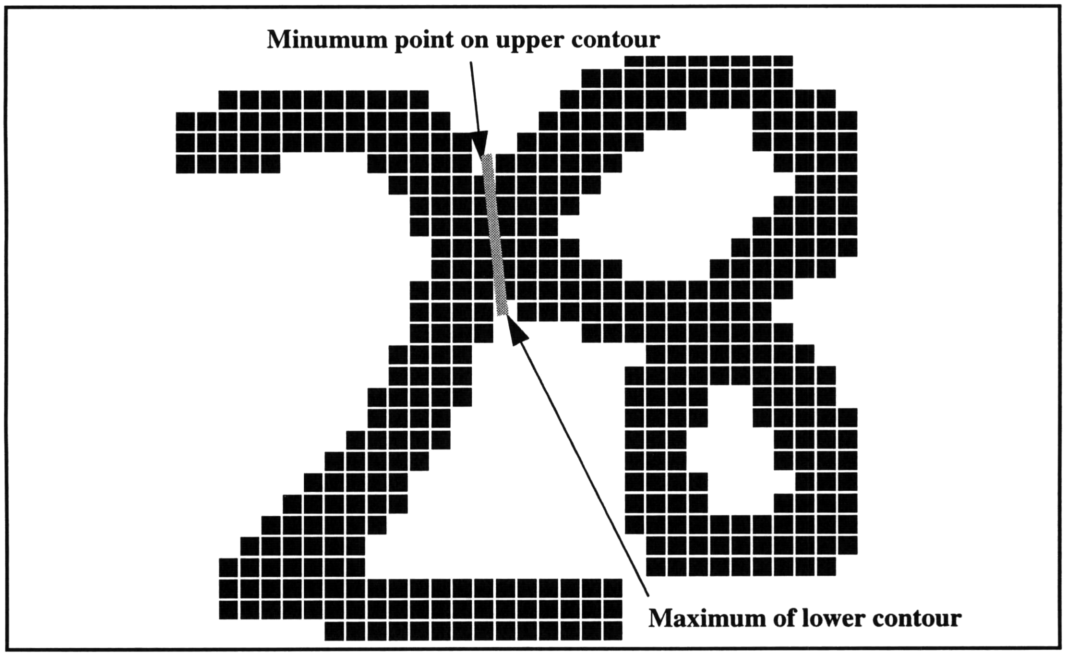


**Figure 6: Example of a cut produced by a min-max based heuristic**

### 3.5 The need for developing more sophisticated segmentation methods

The heuristics described in this chapter work reasonably well for a large number of cases, but they also fail badly for a lot of other cases (Figure 7). Also, the 'global' approaches such as structural feature-based segmentation tend to be extremely computationally expensive as opposed to the more local "Hit and Deflect" and drop fall approaches. Because of this, it is useful to explore ways of modifying and combining these heuristics so as to produce better results more often. It is this search for a more optimal combination of these heuristics which is one of the central points of this thesis. The hybrid algorithm developed is the subject of the next chapter.

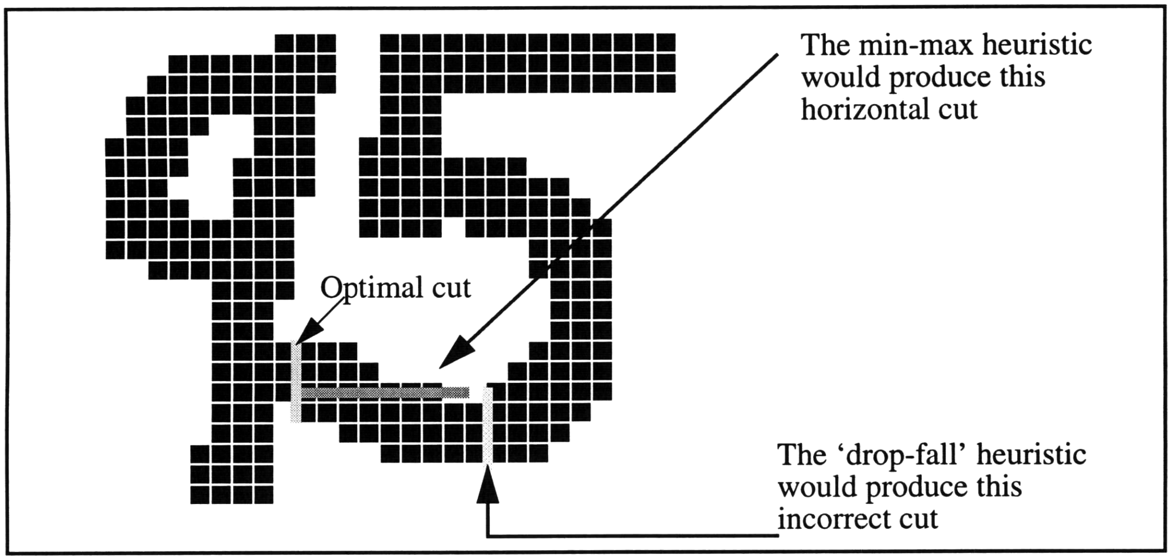


**Figure 7: Both 'drop-fall' and min-max heuristics fail on the above example**

# Chapter 4: The Hybrid Drop Falling Technique

Drop falling algorithms are based on the principle that a fairly optimal 'cut' between two connected characters can be made if one were to role a hypothetical marble off the top of the first character and make the cut where the marble falls. Despite its apparent simplicity, the algorithm has proven itself to be quite useful. In this chapter, the basic steps involved in drop-falling will be explored. We will also take a look at a new heuristic based on the drop-falling principle which is more effective.

## 4.1 The traditional "drop-falling" heuristic

As mentioned above, the drop-falling heuristic is based on the principle of letting a hypothetical marble fall in between two connected characters and making the cut where the marble lands. Based on this simple description of the method, the main issue which needs to be addressed in its implementation is where to drop the marble from. This is important since if the algorithm starts in the wrong place, the 'marble' could easily roll down the left side of the first digit or the right side of the second digit and, thus, would be completely ineffective (Figure 8).

There are several methods available to decide where to start the drop-falling process from. Obviously, it is best to start as close as possible to the point at which the two characters are connected. Dimauro *et al* [1] outline a method which does this quite robustly. In this method, the pixels are scanned row-by-row until a black boundary pixel with another black boundary pixel to the right of it is detected, where the two pixels are separated by only white space. This pixel is then used as the point from which to start the drop-fall (Figure 9).

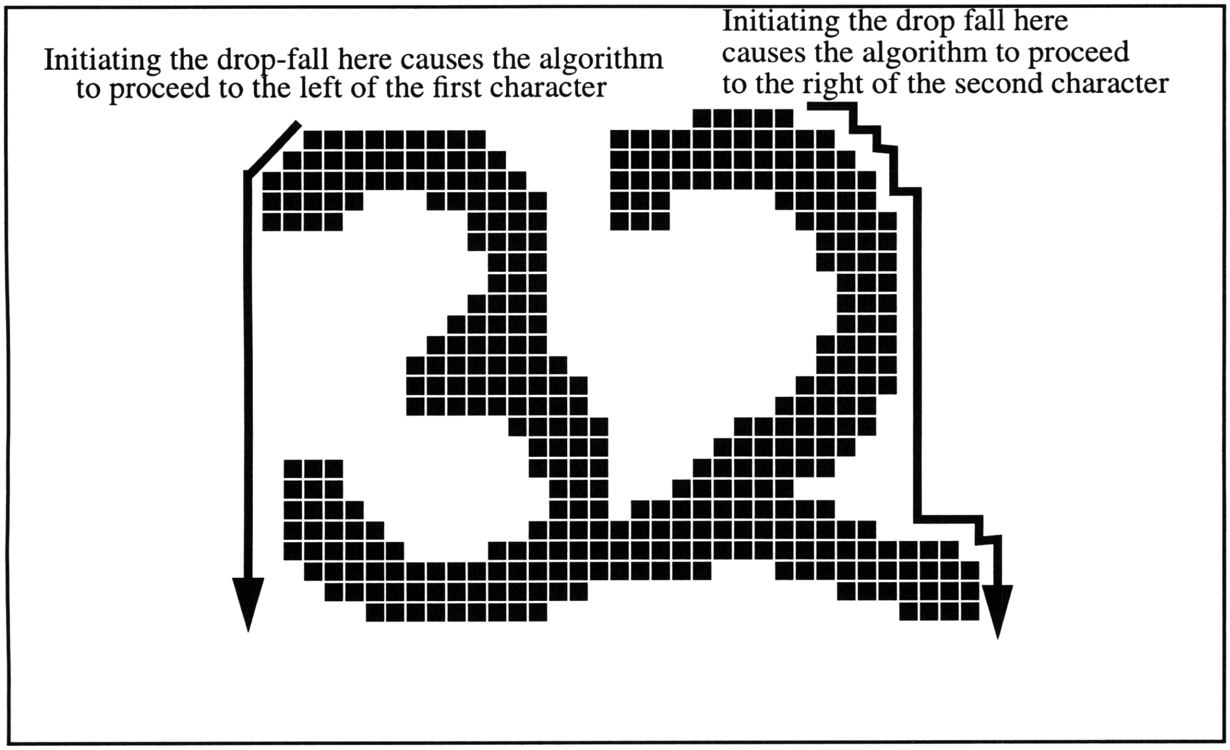


**Figure 8: Examples of bad initial points from which to start the drop-fall**

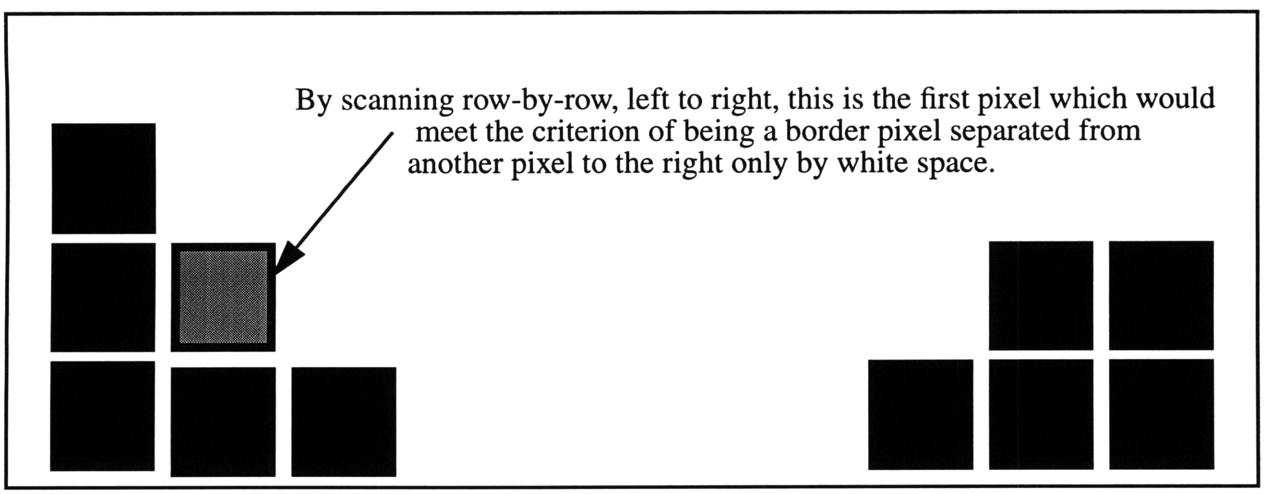


**Figure 9: Pixel from which to commence the drop-fall**

A more naive choice for the initial pixel would be the first pixel found by scanning row-by-row which has white space to the right of it. This method fails, however, in when the first such pixel encountered is part of the second of the two connected characters. In this case, the algorithm will 'fall off' the right side of the characters as depicted in Figure 8.

After the initial pixel is found, the next step is to begin the actual drop fall. The drop-falling algorithm is designed to mimic falling, so it will always move downwards, diagonally downwards, to the right, or two the left. The directions that the algorithm will 'move' in according to the current pixel position and its surroundings are outlined in Figure 10.

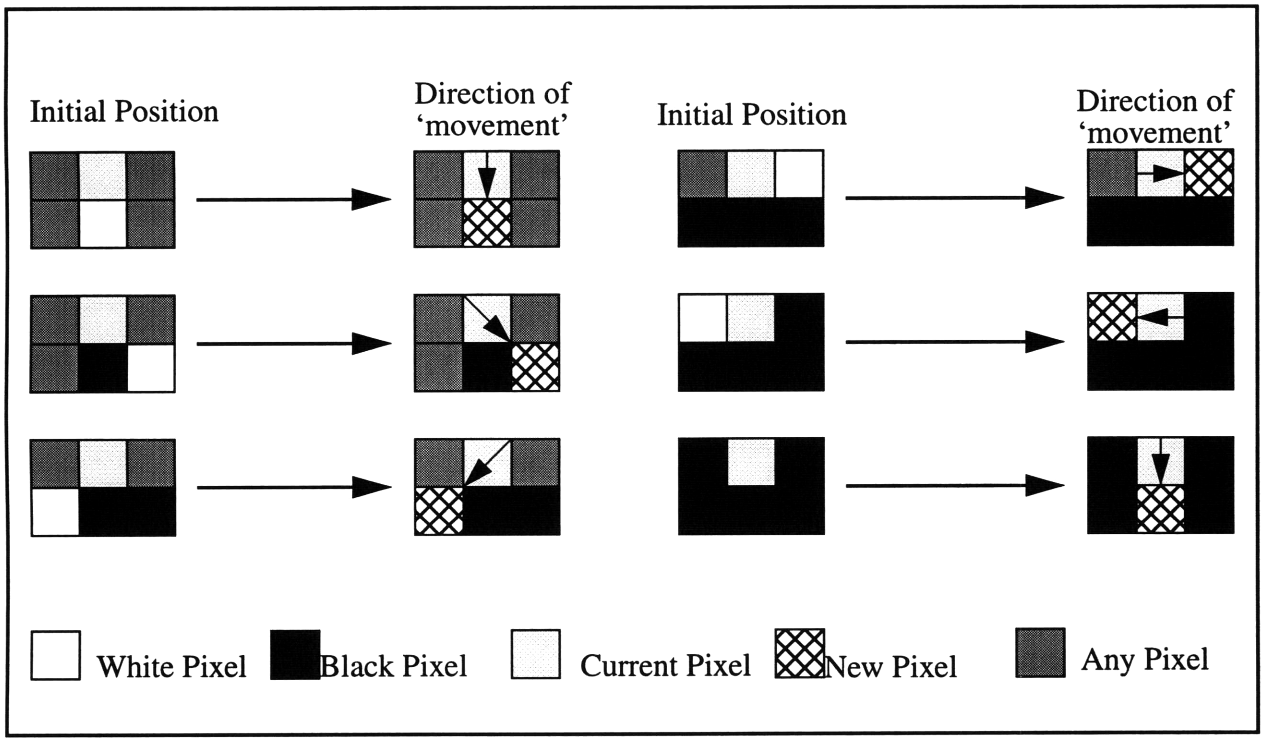


**Figure 10: Pixel-to-pixel path construction based on current position**

As we can see from Figure 10, the algorithm mimics a falling motion. When obstructed by a black pixel directly below it, the algorithm attempts to move to the right and then to the left. In the case when the current pixel is completely obstructed by black pixels, the algorithm bores directly down through the pixels. The algorithm continues to move from pixel to pixel in this fashion until the bottom of the image is reached.

## 4.2 Variations of "drop-falling"

The standard version of the drop-fall algorithm described above falls down and to the right from the top left of the connected component. There are three other variations of the algorithm accept for the fact that they don't necessarily initiate from the top left or fall 'down'. The differences will be made more clear as we look at the three other variations.

- **Top-Right Drop Fall:** This algorithm is identical to the standard drop-fall accept that it initiates from the top-right of the connected component rather than the top left. Also, instead of falling down and to the right, it falls down and to the left (Figure 11b). The standard drop fall algorithm can be used if the input image is 'flipped' vertically (Figure 11a). The resulting segmentation path **P** can be obtained through the transformation of equation (1) where $\mathbf{Pinv}_x$ is a vector of the x coordinates (or column indices) of the segmentation path resulting from the standard drop fall algorithm being performed on the vertically inverted image; $\mathbf{Pinv}_y$ is the vector of the y coordinates (or row indices) of the segmentation path resulting from the standard drop fall algorithm being performed on the vertically inverted images; and $w$ is the width of the image.

