Optical character recognition for checkbox detection

John Michael Istle
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/1724

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
OPTICAL CHARACTER RECOGNITION
FOR CHECKBOX DETECTION

by

John Michael Istle

Bachelor of Science in Computer Science
University of Nevada, Las Vegas
2003

Bachelor of Science in Mathematics
University of Nevada, Las Vegas
2003

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
College of Engineering

Graduate College
University of Nevada, Las Vegas
December 2004

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The Thesis prepared by

John Michael Istile

Entitled

"Optical Character Recognition for Checkbox Detection"

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Examination Committee Co-Chair

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

1037-53

ii
ABSTRACT

Optical Character Recognition for Checkbox Detection

by

John M. Istle

Dr. Evangelos Yfantis, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

Optical character recognition is the branch in computer science that involves reading text from paper and translating the images into a format that computers can manipulate. There are a lot of algorithms for finding letters and numbers, however checkboxes are often overlooked and very difficult to detect. To locate and determine if checkboxes are checked or unchecked is a very useful tool to use on forms. It is difficult to detect since there are so many ways a person can mark a checkbox. This thesis will describe a new algorithm for detecting checkboxes.

Before checkboxes can be searched, certain preprocessing algorithms need to be performed on the form. The preprocessing steps are used to ensure that the width of the pixels that inscribe characters are one pixel. Not all checkmarks are drawn inside the box. Once a box is found, the coordinates are saved for further analysis.
TABLE OF CONTENTS

ABSTRACT ......................................................................................................... iii

LIST OF FIGURES ............................................................................................. v

ACKNOWLEDGMENTS .................................................................................... vi

CHAPTER 1  INTRODUCTION .......................................................................... 1

CHAPTER 2  PREPROCESSING ......................................................................... 6
  2.1 Color to Gray Scale .............................................................................. 9
  2.2 Character Separation ......................................................................... 10
  2.3 Filling Gaps ......................................................................................... 12
  2.4 Thinning ............................................................................................... 13
  2.5 Endpoint Erosion .............................................................................. 20

CHAPTER 3  CHECKBOX RECOGNITION ...................................................... 25
  3.1 Contour Following ............................................................................. 26
  3.2 Roundness .......................................................................................... 32
  3.3 Algorithm Outline .............................................................................. 34

CHAPTER 4  OPTICAL MARK RECOGNITION ............................................. 38

CHAPTER 5  LOCATION .................................................................................. 41

CHAPTER 6  CONCLUSION .......................................................................... 52

APPENDIX  SOURCE CODE ........................................................................... 55

BIBLIOGRAPHY ................................................................................................ 64

VITA .................................................................................................................... 66
LIST OF FIGURES

Figure 1.0.1. Overview of the Checkbox Detection System ................... 3
Figure 1.0.2. A breakdown on some of the procedures. ....................... 4
Figure 2.0.2. An enhancement filter ............................................. 7
Figure 2.0.3. Samples of Checkboxes from several forms .................... 8
Figure 2.2.1. Four points that encapsulates itself within a border ....... 11
Figure 2.5.1. How endpoint erosion works on the checkmarks ............. 21
Figure 2.5.2. A run through the erosion process .............................. 22
Figure 2.5.3. The 3x3 window for endpoint erosion ............................ 23
Figure 3.0.1. Sample of an empty checkbox of poor quality ............. 26
Figure 3.1.1. The starting pixel is located at the top left ................... 27
Figure 3.1.2. Order of the search pattern for the contour following ..... 28
Figure 3.2