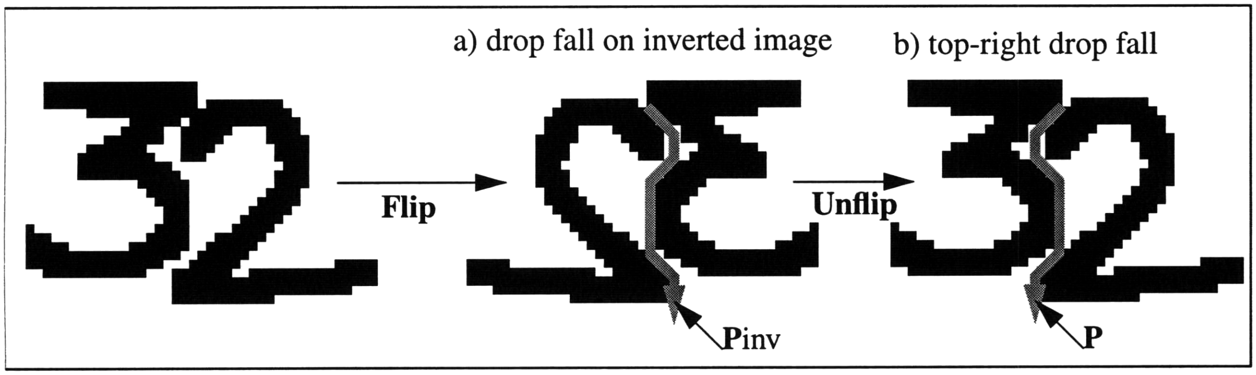


**Figure 11:Implementation of (b)top-right drop fall by applying (a) standard drop fall to an inverted image**

$$\mathbf{P}_x^{\,i} = w - \mathbf{P}\mathrm{inv}_x^{\,i}$$

$$\text{for } i=1, 2, ..., n \text{ where n is the length of } \mathbf{P}\mathrm{inv} \quad (1)$$

$$\mathbf{P}_y^{\,i} = \mathbf{P}\mathrm{inv}_y^{\,i}$$

- **Bottom-Left Drop Fall:** This algorithm is identical in principle to the original drop-fall algorithm accept that it initiates from a pixel at the bottom left of the image and ‘falls’ up and to the right (Figure 12b). This algorithm can be implemented by performing a standard drop-fall on an input image which is flipped along its horizontal axis (Figure 12a). The bottom-left drop fall path can then be obtained using equation (2) where $h$ is the height of the image.

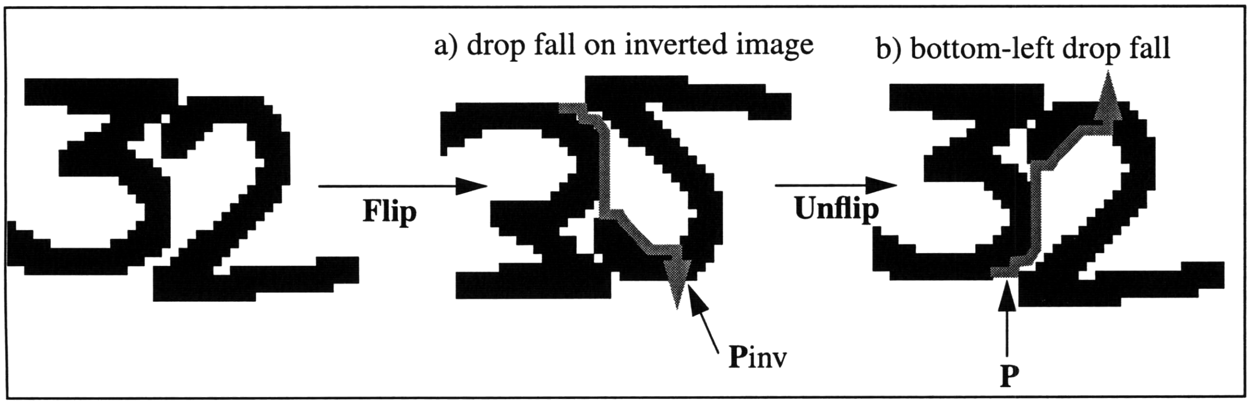


**Figure 12:Implementation of (b)bottom-left drop fall by applying (a) standard drop fall to an inverted image**

$$\mathbf{P}_x^{\,i} = \mathbf{P}\mathrm{inv}_x^{\,i}$$

$$\text{for } i=1, 2, ..., n \text{ where n is the length of } \mathbf{P}\mathrm{inv} \quad (2)$$

$$\mathbf{P}_y^{\,i} = h - \mathbf{P}\mathrm{inv}_y^{\,i}$$

- **Bottom-Right Drop Fall:** This splitting heuristic is identical to the pervious three accept that in this case, the falling commences from the bottom right of the image and goes in an up and leftward direction. It can be viewed as the exact opposite of the standard top-left drop fall. As before, a bottom-right drop fall (Figure 13b) can be implemented by performing a standard

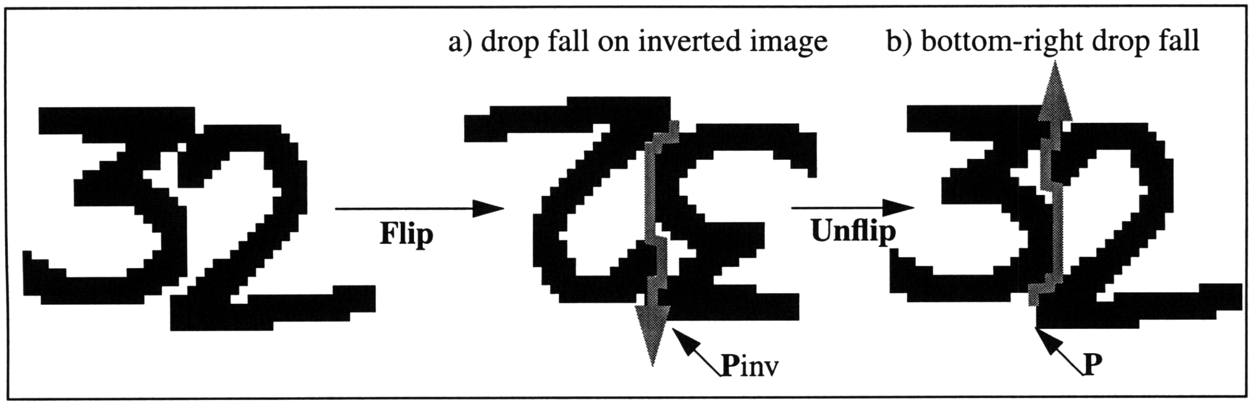


**Figure13:Implementation of (b)bottom-right drop fall by applying (a)standard drop fall to an inverted image**

top-left drop fall after appropriately transforming the input image (Figure 13a). This time, the image must be flipped in both the horizontal and vertical directions. The bottom-right path P can be obtained from the transformation of Pinv given in equation (3).

$$\mathbf{P}_x^{\,i} = w - \mathbf{Pinv}_x^{\,i} \qquad \mathbf{P}_y^{\,i} = h - \mathbf{Pinv}_y^{\,i} \qquad \text{for i=1, 2, ..., n where n is the length of } \mathbf{Pinv} \qquad (3)$$

### 4.3 Evaluation of the variations of drop falling

Despite the apparent similarity between the variations of the drop-fall heuristics outlined above, they do very often provide very different segmentation paths for various test examples. For any of the drop-fall heuristics, it is easy to find cases where they work well and where they don't work well. It is however, much more difficult to find examples when all of them do not work well. In the next section, this principle will provide the basis for the construction of a hybrid heuristic which makes use of all of the aforementioned drop-fall variations. Now, we will look at the kinds of examples where each of the drop-falls tend to succeed or fail.

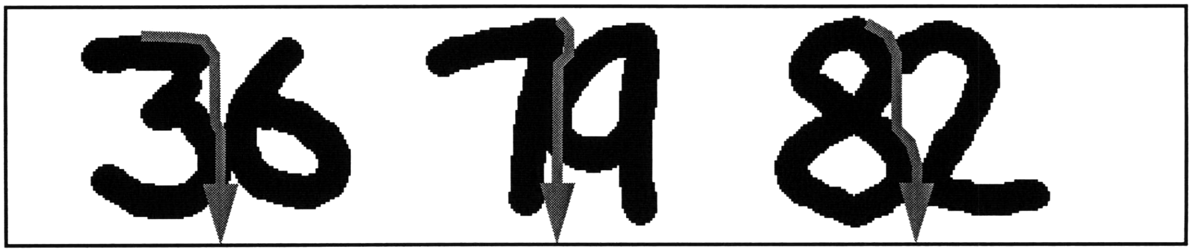


**Figure 14: Examples for which the top-left drop fall heuristic succeeds**

First, we will focus on the standard top-left drop fall. At first glance, this seems to be a very robust method for segmenting two connected characters (Figure 14). However, one does not have to look too long to find test examples where it fails rather miserably (Figure 4). Figure 15 shows several more examples of when the top-left drop fall heuristic will fail.

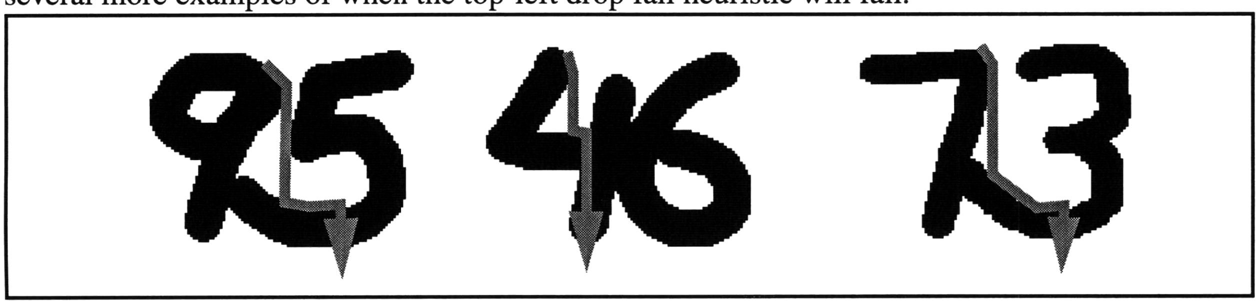


**Figure 15: Examples for which the top-left drop fall segmentation heuristic fails**

As seen from Figure 14, the top-left drop fall tends to work well in cases when the point at which the two characters are joined are a minimum point and it fails when it is not. From Figure 15, we can see that the heuristic will usually fail either when a minimum point does not occur at the point of connection or overlap of the two characters (as is the case with the '95' and '73') or the point of initiation (where the drop fall starts from) is not close enough to the point where the two characters touch (for example, the '46' in Figure 15). The second of the two failures can be corrected by formulating more advanced heuristics for deciding where to initiate the drop fall from. Once such heuristic would be to give preference to candidate initiation points which are closer to the center of the image. It is much harder, however, to make corrections for failures of the first type which tends to happen anytime the second character is a 3 or 5 since both of these numbers have a minimum on their lower strokes which are accessible to the drop fall algorithm (An 8 has a minimum on its lower stroke, but the drop fall algorithm not get to it since its lower stroke is closed). Using other varieties of the drop fall heuristic (top-right, bottom-left, and bottom-right) we will see that it may be possible to correct these failures.

The top-right drop fall tends to succeed and fail on the same cases as the standard top-left algorithm. There are however, some special cases when one can succeed and the other fails (Figure 16). However, these cases tend to be very concocted, so for all general purposes they are equivalent in terms of effectiveness.

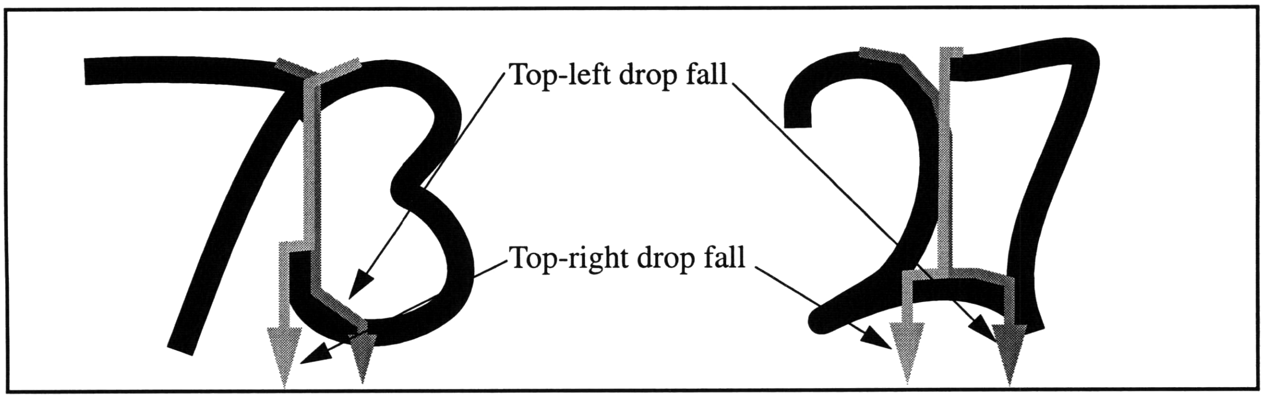


**Figure 16: Examples of when top-right and top-right drop-fall differ in results**

As mentioned earlier, the drop falls which start from the bottom of the image (bottom-left and bottom-right) tend to succeed in most of the cases where the top-based drop falls fail (Figure 15). They also, however, fail on some examples that the top-based drop falls would segment properly (Figure 16). Based on the fact that the top-based and bottom-based drop falls ‘complement’ each other, it is clear that a heuristic which makes use of both of them could potentially be significantly better than any of the drop fall heuristics described so far.

### 4.4 Hybrid drop fall heuristic

Based on the evaluation of the four variations of the drop fall described in the previous section, it is clear that each has advantages over the other. In particular, the top-based and the bottom-based downfalls can each successfully segment cases that the other can't. With this in mind, we will design a hybrid drop fall heuristic which combines the best of both types of elements.

The algorithm will initially attempt a top-left drop fall on the image. Then it will attempt to determine if the algorithm performed a successful segmentation. In order to cover as many cases as possible, the hybrid algorithm needs to be able to recognize when one variety of the drop fall fails in order to know when to use one of the other variations. We could potentially use a character recognizer at this point to determine if each of the resulting characters are legible numeral. This, however, would be computationally expensive and would not necessarily even give a good indication of the quality of the segmentation. Instead, we will make use of the some of the observations about cases where the top-right drop fall fails we made in the last sections to determine whether or not the segmentation was successful.

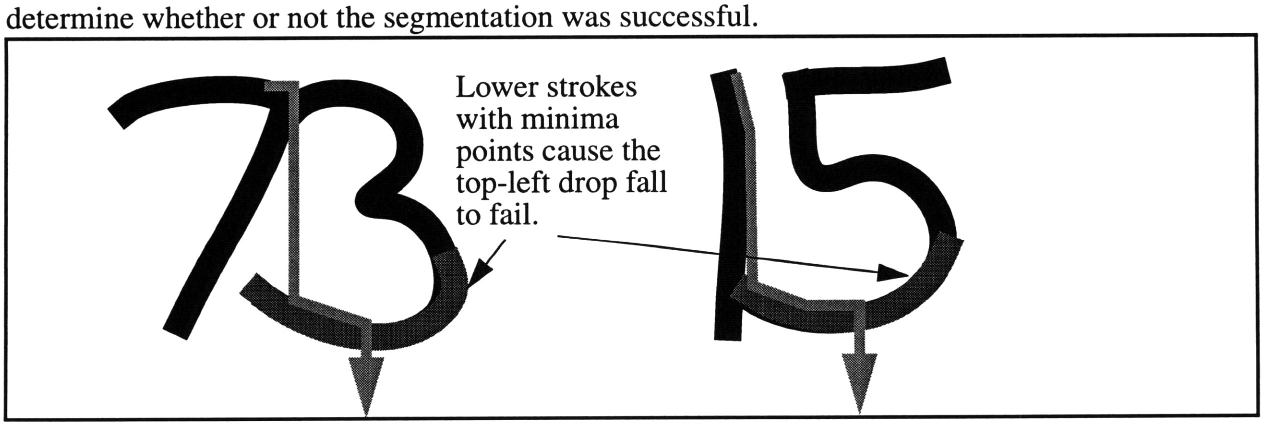


**Figure 17: Concave lower strokes on characters like '3' and '5' cause the top-left drop-fall algorithm to fail.**

As we saw in the previous section and can see in Figure 17, the top left drop-fall tends to fail when the second character has a local minima in its lower stroke. This is true for all threes and fives. Since we cannot know a priori whether or not the second digit is a three or five, some other means for recognizing such a failure has to be established. If we look at the minima of lower strokes of characters versus the minima at the junction point between two characters, there is one fundamental difference-- the minima at the lower stroke of a three or five tends to be very gradual and flat while the minima at the junction point between two characters tends to be very sharp and steep (Figure 18). Based on this observation, all that the algorithm would need to do is recognize whether the minima at which the top-left drop fall begins to cut into the characters is sharp and steep or flat and gradual. If it is sharp and steep, then the top-left drop fall is most probably sufficient. On the other hand, if the minima is flat, then the top-left drop fall probably failed by cutting through the lower stroke of a three or five. In this case, a bottom-right drop fall will be applied. As we saw in the previous section, the bottom-right drop fall usually properly segments cases such as the one described when a top-left drop fall would fail.

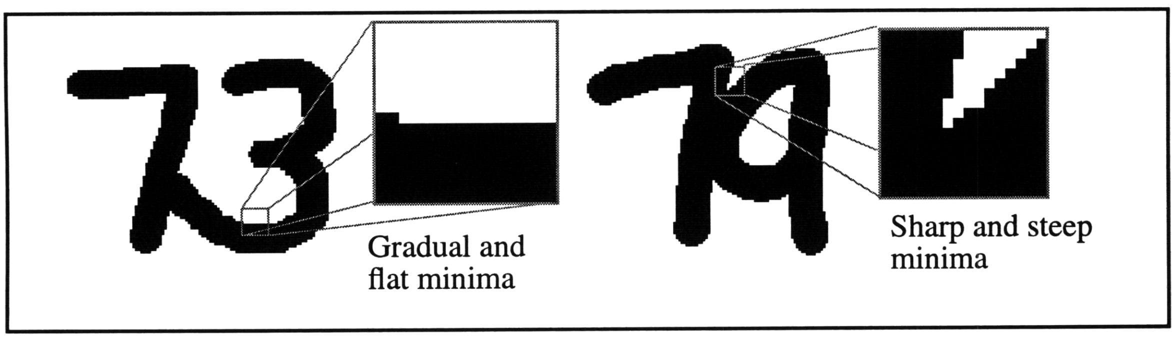


**Figure 18: Differences between minima on the lower stroke of characters and minima at the junction of two characters.**

Even though we know that the top-left drop fall tends to fail when it begins to cut at a flat minima and that it tends to succeed when it cuts at a steep minima, a method for recognizing steep versus flat minima must be developed. The best way to do this would be to travel along the contour of the characters a given number of pixels to the left and to the right. If the contour rises in either direction more than a certain threshold than it is classified as being steep. If it does not, then it is flat (Figure 19). Between 3 and 5 pixels should be travelled in each direction along the contour, depending on the size of the image. The threshold should be half the number of pixels travelled. Based on this, if the slope in either direction is greater than 1/2, then it is steep. Otherwise, it is flat.

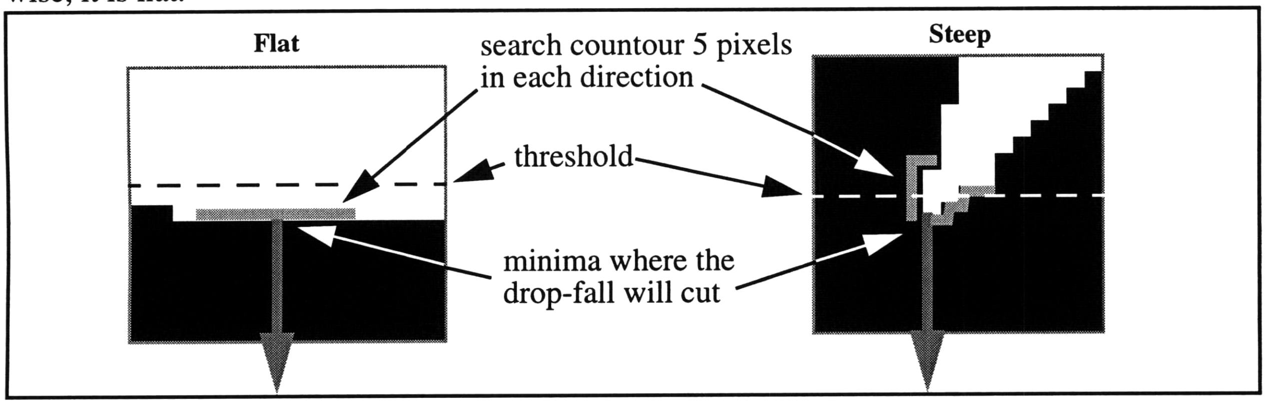


**Figure 19: Using the slope of the neighboring contour to determine if the minima is at a 'steep' or a 'flat' point**

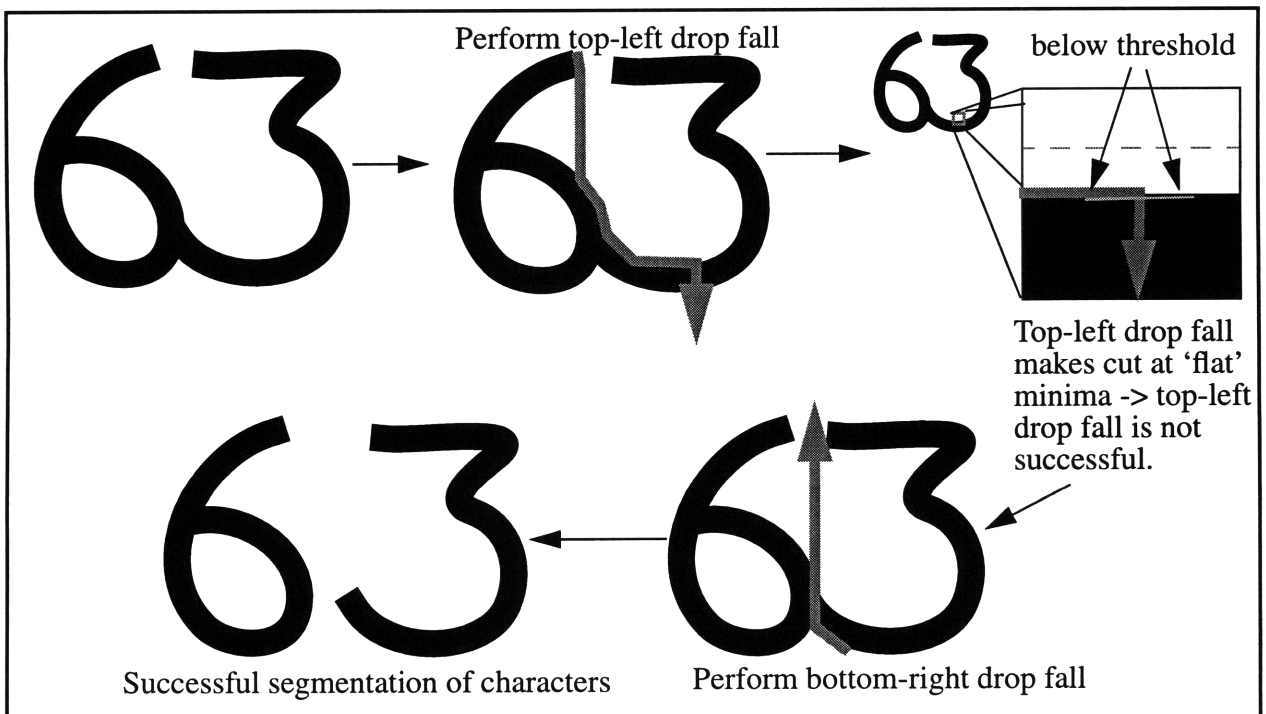


**Figure 20: Performing bottom-right drop fall when top-left drop-fall is recognized to fail**

# Chapter 5: Development and Design of the Segmentation Toolbox

The ideas described in this thesis were implemented into a segmentation workbench affectionately called the Segmentor. It is a graphical application whose main purpose is to ease experimentation with various heuristics for segmentation. Having a stand alone application dedicated to segmentation dramatically reduces the complexity of changing the methods for segmentation. Not only that, but experimenting within the context of a optical character recognition system could potentially break the entire system. For example, if the recognizer portion of the system is neural-net based, then modifying one heuristic in the segmentation process may force you to retrain the entire neural net as well. Both the Segmentor's graphical user interface as well as its underlying code were implemented in Matlab.

## 5.1 Functionality of the segmentation tool box

The Segmentor's main utility lies in its ability to provide any easy way to experiment with methods for segmentation. Its design is focused around the following three features:

- **Flexibility:** As mentioned above, the main purpose of building the Segmentor in the first place was to provide an easy way to test various segmentation methods. Because of this, it is of prime importance that the system be made flexible. This issue is addressed in the design of the Segmentor by allowing the user to add or define his own image formats; change the algorithms used for any step of the segmentation process; and easily incorporate the Segmentor routines into another application. In fact, the system is designed to allow users to either use the predefined graphical user interface or prototype their own.
- **Extensibility:** The system would be of little future use if future users cannot add on their own routines and functionality. The Segmentor takes this into consideration by allowing users to

add on new routines, modify current routing, and add completely new modules to the Segmentor.

- **Ease of use:** In order to optimize the usefulness of the segmentor, much consideration was taken to ensure that it is easy to use, modify, and add on to. To a large degree, the extensibility and structure of Matlab's pre-defined graphical user interface library makes this possible. The design of the system is also extremely modular so as to allow components and routines to be easily replaced and modified.

Within the above-mentioned design considerations, the Segmentor will allow user to do anything from performing a single segmentation step to the entire segmentation process on a character string image. It also allows the user to dynamically change variables used for segmentation without modifying any code. Here are three scenarios depicting possible uses for the Segmentor:

- If a user wishes to build and test a new algorithm to dynamically threshold a character string image, then the Segmentor could significantly reduce the work involved. First, in order to get an estimate of what threshold values work best for which type of images, the user could dynamically adjust the threshold value (using the left-most slider control depicted in Figure 21) and repeatedly perform connected component analysis. Doing this will allow the user to get a 'feel' for the task. Then, he or she could develop a prototype matlab function which implements a thresholding algorithm and connect the function to one of the workbench buttons. Critical variables in the algorithm can then be assigned to the slider so as to allow dynamic modification of the variables without modifying the code. Also, the inputs and the outputs of the algorithm could be easily viewed and manipulated.
- Another use of the Segmentor would be to compare various segmentation algorithms with each other. In this case the user could load an array of test images into the segmentor and then

apply the two algorithms to all of the images. The resulting segmented images could then be displayed side by side for each of the algorithms.

- With the scope of the research performed in this thesis, the Segmentor was used to prototype a new hybrid, drop fall algorithm. Using the Segmentor allowed the algorithm to be developed relatively quickly without having to repeatedly recompile code or worry about unwanted side effects on other components of a handwriting recognition system. Only after the algorithm was developed and tested using the Segmentor, was it hardcoded in C++ into the complete handwriting recognition system.

## 5.2 Structure and implementation of the graphical user interface

The Segmentor's main purpose is to be an easy to use application for studying segmentation heuristics. With this goal in mind, it is of utmost importance that its interface be intuitive and simple. The interface consists of a main menu, a controls area, and a display area. The main menu contains options which allow the user to select new image files, print the display, or exit the application. The image file selected from the menu will be the one used in all future operations. The controls area is where the user is allowed to define segmentation steps and control the entire process in general. Using the controls in this area, the user is able to step through various segmentation steps, and manipulate thresholds and other variables involved in segmentation. Finally, the display area is where all of the segmentation data are shown to the user.

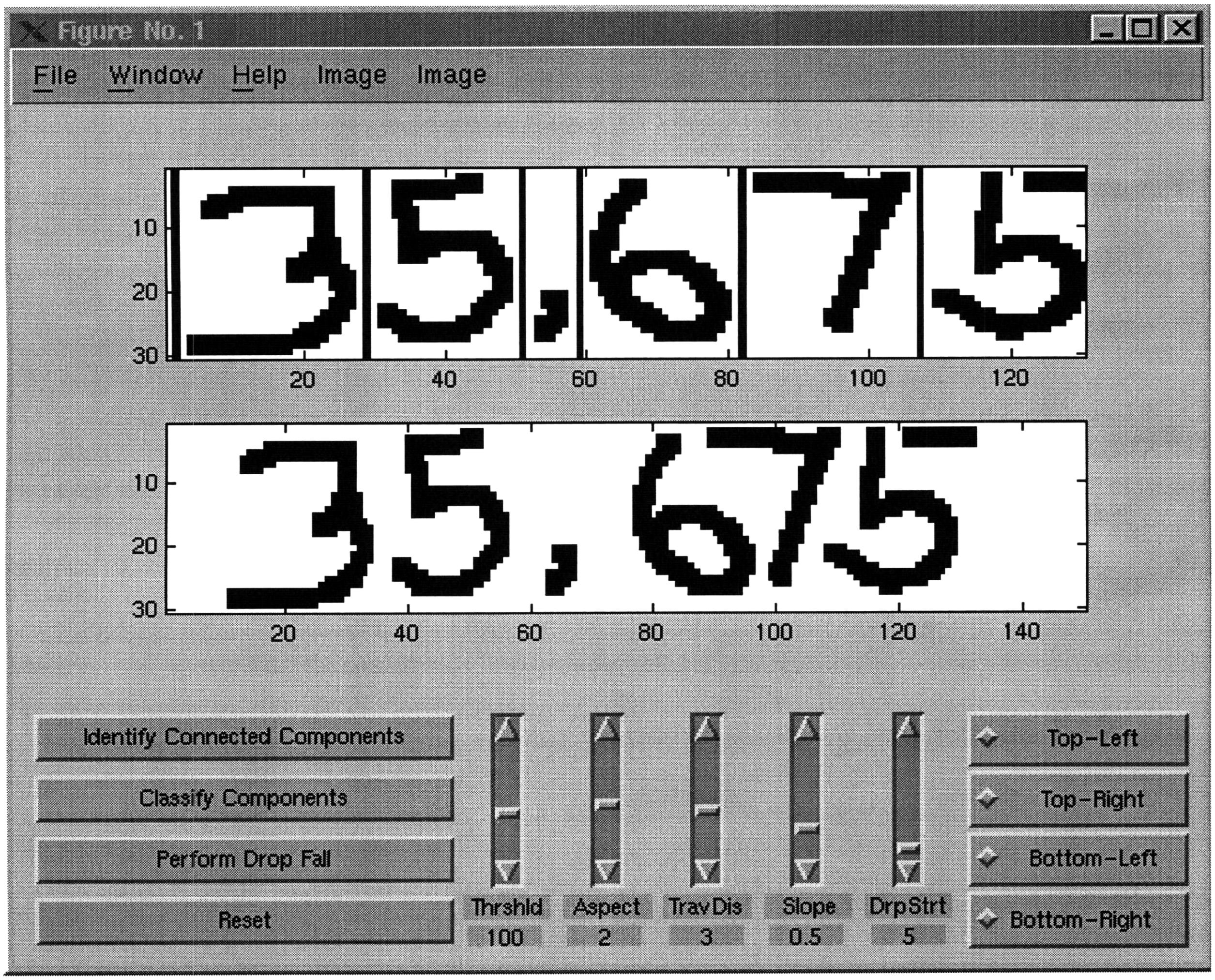


**Figure 21: Image of the Segmentor's graphical user interface**

## 5.3 Image formats and representations

The Segmentor tool box is capable of reading any of the standard image formats as the inputs, including JPEGS, TIFFs, and BMPs. The images are read using the *imread* (part of the matlab image manipulation library) utility in matlab which can convert any of the aforementioned image formats to Matlab matrices. In designing the tool box, an assumption is made that the images used will be only of the courtesy amount and not include any extraneous noise or characters. This assumption can be made due to the fact that there are already well-established algorithms for locating the courtesy amount on a check [2]. It is also assumed that the input image is a 256-shade gray scale image.

### 5.4 Implementation of connected component finder

The first step in segmenting any character string is to separate out the connected components. This is the most basic, and potentially most important step. Ideally, if all of the characters are well-spaced, then finding the connected components would be equivalent to segmenting the whole string. In the Segmentor application, connected components were identified using basic search algorithm.

The image is first scanned column-by-column to identify the first non-white pixel. A non-white pixel is recognized to be any pixel with an intensity less than 100. This threshold value can either manually set or dynamically adjusted depending on the average intensity and contrast of the image. The image is scanned column-by-column as opposed to row-by-row to maximize the chances that the first pixel encountered belongs to the first digit of the character string (Figure 21). The only time that this may not work properly is if the top stroke of a 7 completely covers the preceding character (Figure 22).

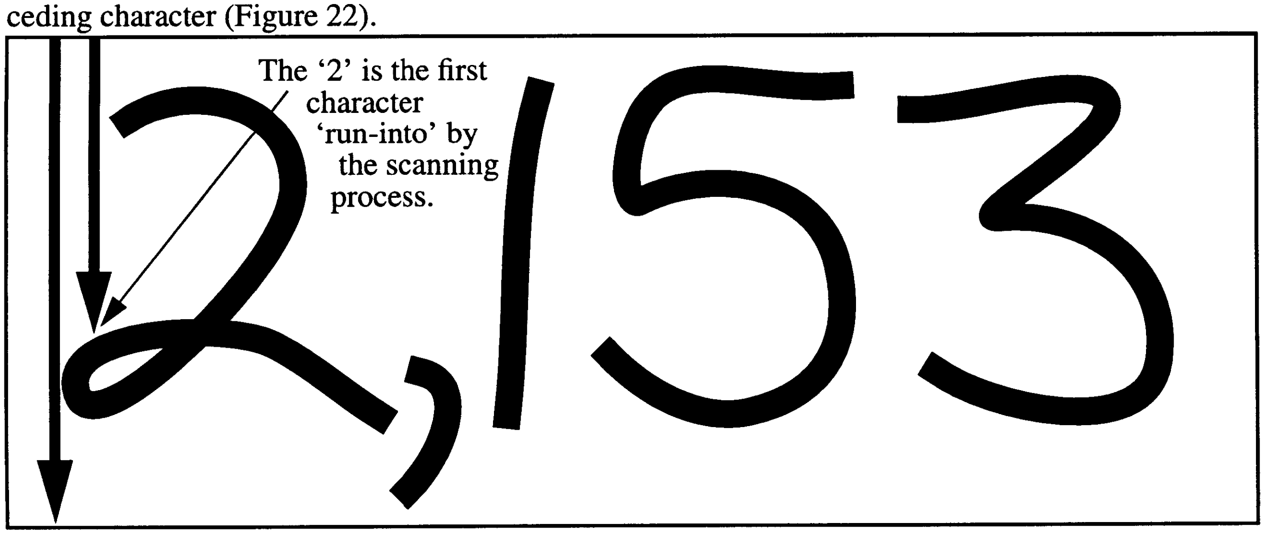


**Figure 22: (a) Image is scanned column-by-column to detect connected components in order**

Once the initial black pixel is found, a bread first search is performed to identify all of the pixels connected to it. For the purposes of the breadth-first search, a pixel is considered connected to another pixel if it lies directly above, below, to the right, or to the left of the given pixel. Two pixels lying diagonal to each other are not immediately identified as being connected due to the common case depicted in Figure 23. Also, it is unlikely that two pixels are from the same character if they are only diagonal to each other (i.e. they share no other neighbors).

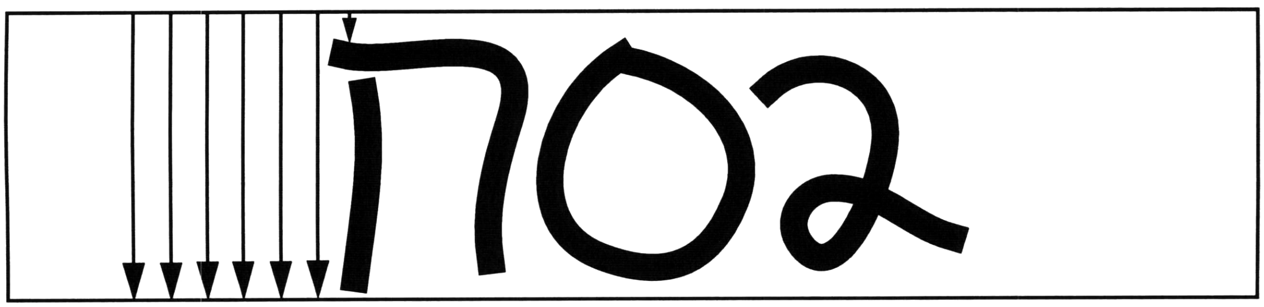

**Figure 22: Scanning method fails only in the unusual case of a '7' hanging completely over the preceding character. This is compensated for by deciding character position by the centroid of their respective bounding boxes.**

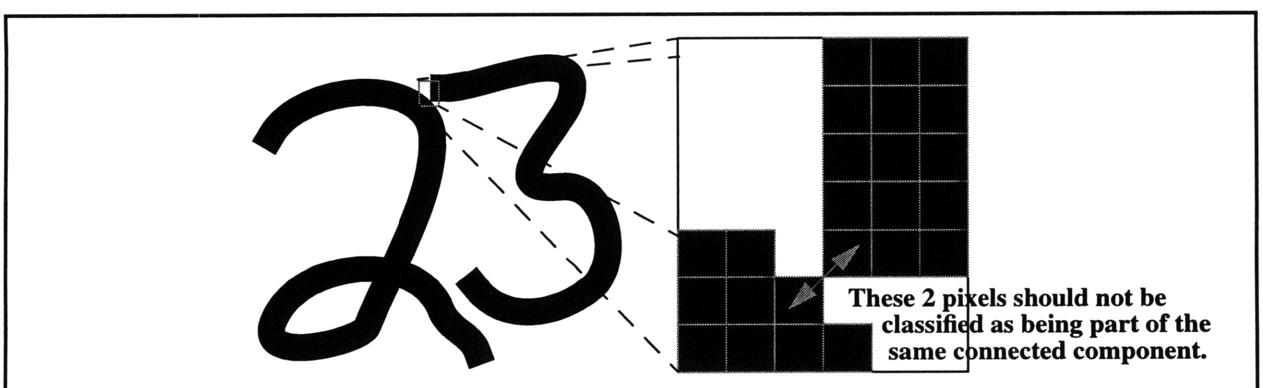


**Figure 23: Search algorithm does not move to diagonal pixels to avoid the case where characters are only connected by diagonal pixels**

The breadth-first search routine (see Appendix) operates by tagging each connected pixel it encounters and copying it into a buffer the same size of the original image. It also keeps track of the bounding box of the connected component. The routine suspends once it cannot find any new connected pixels. At this point, it crops the buffer based on the size of the bounding box.

Finally, it returns, the bounding box of the connected region, the modified image (with all of the connected pixels tagged) and the cropped copy of the connected region.

The connected component finder then continues to scan the image in the manner described above until it encounters another black pixel (which has not been tagged by the bread-first-search routine). The search routine is then executed again on this pixel, resulting in a new connected component. The cropped image of each new connected component is stored in an indexed table to facilitate further processing. This process is repeated until every pixel of the image has been scanned by the connected component finder.

### 5.5 Heuristic for recognizing and connecting disjointed fives

One of the most common occurrences of a numerical character being broken into two connected components is when the top stroke of a '5' is disjointed from the body of the numeral (Figure 24).

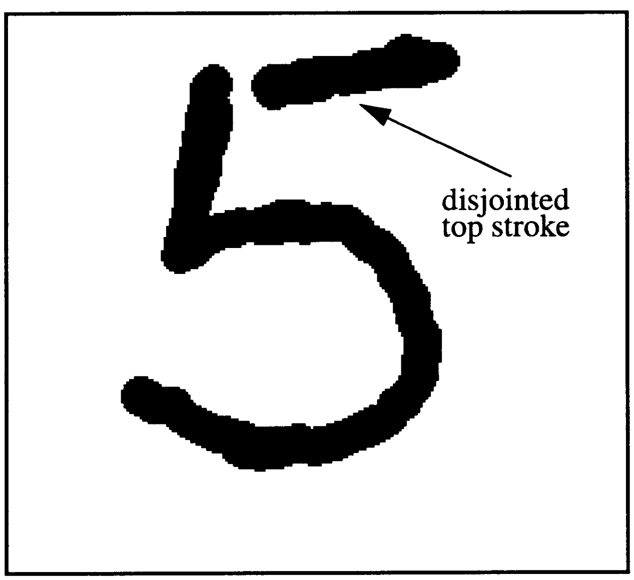


**Figure 24: Example of a '5' with a disjointed top stroke**

The Segmentor uses a simple heuristic which makes use of the size and position information of the connected components to make a decision as to whether a given connected component is a disjointed top stroke of a 5. The first test used is the aspect ratio (The ratio of the height to the width) of the connected component. If the width is greater than two times the height, then the component is candidate for being a disjointed top stroke of a 5. Then, if the component is 1/3 the size of the previous component and is completely contained in the top half of the image, it is considered to be a disjointed top half of a 5. The logic behind the heuristic is as follows:

- If the width of a connected component is greater than its height, then the component can either be two or more connected components, an extremely slanted comma, a wide decimal point, a disjointed top stroke of a five, or the divider between the numerator and denominator of the cent field of the number (Figure 25).
-

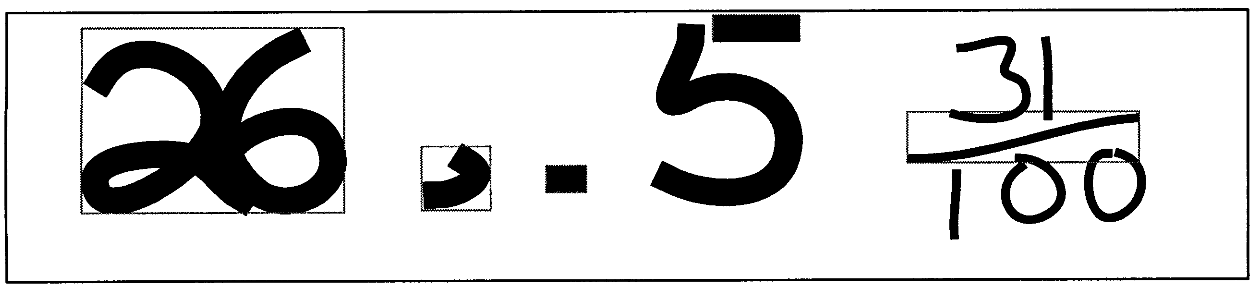


**Figure 25: Examples of connected components for which the height is less than the width**

- If the component is one-third the height of the previous character, then the possibility that it is two or more connected characters can be ruled out (It would be extremely unusual for one entire character to be less than one third the height of the previous character).
- Finally, if the component is completely contained within the upper half of the image, then the probability that it is a comma, period, or division character of cent field is extremely low. There are special cases where the division sign of the cent field can meet the above criteria. To prevent such a character from being recognized as a disjointed top stroke of a 5, an added check can be made as to whether any other connected components are contained entirely

above the character (Only the division character in the cent field would have connected components contained entirely above it-- namely, the numerator).

Once a component has met the above-defined criteria, and thus is classified as being the disjointed top stroke of a five, it is pasted on top of the previous component so that they are later processed together as one character. In order to optimize the recognition of the disjointed five later on, a connection routine could be used to connect the top stroke with the rest of the five. Currently, such a connection routine is not implemented in the Segmentor, but a possible implementation of it could be as follows (Figure 26):

1. Identify the left-most pixel of the top stroke of the five.
2. Identify the maximum point on the body of the five which is closest to the point identified in step 1.
3. Connect the two points with a width equivalent to the height of the top stroke.

The above defined heuristics are extremely robust and only fail in very unlikely instances. The performance and examples of the type of connected components on which it would fail are described in more detail in the results chapter.

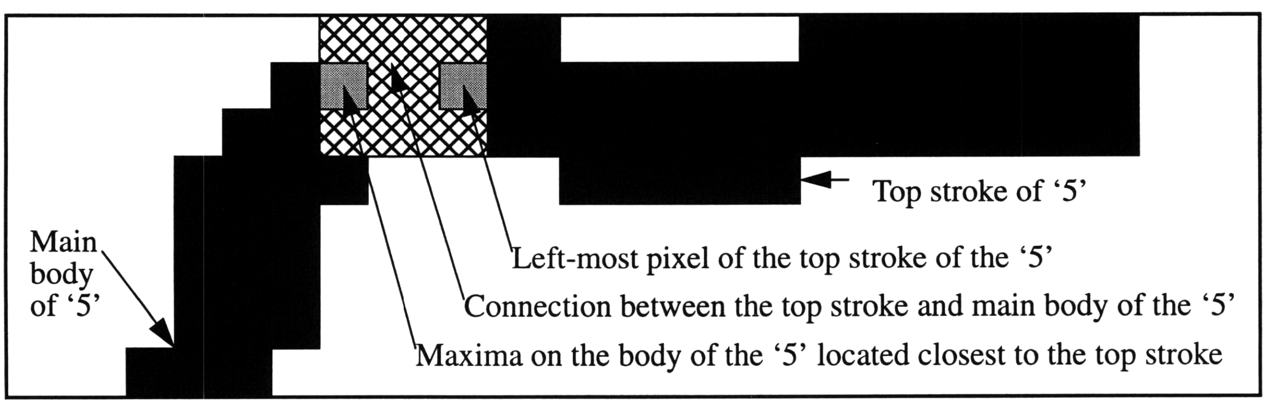


**Figure 26: Enlarged depiction of the 'bridge' formed between the main body and disjointed top stroke of a '5'.**

Once all of the connected components are identified and any disjointed top strokes of '5's are connected, the connected component finder module displays both the original image and an image of the connected components separated by a divider.

### 5.6 Heuristics for recognizing touching or overlapping characters

Currently, the best available methods for determining whether a connected component consists of two or more overlapping characters make use of the component's aspect ratio and position. The aspect ratio, or the width of the image to the height, tends to be greater than 1 for connected characters and less than 1 for single characters. Since, as mentioned in the previous section, other characters such as divider between the numerator and denominator of the cent field and a disjointed top-stroke of a five can have an aspect ratio greater than 1 as well, the position and size of the connected component are used as well to decide if the component is made up of multiple touching characters.

This procedure is implemented in the Segmentor as follows. As mentioned in Section 5.4, the connected component finder module of the Segmentor outputs the connected components in an indexed table. This table is then passed to the module which identifies which components are made up of touching or overlapping characters. This procedure then loops through the table, checking whether any of the aspect ratios are greater than 1. If they are, then the procedure checks the size of the bounding box. If the bounding box's area is greater than 500, then the component is classified as being composed of two touching or overlapping characters. For the sake of simplicity, the assumption is made that more than two touching or overlapped characters will not be encountered. The current method could be extended to the more general multiple-character case by including more clauses to which would consider even larger bounding box areas. The procedure then adds the index of that component to a newly created table which keeps track of which of the components need to be further processed. These components are then passed on to the module which attempts to perform the hybrid drop fall algorithm described in Chapter 3 on them.

### 5.7 Techniques for splitting connected characters

The Segmentor uses the hybrid drop-falling algorithm described in the previous chapter in order to segment connected characters. The module which takes care of this is passed the indexed table of connected components produced by the connected component finder as well as the table described in the previous section which indexes the connected components which need to be further segmented. The drop fall algorithm is then performed on each of these connected components as follows:

1. A one dimensional array, **cut[]**, which has length equal to the height of the image, is initialized. This array will store the pixels which represent the border between the two connected characters.
2. The image is scanned row-by-row until a black border pixel separated from another black pixel to the right of it by only white space (Figure 9) is identified. The x-coordinate of that pixel is stored in **cut[**y-coordinate of pixel**]**. Since there should be nothing but white space above this pixel, **cut[** 1, 2, ..., y-coordinate of pixel**]** is set to equal the x-coordinate of the pixel.
3. A top left drop fall commences from this pixel as described by Figure 10. The x-coordinate of each new pixel in the drop fall is stored in **cut[** y-coordinate of current pixel**]**. If there is already an x-coordinate written in that position in the array, then it is overwritten.

4. The drop fall continues until it reaches a minimum at which point it must begin to 'cut' into the black pixels below it. The algorithm will then traverse the border six pixels (the number of pixels traversed can be dynamically changed using one of the sliders on the graphical user interface) to the left and right. If the contour has risen by three or more pixels in either direction then the top-left drop fall continues. Otherwise, the **cut** array is reinitialized and a bottom-right drop fall algorithm is called.
5. If the algorithm continues, it will continue to drop fall as described in Figure 10 until it reaches the bottom of the image. The **cut[]** array will have a value for everyone of its entries at this point.
6. A blank image with height equal to the height of the connected component image and width equal to the maximum value in **cut[]** is created. The connected component image will be scanned row by row. For each pixel in a row, if it has an x-coordinate value less than the entry in the cut array for that row, then it is copied into the blank image. That pixel will also be erased from the connected component image.
7. The connected component image is cropped so as to only include the bounding box of what is left of the connected component.
8. At this point the first character will be in the image created in step 6 and the second character will be in the cropped version of the connected component image.

The top-right drop fall algorithm described in step 5 uses Matlab's matrix operations to reverse the indices of the image matrix in both height and width directions (In effect, flipping the image over on its vertical axis and then flipping it on its horizontal axis). It then performs essentially the top-left drop fall algorithm performed in steps 1-9 above without the check to see the steepness of the minima at which the cut begins.

When the top-left drop fall algorithm encounters a minima, it has to decide the 'concavity' of the minima (Step 4). This is done using a contour following procedure. First this performs the following steps which traverse the contour to the right:

1. If there is a white pixel to the right of the current pixel, then the algorithm sets the set the current pixel to the pixel to the right. Otherwise, the algorithm scans every pixel directly above that pixel until it find a white pixel and assigns that to be the current pixel.
2. Step 1 is repeated 5 times or until the current pixel rises more than three pixels.
3. If the current pixel rises more than 3 pixels, then the routine is done. Otherwise, it repeats steps 1 and 2 to the left.

Once it was built and test using the segmentation toolbox, the hybrid drop fall segmentation heuristic was implemented as part of a larger check amount verification system called WinBank (See appendix for code). This incarnation of the algorithm was written in C++, but conceptwise is identical to the algorithm described above.

# Chapter 6: Implementation of Hybrid Algorithm within WinBank

WinBank is a system developed at MIT's Sloan School of management with the intent of automatically verifying check amounts. As with most handwriting recognition systems, WinBank consists of several preprocessing stages which include character segmentation, a neural net based recognition engine, and well as several post-processing stages. Since the hybrid drop fall algorithm developed in this thesis focuses only on segmentation and WinBank is a relatively large and complex system, we will look only at the way that WinBank currently handles only this process.

WinBank currently employs two waves of segmentation heuristics to connected components composed of two characters. The first of these is a min-max based procedure and the second is a particular type of hit and deflect heuristic. The hybrid drop fall algorithm is incorporated into the system as an optional third wave heuristic. This chapter will focus on how segmentation is currently done in WinBank, how the hybrid algorithm is implemented, and the benefits of having the hybrid algorithm in the system.

## 6.1 The min-max based algorithm

As part of a first attempt at segmenting a connected component into two numeric characters, the WinBank system uses a min-max algorithm. The particular version used in WinBank is min-max in its simplest form. The routine first scans the upper and lower profiles of the connected component within some deviation limit to both sides of the center of the image. It then stores the minimum and maximum values of the upper and lower contours, respectively. These values are then passed to a 'bridging' routine which attempts to join the two point with a path that traverses at most two black-to-white or white-to-black transitions. In constructing the path, the routine also attempts to avoid having the resulting separated components having irregular aspect ratios. If the procedure cannot find a path under these constraints, then it quits. In fact, it even quits if the minimum point on the upper contour is lower than the maximum point on the lower contour.

### 6.2 The hit and deflect algorithm

The WinBank system turns to a variation of the hit and deflect strategy when the min-max does not prove to be successful. This implementation is very similar to the standard drop fall algorithms discussed in previous chapters accept that it varies by one rule. In the drop fall algorithms, if the current pixel is 'blocked' from below, but not from the sides, it will move to one side or another. The hit and deflect algorithm implemented in Winbank will 'cut' into the pixels below it instead. For most cases, however, the algorithm produces similar results as a standard top-left drop fall. Just as with the min-max algorithm, this algorithm attempts to find a suitable path within dimensions constraints on the resulting segments.

### 6.3 Implementation of the hybrid drop fall algorithm

The hybrid drop fall algorithm described in Chapter 4 was implemented into the WinBank system as an optional third pass algorithm for improving segmentation (See appendix A.7 for code). Based on the arguments passed to the drop fall routine, it is capable of performing a top-left, top-right, bottom-left, or bottom-right drop fall. The routine will return both a segmentation path and the 'flatness' at the point where the routine 'cuts' into the connected component. In order to implement the hybrid routine, the routine should first be called with arguments to make it perform a top left drop fall. Then, if the routine returns a flatness below a given threshold, it should call the routine again with arguments which would make it a bottom-right drop fall. Figure 26 shows examples of when the hybrid drop fall algorithm as implemented in WinBank will outperform the hit and deflect and min-max algorithms.

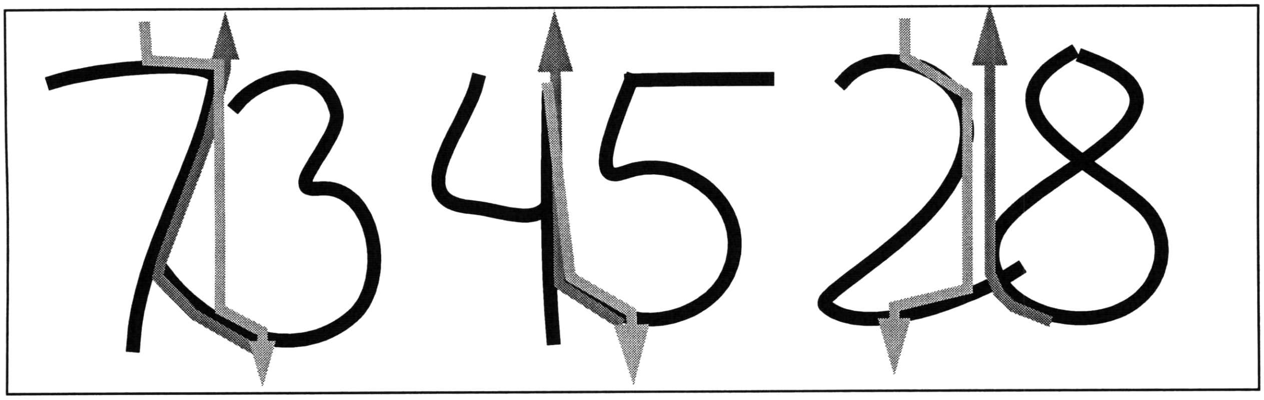

**Figure 26: The min-max and Hit and Deflect (light gray arrows) fail on every one of these examples. The min-max fail because in every case the minimum on the upper contour is lower than the maximum of the lower contour. The hybrid algorithm (dark gray arrows) is successful in every case shown**

# Chapter 7: Performance Results

## 7.1 Overall performance

Overall, the hybrid drop fall heuristic, the disjointed-five heuristic, and the functionality of the Segmentor show extremely promising results. As we will see, the heuristics described in this thesis provide for an extremely robust method for classifying and segmenting an image of a character string.

## 7.2 Effectiveness of disjointed five heuristic on extreme cases

The disjointed five heuristic presented in Chapter 4 provides a simple non-computationally intensive method by which to recognize and properly segment a '5' which has a disjointed top stroke. In this section, we will look at the cases for which the heuristic will fail and the cases for which it will fail.

Before going over specific examples, let us review the criterion by which the heuristic judges a connected component to be the disjointed top stroke of a '5'. The heuristic judges a connected component to be the disjointed top stroke of a '5' if the following criterion are met:

- The aspect ratio of the connected component is greater than 1.
- The height of the component is less than 1/3 the height of the previous character.
- The component is in the upper half of the image.

In order to evaluate the efficacy of these criterion, we need to look at 1) possible situations in which the top-stroke a disjointed '5' do not meet the above criterion and 2) possible situations in which another connected component could meet these criterion.

First we will look at cases in which a top-stroke of a disjointed '5' (TSDF) does not meet one of the above-defined criterion. If a TSDF does not meet the first of the criterion then it must have an aspect ratio less than or equal to '1'. A best case image of such a TSDF is depicted in Figure 27. As we can see, this '5' looks highly irregular and would most probably be misrecognized by a human reader.

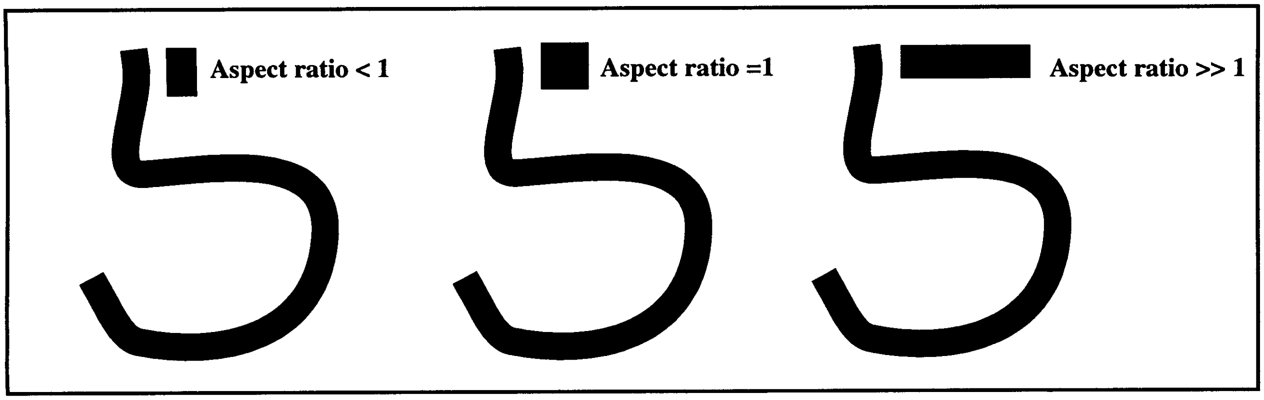


**Figure 27: The top stroke of a '5' must have an aspect ratio well over 1 in order for the '5' to be recognizable.**

If a TSDF does not satisfy the second criteria, then its height must be more than 1/3 the height of the previous criterion. The best case for this scenario (in which its height is exactly one-third the height of the main body of the '5') is depicted in Figure 28. Once again, we see that such a character would be hard for a human to recognize as a '5'. The likelihood of such a character being written using a pen is also highly unlikely.

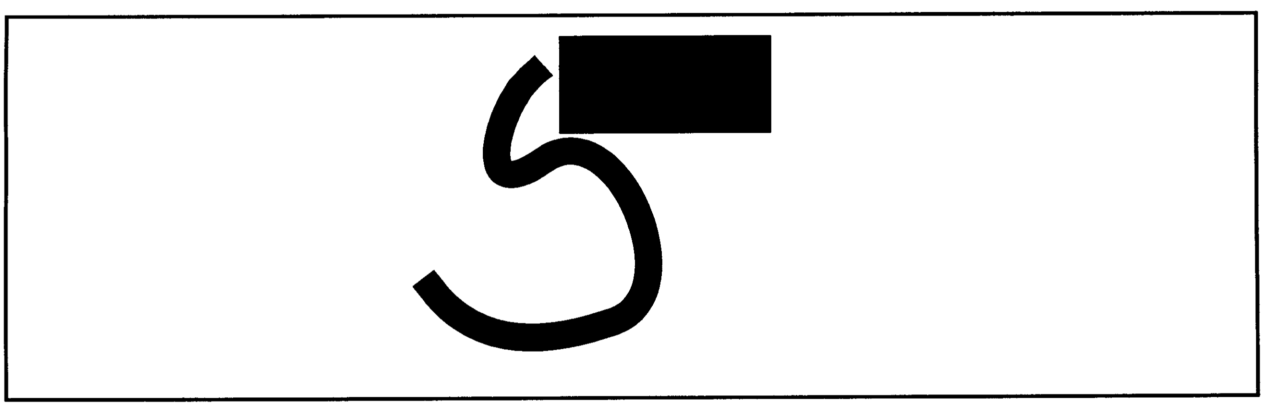

**Figure 28: Depiction of a case when the top stroke of a disjointed '5' has a height equal to one-third of the main body.**

If a TSDF does not satisfy the last criterion, then it cannot be in the upper half of the image. Care must be taken to define what exactly is meant by the upper half of the image since the third criterion will always fail if, for example, all of the characters are within the lower half of the image (which is entirely possible if the image of the numeral string has too much white space in its top half). Because of this, the top half of the image will be defined as the y-coordinates from the midpoint of the bounding box of the previous full sized character. A full sized character is any character with height greater than a certain threshold which can easily determined from the sizes of the various characters (since there is such a major height disparity between the full size characters such as the numbers and the other characters such as the decimal points and commas). Based on these definitions, Figure 29 depicts examples of TSDFs which are not in the top half of the image. Once again, all of these examples appear highly irregular and are not likely to be encountered in a real-world check processing scenario

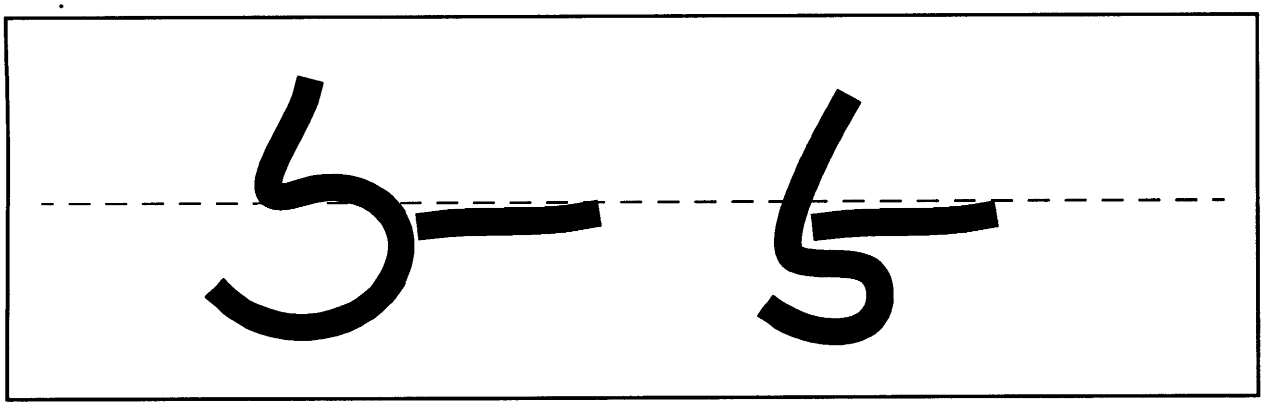

**Figure 29: Examples of TSDFs which are not contained in the top-half of the image**

As we can see, if there is a TSDF which does not satisfy all three of the above criterion, it is highly unlikely that it is legible even by a human reader. Because of this, if the correctness of the heuristic is measured by the ability of a human, this heuristic would have an extremely low false-negative rate.

All that is left in determining the effectiveness of the three criteria for identifying TSDFs is to look at possible false positive cases. In other words, we will consider other characters which could potentially satisfy the three criteria. It is clear that a full sized numeric character (even several connected ones) would not satisfy the criteria because they are too large. Decimals and commas could not be mistakes for a TSDF because they will always lie in the bottom half of the image. The only character other than a TSDF which could possibly meet the requirements is the divider of the numerator and denominator of the cent field written as a fraction. As mentioned in Chapter 5, this possibility can be avoided by adding a simple clause to the heuristic which tests whether the connected component any characters directly above it. If it does, then the character can be classified as a divider and not as a TSDF. As we can see, it is very difficult for another character to be misclassified as a TSDF based on the previously defined criteria.

### 7.3 Recognition of connected characters with heavily skewed aspect ratios

In Chapter 6, we saw that connected components composed of connected or overlapping characters are identified by their irregularly large aspect ratios. The question is how robust such a method of classification is. We will explore this question by, once again, looking at two possible scenarios where the heuristic could fail (i) connected components consisting of more than one character whose aspect ratio is less than 1; and (ii) single characters which could have aspect ratios greater than 1.

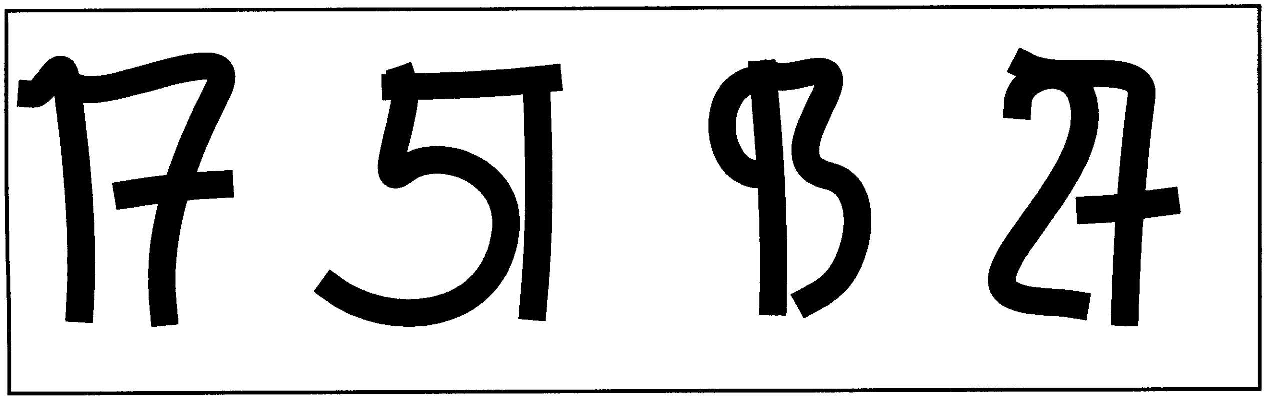

**Figure 30: Examples of two overlapping characters having a combined aspect ratio less than 1**

Figure 30 shows several examples of connected components consisting of more than one character which have aspect ratios less than 1. As we can see, these are special cases, but are not completely improbable. In fact, when the characters '1' or '7' touch another character, it is very possible that they would not significantly add to the total aspect ratio of the combined characters. Because of this, the aspect ratio-based method for classifying connected components may be useful, but it is not by any means have a zero false negative rate when it comes to declaring connected components as consisting of more than one touching or overlapping character.

Figure 31 shows examples of single characters written so that they have aspect ratios greater than one on their own. Basically, all of these characters are just 'plump' versions of how most people would write them. Despite the fact that they are written rather unusually, it would not be impossible to encounter handwriting like this. So once again we see that although the aspect-ratio approach may be good, it is not perfect.

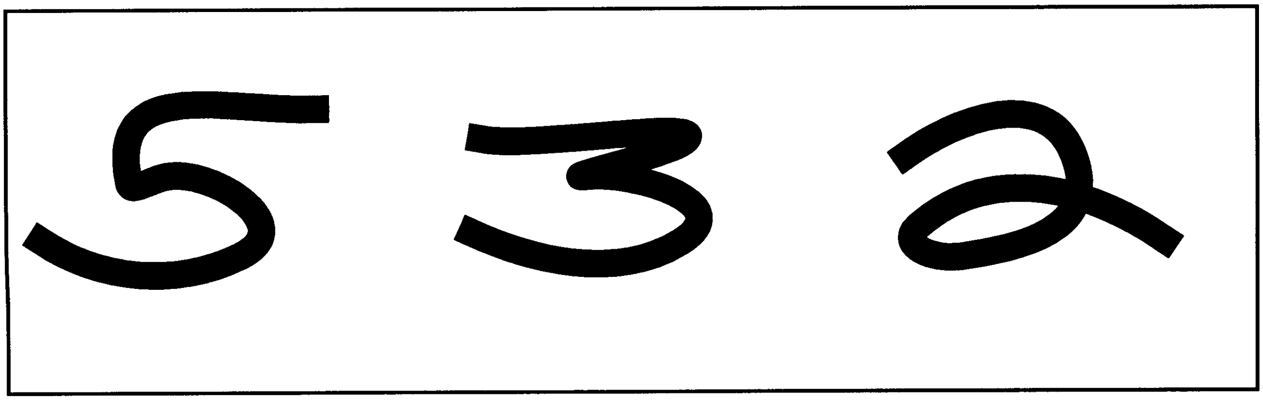

**Figure 31: Examples of single characters with aspect ratios greater than 1.**

As we can see. the aspect-ratio based heuristic for determining which connected components are composed of touching or overlapped character is not air tight. It is currently, however, the main way that this type of classification is done in most segmentation systems, mainly because of its speed and simplicity. Developing a more sophisticated version of this heuristic or making an entirely new, more robust heuristic for this purpose would do a lot in terms of improving current segmentation systems. This is discussed in further detail in the next chapter.

### 7.4 Performance of splitting heuristics

Now we will evaluate the central theme of this thesis, namely, the hybrid drop fall algorithm described in Chapter 4. As mentioned before, this algorithm attempts to merge the best aspects of the top-left and bottom-right drop fall algorithms. The algorithm is based on the central premise that a 'cut' made at a relatively flat minimum point is most likely an incorrect one (Figure 18). In order to properly evaluate this assumption, one needs to look at the possibility of 1) cases where a cut made at a relatively flat minima is correct and 2) cases when a cut made at a sharp minima is incorrect.

Cuts made at flat minima can only be correct if two relatively flat regions of the two different characters meet exactly at the cut point. Of course, this can only happen with two numbers which have flat regions, namely ‘3’, ‘5’, and ‘7’. ‘2’ is not included despite the fact that its bottom stroke is flat because not other character can touch the bottom stroke of a ‘2’ at a point which is flat at that character as well (i.e. no other characters have flat bottom strokes). Examples of two characters making contact at a flat point are shown in Figure 32. In these special cases, the characters have to line up exactly so as to not allow for any ‘sharp’ minima at their junction. Also, even though the hybrid drop fall may mistakenly judge the top-left-based cut to be at the wrong minima, it could still make the proper cut when it makes a bottom-right drop fall attempt

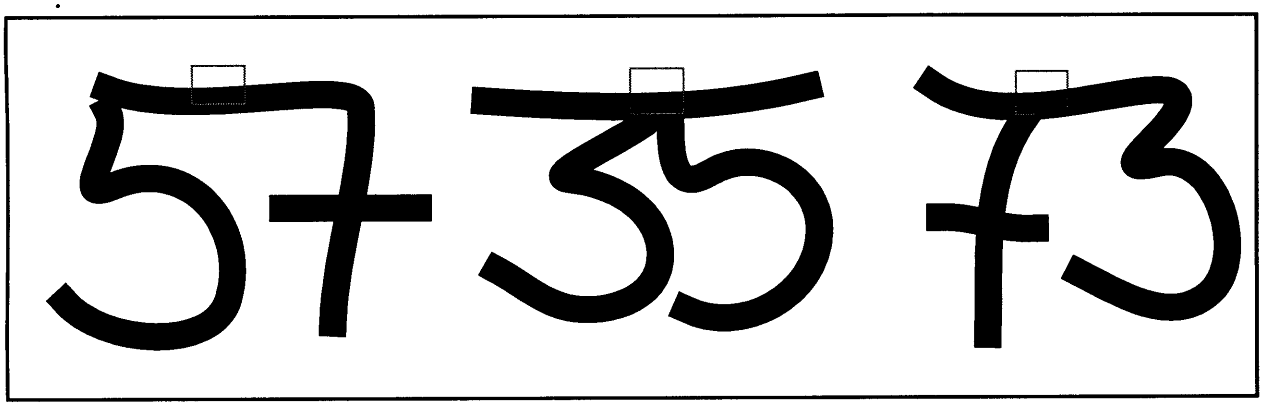

**Figure 32: Examples of two characters touching at a flat minimum point**

In order for a ‘sharp’ minima to be the incorrect point at which to cut into a connected component, it must occur within a character rather than at the junction of two characters. Only ‘2’, ‘4’, and ‘9’ can potentially have sharp minimum points on them as is shown in Figure 33. Even though the characters depicted could be encountered, in order for the drop fall algorithm to locate the minima under question in every case other than the ‘2’, it would have to start in the wrong place. In the case of the ‘2’, the example depicted is skewed quite a bit, but it does, however, show that under the right conditions the algorithm is not perfect.

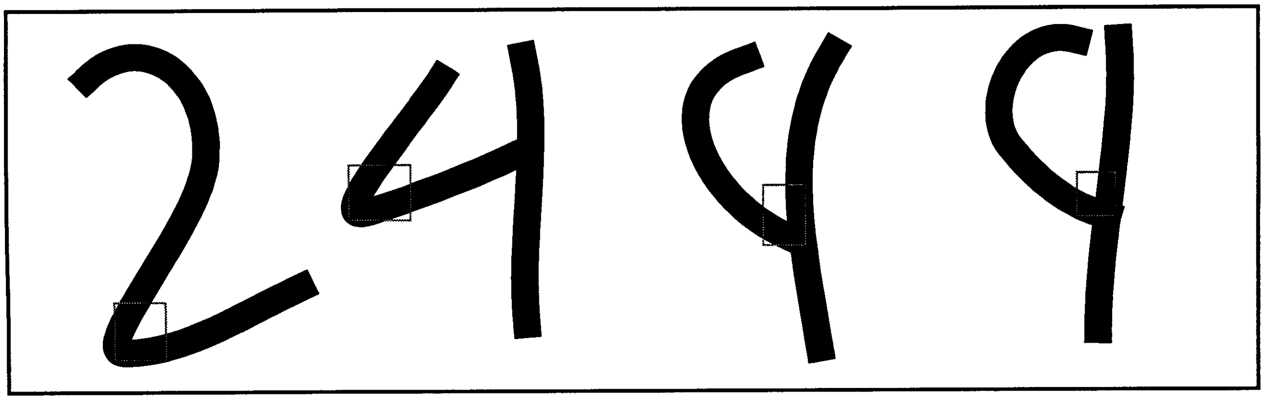

**Figure 33: Examples of numbers with potentially sharp internal cut points**

The above analysis only critiques the hybrid drop fall heuristic up until the first top-left is performed. The second bottom-right drop fall will not improve the results in the case when a sharp cut within a character is mistaken as the proper cut (mainly because it will not even be performed), but it could make up for any errors made in the first wave due to flat minima (Figure 32). In this case, the effectiveness of the overall algorithm reduces to the effectiveness of the traditional bottom-right drop fall outlined in Chapters 3 and 4. Since the algorithm reduces to the top-left or bottom-right drop fall only in the worst case, it is at least as good as the better of the two. Since the hybrid algorithm successfully segments many cases on which the traditional drop falls fail, it is clearly a significant improvement.

# Chapter 8: Conclusions

The algorithms and applications described in this thesis were made for the sole purpose of improving the performance of segmentation systems which has the broader impact of improving the quality of automatic handwriting recognition. Despite the fact that segmentation of a character string is a relatively simple concept, we have seen that it is an extremely difficult task and is far from being perfected.

Handwritten character recognition, and segmentation in particular, is a process that we humans take for granted. Very seldom do we think of the complex interactions between millions of neurons which have to take place in our brains in order to perform such tasks. Besides the obvious visual cues of the strokes which make up the character, a human is also able to use knowledge of known words and the context of the writing to further recognize written text. When a human reads a character string, the segmentation and recognition processes are performed simultaneously and in conjunction with each other. Taking these factors into consideration, it is clear that a 100% accurate recognizer will only be able to built using technology which can model the human mind. Since we are currently no where near that point, the best we can currently do is study the patterns and features of handwriting and attempt to develop segmentation and recognition heuristics based on our findings.

The segmentation process as described in this paper consisted of three steps: 1) Identification of connected components, 2) Classification of connected components, and, finally, 3) splitting touching characters. As we saw in Chapters 3 and 4, the first step is relatively straight forward to perform perfectly. Steps 2 and 3, however, are no where near perfected and could strongly benefit from further research.

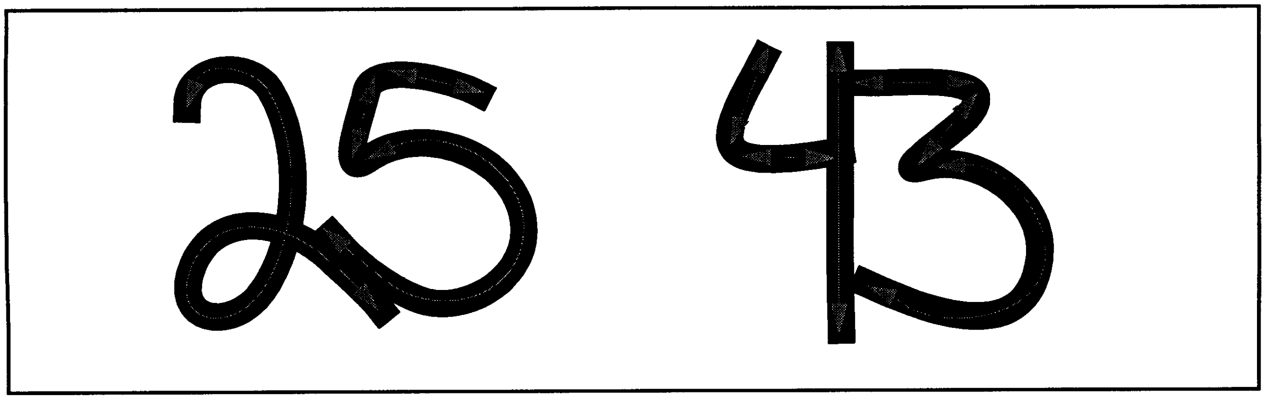

**Figure 34: Identifying the straight or flat strokes could give a clue as to how characters there are.**

In Chapter 5 we saw that the current component classification mechanism is extremely simple. In Chapter 7, it was shown that this aspect-ratio-based heuristic fails on many cases and, therefore, could definitely make use of further innovations. There are several potential add-ons to this heuristic which could potentially make it more robust and, therefore, should be the topic of future research in this area. First, information from the strokes which make up a connected component could potentially help decide how many characters make up the component (Figure 34). Also information of the number of minima and maxima which occur within a certain pixel range could be used as well (Figure 35). Finally, the best classifier of how many characters make up a connected component would probably be a properly trained neural network. This would most closely model how the human mind probably segments characters. The last case, however, has not been seriously considered as yet because would require a significant increase in complexity and computation power.

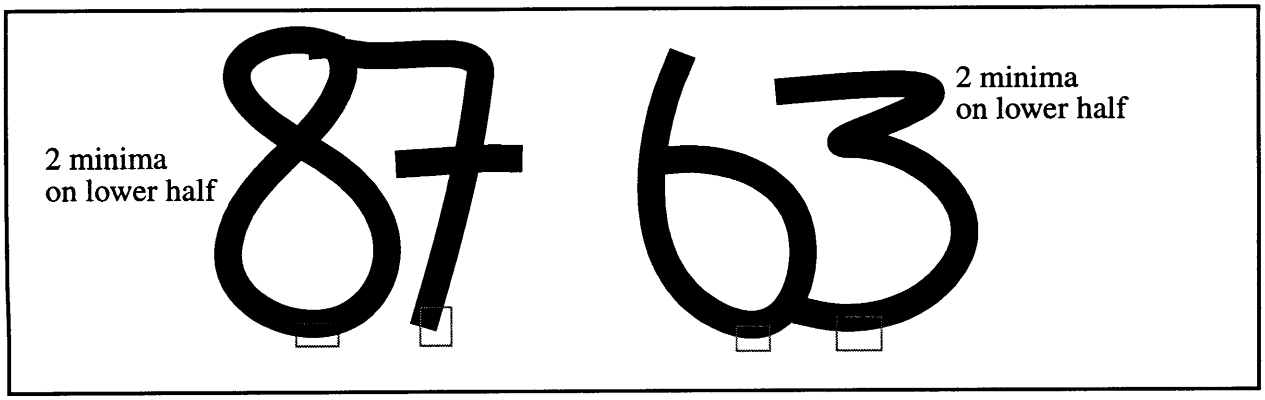


**Figure 35: Counting the number of minima on the lower half of the connected components could give insight as to how many characters the connected component is made of.**

The last step in the segmentation process, namely splitting touching characters, has been worked on by many researchers [1, 4, 5, 6, 7, 9, 10], but it too has not been perfected. As mentioned in the first several chapters, all of the current methods (min-max, hit-and-deflect, and drop fall) try to make use of knowledge of the point where the two characters may potentially touch in order to make a good 'splitting' decision. The hybrid drop fall algorithm presented in Chapter 3 was designed along these lines as well, although it has improved performance. In order to continue to improve the segmentation of connected characters, two distinct research paths can be taken: 1) continue to develop along the current lines of studying how characters may touch and make increasingly sophisticated heuristics based on this knowledge and 2) develop 'intelligent' algorithms capable of learning new ways to split touching characters. The first of the two directions is where most of the current research has going. The second direction, which could use learning systems such as neural nets is significantly more complicated, but is most probably the direction which the research will have to go in if the eventual goal is to attain performance comparable to that of a human.

## 9 References


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

[2] Agarwal, A.; Hussein, K.; Gupta, A.; Detection of courtesy amount block on bank checks. *Journal of Electronic Imaging.* **5**(2), 214-224 (April 1996).

[3] Nagendraprasad, M.; Wang, P.; Gupta, A.; Liu, A.; "A neural net based system for recognition of unconstrained handwritten numerals, presented at the Second International Conference on Automation, Robotics and Computer Vision, Singapore, 1992.

[4] Sparks, P.; Nagendraprasad, M.; Gupta, A.; "An algorithm for segmenting handwritten numeral strings," presented at the Second International Conference on Automation, Robotics, and Computer Vision, Singapore, 1992.

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

[6] Lu, Y.; Shridhar, M.; Character Segmentation in Handwriiten Words - An Overview, *Pattern recognition.* **29**(1) pp. 77 (1996)

[7] Shridar, M.; Badredlin; Recognition of Isolated and Simple Connected Handwritten Numerals, Pattern Recignition, **19**(1) pg.1, (1986)

[8] R. Fenrich and K. Krishnamoorthy, Segmenting diverse quality handwritten digit strings in near real-time, *Procedings of the 4th Advanced Technology Conf.* 523-537, (1990)

[9] N. W. Strathy, C. Y. Suen and A. Kryzak, Segmentation of handwritten digits using contour features, *Second Int. Conf. Document Anal. Recognition* 577-580 (October 1993)

[10] G. Congedo, G. Dimauro, S. Impedovo, and G. Pirlo, Segmentation of Numeric Strings, *Proc. of Third Int. Conf. on Document Analysis and Recognition*, Montreal, August 14-16, 1995, pp.1038-1041

[11] A. Senior and A. Robinson, An Off-Line Cursive Handwriting Recognition System, *IEEE transactions on pattern analysis and machine intelligence.* **20**(3), pp. 309 (1998)

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

# Appendix

## A.1 Code for Segmentor's graphical user interface

```
%Following line create the menus which allow the user to select an image file

image_opt=uimenu(gcf, 'Label', 'Image');

easy_opt=uimenu(image_opt, 'Label', 'Easy');

easy_image1=uimenu(easy_opt, 'Label', '12,856', 'CallBack', 'num_image=imread(''easy/
                    image1'', ''jpg''); imshow(num_image); segmentor;');

medium_opt=uimenu(image_opt, 'Label', 'Medium');

medium_image1=uimenu(medium_opt, 'Label', '35,675', 'CallBack',
                      'num_image=imread(''medium/image1'', ''jpg'');
                      imshow(num_image); segmentor;');

difficult_opt=uimenu(image_opt, 'Label', 'Difficult');

difficult_image1=uimenu(difficult_opt, 'Label', '95,732', 'CallBack', 'num_image=imread(''dif-
                      ficult/image1'', ''jpg''); imshow(num_image); segmentor;');

%This button separates the connected components from each other
%In the process, it tries to keep didjoited components of one character
%such as the upper stroke of a 5 together.

pbseparate= uicontrol(gcf,...
                    'style', 'push',...
                    'position', [10 100 200 25],...
                    'string', 'Identify Connected Components',...
                    'Callback', '[counter, component_tiles,
                    size_matrix]=step1(num_image);');

%The function returns all the tiles and it returns an index
%to the tiles that have to be further segmented.

pbrecognize_special_tiles= uicontrol(gcf,...
                              'style', 'push',...
                              'position', [10 70 200 25],...
                              'string', 'Classify Components',...
                              'Callback', 'special_tiles=step2(counter, component_tiles,
                              size_matrix);');
```

```
pbrecognize_dropfall_tiles= uicontrol(gcf,...
                                      'style', 'push',...
                                      'position', [10 40 200 25],...
                                      'string', 'Perform Drop Fall',...
                                      'Callback', '[counter2, component_tiles2,
                                       size_matrix2]=step3(counter, component_tiles, size_matrix,
                                       special_tiles, dir);');

pbreset= uicontrol(gcf,...
                   'style', 'push',...
                   'position', [10 10 200 25],...
                   'string', 'Reset',...
                   'Callback', '[counter, indices, component_tiles]=step2(counter,
                     component_tiles, size_matrix);');

rbtop_left = uicontrol(gcf, ...
                       'Units','points', ...
                       'Position',[380 80 90 25], ...
                       'Style','radiobutton', ...
                       'Value', 1, ...
                       'Tag','Radiobutton1', ...
                       'String', 'Top-Left', ...
                       'Callback' , [...
                       'set(rbtop_right, ''Value'',0);' ...
                       'set(rbbottom_left, ''Value'',0);'...
                       'set(rbbottom_right, ''Value'',0);'...
                       'dir=''top-left'';']);

dir='top-left';

rbtop_right = uicontrol(gcf, ...
                        'Units','points', ...
                        'Position',[380 55 90 25], ...
                        'Style','radiobutton', ...
                        'Tag','Radiobutton2', ...
                        'String', 'Top-Right', ...
                        'Callback' , [...
                        'set(rbtop_left, ''Value'',0);' ...
                        'set(rbbottom_left, ''Value'',0);'...
                        'set(rbbottom_right, ''Value'',0);'...
                        'dir=''top-right'';']);

rbbottom_left = uicontrol(gcf, ...
                          'Units','points', ...
                          'Position',[380 30 90 25], ...
                          'Style','radiobutton', ...
```

```
                    'Tag','Radiobutton3', ...
                    'String', 'Bottom-Left', ...
                    'Callback' , [...
                    'set(rbtop_right, ''Value'',0);' ...
                    'set(rbbottom_right, ''Value'',0);'...
                    'set(rbtop_left, ''Value'',0);'...
                    'dir=''bottom-left'';']);

rbbottom_right = uicontrol(gcf, ...
                    'Units','points', ...
                    'Position',[380 5 90 25], ...
                    'Style','radiobutton', ...
                    'Tag','Radiobutton4', ...
                    'String', 'Bottom-Right', ...
                    'Callback' , [...
                    'set(rbtop_right, ''Value'',0);' ...
                    'set(rbbottom_left, ''Value'',0);'...
                    'set(rbtop_left, ''Value'',0);'...
                    'dir=''bottom-right'';']);

%*********************************************************

label1 = uicontrol(gcf, ...
               'Units','points', ...
               'BackgroundColor',[0.701961 0.701961 0.701961], ...
               'Position',[180 18 35 10], ...
               'Style','text', ...
               'Tag','StaticText1', ...
               'String', 'Thrshld');

slider1 = uicontrol(gcf, ...
               'Units','points', ...
               'Position',[190 30 15 75], ...
               'Style','slider', ...
               'Tag','Slider1', ...
               'Min', 1, ...
               'Max', 255, ...
               'Value', 100, ...
               'Callback' , [...
               'set(V1n_current, ''String'', ', ...
               'num2str(get(slider1, ''Val'')))']);

V1n_current = uicontrol(gcf, ...
                    'Style', 'text', ...
                    'Pos', [215 10 35 10], ...
                    'String', num2str(get(slider1, 'Value')));
```

```
%*****************************************************

label2 = uicontrol(gcf, ...
                  'Units','points', ...
                  'BackgroundColor',[0.701961 0.701961 0.701961], ...
                  'Position',[220 18 35 10], ...
                  'Style','text', ...
                  'Tag','StaticText2', ...
                  'String', 'Aspect');

slider2 = uicontrol(gcf, ...
                  'Units','points', ...
                  'Position',[230 30 15 75], ...
                  'Style','slider', ...
                  'Tag','Slider2', ...
                  'Min', .25, ...
                  'Max', 4, ...
                  'Value', 2, ...
                  'Callback' , [...
                  'set(V2n_current, ''String'', ', ...
                  'num2str(get(slider2, ''Val'')))']);

V2n_current = uicontrol(gcf, ...
                        'Style', 'text', ...
                        'Pos', [262.5 10 35 10], ...
                        'String', num2str(get(slider2, 'Value')));

%*****************************************************

label3 = uicontrol(gcf, ...
                  'Units','points', ...
                  'BackgroundColor',[0.701961 0.701961 0.701961], ...
                  'Position',[260 18 35 10], ...
                  'Style','text', ...
                  'Tag','StaticText3', ...
                  'String', 'TravDist');

slider3 = uicontrol(gcf, ...
                  'Units','points', ...
                  'Position',[270 30 15 75], ...
                  'Style','slider', ...
                  'Tag','Slider3', ...
                  'Min', 1, ...
                  'Max', 6, ...
```

```
                'Value', 3, ...
                'Callback' , [...
                'set(V3n_current, ''String'', ', ...
                'num2str(get(slider3, ''Val'')))']);

V3n_current = uicontrol(gcf, ...
                    'Style', 'text', ...
                    'Pos', [310 10 35 10], ...
                    'String', num2str(get(slider3, 'Value')));

%*****************************************************

label4 = uicontrol(gcf, ...
                'Units','points', ...
                'BackgroundColor',[0.701961 0.701961 0.701961], ...
                'Position',[300 18 35 10], ...
                'Style','text', ...
                'Tag','StaticText4', ...
                'String', 'Slope');

slider4 = uicontrol(gcf, ...
                'Units','points', ...
                'Position',[310 30 15 75], ...
                'Style','slider', ...
                'Tag','Slider4', ...
                'Min', .1, ...
                'Max', 2, ...
                'Value', .5, ...
                'Callback' , [...
                'set(V4n_current, ''String'', ', ...
                'num2str(get(slider4, ''Val'')))']);

V4n_current = uicontrol(gcf, ...
                    'Style', 'text', ...
                    'Pos', [357.5 10 35 10], ...
                    'String', num2str(get(slider4, 'Value')));

%***********************************************************

label5 = uicontrol(gcf, ...
                'Units','points', ...
                'BackgroundColor',[0.701961 0.701961 0.701961], ...
                'Position',[340 18 35 10], ...
                'Style','text', ...
                'Tag','StaticText5', ...
                'String', 'DrpStrt');
```

```
slider5 = uicontrol(gcf, ...
                    'Units','points', ...
                    'Position',[350 30 15 75], ...
                    'Style','slider', ...
                    'Tag','Slider5', ...
                    'Min', 1, ...
                    'Max', 100, ...
                    'Value', 5, ...
                    'Callback' , [...
                    'set(V5n_current, ''String'', ', ...
                    'num2str(get(slider5, ''Val'')))']);

V5n_current = uicontrol(gcf, ...
                        'Style', 'text', ...
                        'Pos', [405 10 35 10], ...
                        'String', num2str(get(slider5, 'Value')));
```

### A.2 Connected Component Finder Routine

```
function [counter, image_tiles, size_matrix]=step1(img)
    %Step1 finds all of the connected components using a breadth first search
    [height, width]=size(img);
    image2=img;
    bound_box_list_entry=1;
    bound_box_list=[];
    image_tiles=[];
    display_tiles=[]; %This array is used to display the connected components nicely
    separator=zeros(height, 3); %All of the separator lines are for display purposes
    separator(:,1)=ones(height, 1)*255;
    separator(:,3)=separator(:,1);
    size_matrix=[];
    counter=0;
    x2=1;
    x1=1;
    for x=1:width
        for y=1:height
            if image2(y,x)<100 %IMPLEMENT THRESHOLD ADJUSTMENT and DYNAMIC
                THRESHOLD ADJUSTMENT
                old_x2=x2;
                old_x1=x1;
                old_width=x2-x1;
                [x1,y1,x2,y2,image2,cropped_component]=bfs(image2,y,x);

                %Is the height of the image less than half of the width
                %and, if so, is it less than 1/3rd the height of the
                %previous characters and if so is it in the top half of the image
                %***You should also check to see if the character has any other connected
```

```
            %***Components contained entirely above it. In this case, it is probably
            %***the divisor in the cent field of the character string.

            if (x2-x1)>(2*(y2-y1)) & (y2-y1)<size_matrix(counter,1) & y2<height/2 &
            x1<old_x2
                tiles_width=size(image_tiles, 2)
                start_x = tiles_width-old_width+x1-old_x1
                end_x = start_x+x2-x1
                x1
                if (end_x<tiles_width)
                    image_tiles(y1:y2,start_x:end_x)=...
                    image_tiles(y1:y2,start_x:end_x)|cropped_component(y1:y2,:);
                else
                    image_tiles(y1:y2,start_x:tiles_width)=...

            image_tiles(y1:y2,start_x:tiles_width)|cropped_component(y1:y2,1:tiles_width-
            start_x+1);
                end
            else
                image_tiles=[image_tiles  cropped_component ];
                counter=1+counter;
                size_matrix(counter,1)=y2-y1;
                size_matrix(counter,2)=x2-x1;
            end
        end
    end
end
counter_place=0;
for n=1:counter
    display_tiles=[ display_tiles separator image_tiles(:,
            counter_place+1:1+counter_place+size_matrix(n, 2))];
    counter_place=counter_place + size_matrix(n, 2) + 1;
end
subplot(3,1,1);
image(display_tiles);
subplot(3,1,2);
image(img);
```

### A.3 Breadth first search routine used in Segmentor

```
function [counter, image_tiles, size_matrix]=step1(img)
    %Step1 finds all of the connected components using a breadth first search
    [height, width]=size(img);
    image2=img;
    bound_box_list_entry=1;
```

```
bound_box_list=[];
image_tiles=[];
display_tiles=[]; %This array is used to display the connected components nicely
separator=zeros(height, 3); %All of the separator lines are for display purposes
separator(:,1)=ones(height, 1)*255;
separator(:,3)=separator(:,1);
size_matrix=[];
counter=0;
x2=1;
x1=1;
for x=1:width
    for y=1:height
        if image2(y,x)<100 %IMPLEMENT THRESHOLD ADJUSTMENT and DYNAMIC
        THRESHOLD ADJUSTMENT
            old_x2=x2;
            old_x1=x1;
            old_width=x2-x1;
            x1,y1,x2,y2,image2,cropped_component]=bfs(image2,y,x);

            %Is the height of the image less than half of the width
            %and, if so, is it less than 1/3rd the height of the
            %previous characters and if so is it in the top half of the image
            %***You should also check to see if the character has any other connected
            %***Components contained entirely above it. In this case, it is probably
            %***the divisor in the cent field of the character string.

            if (x2-x1)>(2*(y2-y1)) & (y2-y1)<size_matrix(counter,1) & y2<height/2 &
                x1<old_x2
                tiles_width=size(image_tiles, 2)
                start_x = tiles_width-old_width+x1-old_x1
                end_x = start_x+x2-x1
                x1
                if (end_x<tiles_width)
                    image_tiles(y1:y2,start_x:end_x)=...
                    image_tiles(y1:y2,start_x:end_x)|cropped_component(y1:y2,:);
                else
                    image_tiles(y1:y2,start_x:tiles_width)=...

                image_tiles(y1:y2,start_x:tiles_width)|cropped_component(y1:y2,1:tiles_widt
                h-start_x+1);
                end
            else
                image_tiles=[image_tiles  cropped_component ];
                counter=1+counter;
                size_matrix(counter,1)=y2-y1;
                size_matrix(counter,2)=x2-x1;
```

```
                end
            end
        end
    end
    counter_place=0;
    for n=1:counter
        display_tiles=[ display_tiles separator image_tiles(:,
                    counter_place+1:1+counter_place+size_matrix(n, 2))];
        counter_place=counter_place + size_matrix(n, 2) + 1;
    end
    subplot(3,1,1);
    image(display_tiles);
    subplot(3,1,2);
    image(img);
```

**A.4 Classifier Routine**

```
function special_tiles=step2(counter, component_tiles, size_matrix)
    special_tiles=[];
    for i=1:counter
        height=size_matrix(i,1)
        width=size_matrix(i,2)
        if width>height
            special_tiles=[special_tiles i];
        end
    end
```

**A.5 Connected Component Splitting Routine**

```
function [counter2, component_tiles2, size_matrix2] = step3(counter, component_tiles,
            size_matrix, special_tiles, dir)
    counter2=0;
    x_start=1;
    component_tiles2=[];
    size_matrix2=[];
    hght=size(component_tiles,1);
    count2=1;
    separator=zeros(hght, 3); %All of the separator lines are for display purposes
    separator(:,1)=ones(hght, 1)*255;
    separator(:,3)=separator(:,1);
    display_tiles=[]; %This array is used to display the connected components nicely
    for count=1:counter
        width=size_matrix(count, 2);
```

```
        x_end=x_start+width;

        tile=[];
        tile(1:hght,1:width+1)=component_tiles(1:hght,x_start:x_end);
        x_start=x_start+1+width;
        special_tile=0;
        for i=1:length(special_tiles)
            if special_tiles(i)==count
                special_tile=1;
            end
        end
        if special_tile==1
            [tile1, tile2]=dropfall(tile, dir);
            component_tiles2=[component_tiles2 tile1 tile2];
            display_tiles=[display_tiles separator tile1 separator tile2];
            size_matrix2(count2, 1)=size_matrix(count, 1);
            size_matrix2(count2, 2)=size(tile1, 2);
            count2=count2+1;
            size_matrix2(count2, 1)=size_matrix(count, 1);
            size_matrix2(count2, 2)=size(tile2, 2);
        else
            component_tiles2=[component_tiles2 tile];
            display_tiles=[display_tiles separator tile];
            size_matrix2(count2, 1:2)=size_matrix(count, 1:2);
        end

        count2=count2+1;
    end
    image(display_tiles);
```

**A.6 Drop Fall Routine**

```
function [tile1, tile2] = dropfall(tile, dir)
    width=size(tile,2);
    height=size(tile,1);

    dir
    cut=zeros(height,1);

    if strcmp(dir,'bottom-right')==1
        tile=fliplr(tile);
        tile=flipud(tile);
    elseif strcmp(dir, 'bottom-left')==1
        tile=flipud(tile);
    elseif strcmp(dir, 'top-right')==1
```

```
    tile=fliplr(tile);
end

%look for starting point to begin cutting at
%Stored under variable start_x
for row=1:height
    found_candidate=0;
    found_start=0;
    candidate_x=0;
    candidate_y=0;
    start_x=0;
    start_y=0;
    %Check the "right three fourths minus some" of each
    %row for the candidate
    for col=round(width/4):width-1
        pres_pix = tile(row,col);
        prev_pix = tile(row,col-1);
        next_pix = tile(row,col+1);
        if pres_pix==0 & next_pix>0
            found_candidate=1;
            candidate_x=col;
            candidate_y=row;
        end
        if found_candidate==1 & pres_pix==0 & prev_pix>0
            found_start=1;
            start_x=candidate_x+1
            start_y=candidate_y
        end
    end
    if found_start==1
        break;
    end
end



%Start the drop fall!

%Start defining the cut
cut(:,1)=start_x*ones(height,1);
row=start_y;
col=start_x;
path_tile=tile;

cut_point_x=0;
cur_point_y=0;
```

```
while (row<height)
    if path_tile(row+1,col)==255
        row=row+1;
    elseif path_tile(row+1,col+1)==255
        row=row+1;
        col=col+1;
    elseif path_tile(row+1,col-1)==255
        row=row+1;
        col=col-1;
    elseif path_tile(row, col+1)==255
        col=col+1;
    elseif path_tile(row, col-1)==255
        col=col-1;
    else
        if tile(row,col)==255
            cut_point_y=row; %point at which the cutting into the black begins
            cut_point_x=col;
        end
        row=row+1;
    end
    path_tile(row,col)=500; %This marks where the path has been
    cut(row,1)=col;
end


width1=max(cut)-1;
min_cut=min(cut);
width2=width-min_cut+1;
tile1=255*ones(height, width1);
tile2=255*ones(height, width2);

for i=1:height
    x_cut=cut(i);
    tile1(i, 1:x_cut-1)=tile(i, 1:x_cut-1);
    tile2(i, x_cut-min_cut+1: width2)=tile(i, x_cut:width);

end

if strcmp(dir,'bottom-right')==1
    tile1=fliplr(tile1);
    tile1=flipud(tile1);

    tile2=fliplr(tile2);
    tile2=flipud(tile2);
```

```
        %Switch tiles 1 & 2, they get reversed because of all the flipping
        dummy=tile1;
        tile1=tile2;
        tile2=dummy;
    elseif strcmp(dir, 'bottom-left')==1
        tile1=flipud(tile1);
        tile2=flipud(tile2);
    elseif strcmp(dir, 'top-right')==1
        tile1=fliplr(tile1);
        tile2=fliplr(tile2);
        dummy=tile1;
        tile1=tile2;
        tile2=dummy;
    end
```

**A.7 Drop Fall algorithm as implemented in WinBank**

```
/************************************************/
/************************************************/
/*              Drop Fall                       */
/*      with its supporting routines            */
/* This algorithm can perform 4 different drop  */
/* falling algorithms: top-left, top-right, */
/* bottom-left, and bottom-right.  Each of the*/
/* aforementioned directions refer to the */
/* direction from which to begin the 'drop fall */
/************************************************/
/************************************************/
#include <stdio.h>
#include <windowsx.h>
#include "segtyp.h"
#include "protos.h"
#include "typedef.h"
#include "head2.h"

#define TRUE  1
#define FALSE 0

#define TOPLEFT 1
#define TOPRIGHT 2
#define BOTTOMLEFT 3
#define BOTTOMRIGHT 4

#define STEEP 1
```

```
#define FLAT 0

/****************Functions in this file*****************/

//Performs a drop fall in the specified direction
tpSTEP DropFall(HWND hWnd, tSTRMAP component_map, tDIMEN component_dims,
                    int direction, int *concativity);
//Flips an image map over its vertical axis
void Fliplr(tSTRMAP *component_map, tDIMEN component_dims);

//Flips an image map over its horizontal axis
void Flipud(tSTRMAP *component_map, tDIMEN component_dims);

/******************************************************/

tpSTEP DropFall(HWND hWnd, tSTRMAP component_map, tDIMEN component_dims,
                    int direction, int *concativity)
{
 tSTRMAP *copy_map;
 tINDEX  row,col,endrow,(*Forw)(int);
 tCOORD  startpt,currpt,cutpt; // Variables to store the drop fall's start and
                 // current positions
 int height;
 char tmp;
 tpSTEP  psegpath,pnewstep,pcurrstep,pprevstep,ptmppath1,ptmppath2;

 //Flip the component map appropriately so as to perform the proper dropfall
 if (direction==BOTTOMRIGHT) {
        Flipud(&component_map, component_dims);
    Fliplr(&component_map, component_dims);
    }
 else if (direction==BOTTOMLEFT)
        Flipud(&component_map, component_dims);
 else if (direction==TOPRIGHT)
        Fliprl(&component_map, component_dims);

 //Don't flip if the direction is topleft because this is
 //what the function does naturally

 // Allocate the necessary space for copy_map
 copy_map = (tSTRMAP *)GlobalAllocPtr(GHND,sizeof(tSTRMAP));
 // copy component_map to copy_map
 // All modifications will be performed on copy_map
 for(col = component_dims.startc ;col <= component_dims.stopc; ++col)
    {
    for(row = component_dims.startr;row <= componet_dims.stopr; ++row)
```

```
        {
            (*copy_map)[row][col] = component_map[row][col];
            }
    }

// For now, we will only perform a top-left initiating drop-fall
// Later we can manipulate the image and path accordingly so
// as to perform the other 3 variations of drop-fall

//Find the start point for the drop fall by scanning the image
// row-by-row until a black border point with another black border
// point separated by white space to the right of it is detected


tINDEX potential_start_x=component_dims.startc;
tINDEX potential_start_y=componetn_dims.startr;
bool FOUND_START=FALSE;
bool FOUND_POTENTIAL;
for (row=component_dims.startr; row< component_dims.stopr; row++)
   {
   FOUND_POTENTIAL==FALSE;
   // Only search the second through third to last pixels in any given row
   // since these are the only ones that could potentially be the start
   // pixel
   for (col=component_dims.startc+1;col< component_dims.stopc-2; col++)
   {
       if component_map[row][col]==ZERO &&
       component_map[row][col+1]==ONE &&
        FOUND_POTENTIAL==FALSE
         {
       potential_start_x=col;
         potential_start_y=row;
           FOUND_POTENTIAL=TRUE
         }
       else if component_map[row][col]==ZERO &&
       component_map[row ][col-1]==ONE
           && FOUND_POTENTIAL==TRUE
       {
         FOUND_START=TRUE;
              startpt.col=potential_start_x;
              startpt.row=potential_start_y;
              break;
         }
     }
     if FOUND_START==TRUE
       break;
```

```
    }

 if FOUND_START==FALSE
   MessageBox(hWnd, "Could not find dropfall start point", "ERROR!",
  MB_OK | MB_ICONHAND);

// perform dropfall until end row
currpt.row = startpt.row;
currpt.col = startpt.col + 1;

// initialize the first step of the segmentation path
psegpath = StepCreate();
psegpath->point.row=component_dims.startr;
psegpath->point.row=currpt.col;
pprevstep=psegpath;
// This for loop constructs a vertical path from the top of the
// image to the start point
for (row=component_dims.startr+1; row<=currpt.row; row++)
  {
      pnewstep = StepCreate();
      pnewstep->point.row = row;
      pnewstep->point.col = currpt.col;
      pnewstep->next = NULL;
      pprevstep->next=pnewstep;
      prevstep=pnewstep;
  }

//Perform the actual drop fall
while(currpt.row <component_dims.stopr)
  {
  (*copy_map)[currpt.row][currpt.col]=SEEN
  pnewstep = StepCreate();

  if component_map[currpt.row+1][currpt.col]==ONE  &&
    (*copy_map)[currpt.row+1][currpt.col]!=SEEN
  {
    currpt.row++;
    }
  else if component_map[currpt.row+1][currpt.col+1]==ONE  &&
      (*copy_map)[currpt.row+1][currpt.col+1]!=SEEN
          {
          currpt.row++;
          currpt.col++;
          }
  else if component_map[currpt.row+1][currpt.col-1]==ONE  &&
      (*copy_map)[currpt.row+1][currpt.col-1]!=SEEN
```

```
            {
            currpt.row++;
            currpt.col--;
            }
        else if component_map[currpt.row][currpt.col+1]==ONE  &&
        (*copy_map)[currpt.row][currpt.col+1]!=SEEN
            {
            currpt.col++;
            }
  else if component_map[currpt.row][currpt.col-1]==ONE  &&
        (*copy_map)[currpt.row][currpt.col-1]!=SEEN
            {
    if component_map[currpt.row][currpt.col]==ONE
        {
        cutpt.row=currpt.row;
          cutpt.col=currpt.col;
        }
            currpt.col--;
            }
  else
  {
    currpt.row++;
    }
  pnewstep->point.row=currpt.row;
  pnewstep->point.col=currpt.col;
  pnewstep->next=NULL;
  pprevstep->next=pnewstep;
  pprevstep=pnewstep;

}

//Test for concativity

//First to the left...
height=0;
row=cutpt.row;
col=cutpt.col+2;

while(component_map[row][col]==ZERO && height<6)
{
  height++;
  row--;
}
if height<3
  (*concativity)=FLAT;
else
```

```
{
  //The curve is not flat to the right of the cut point
  //So check to the left...
  height=0;
  row=cutpt.row;
  col=cutpt.col-2;
  while(component_map[row][col]==ZERO && height<6)
  {
      height++;
  row--;
  }
  if height<3
      (*concativity)=FLAT;
  else
  (*concativity)=STEEP;
   }

  //Unflip the component map appropriately so as to perform the proper dropfall
  if (direction==BOTTOMRIGHT) {
      Flipud(&component_map, component_dims);
      Fliplr(&component_map, component_dims);
      }
  else if (direction==BOTTOMLEFT)
      Flipud(&component_map, component_dims);
  else if (direction==TOPRIGHT)
      Fliplr(&component_map, component_dims);

  //Adjust the path based on which type of drop fall was performed
  pnewstep=psegpath;
  while(pnewstep!=NULL)
      {
          if (direction==BOTTOMRIGHT) {
                  pnewstep->point.row=component_dims.stopr-pnewstep->point.row;
                  pnewstep->point.col=component_dims.stopc-pnewstep->point.col;
              }
          else if (direction==BOTTOMLEFT)
                  pnewstep->point.row=component_dims.stopr-pnewstep->point.row;
          else if (direction==TOPRIGHT)
                   pnewstep->point.col=component_dims.stopc-pnewstep->point.col;

           pnewstep=pnewstep->next;
      }

  GlobalFreePtr(copy_map);
  return(psegpath);
}
```

```
/***********************************************/
void Fliplr(tSTRMAP *component_map, tDIMEN component_dims)
{
 tSTRMAP *copy_map;
 tINDEX  row,col;

 // Allocate the necessary space for copy_map
 copy_map = (tSTRMAP *)GlobalAllocPtr(GHND,sizeof(tSTRMAP));
 // copy component_map to copy_map
 // the copying is dependent on
 for(col = component_dims.startc ;col <= component_dims.stopc; ++col)
    {
    for(row = component_dims.startr;row <= componet_dims.stopr; ++row)
        {
           (*copy_map)[row][col] = (*component_map)[row][col];
           }
    }
 for(col = component_dims.startc ;col <= component_dims.stopc; ++col)
    {
    for(row = component_dims.startr;row <= componet_dims.stopr; ++row)
        {
           (*component_map)[row][col] =
        (*copy_map)[row][component_dims.stopc-col];
           }
    }
    GlobalFreePtr(copy_map);
}

void Flipud(tSTRMAP *component_map, tDIMEN component_dims)
{
 tSTRMAP *copy_map;
 tINDEX  row,col;

 // Allocate the necessary space for copy_map
 copy_map = (tSTRMAP *)GlobalAllocPtr(GHND,sizeof(tSTRMAP));
 // copy component_map to copy_map
 // the copying is dependent on
 for(col = component_dims.startc ;col <= component_dims.stopc; ++col)
    {
    for(row = component_dims.startr;row <= componet_dims.stopr; ++row)
        {
           (*copy_map)[row][col] = (*component_map)[row][col];
           }
    }
```

```
 for(col = component_dims.startc ;col <= component_dims.stopc; ++col)
    {
    for(row = component_dims.startr;row <= componet_dims.stopr; ++row)
         {
              (*component_map)[row][col] =
         (*copy_map)[component_dims.stopr-row][col];
              }
    }
    GlobalFreePtr(copy_map);
}
```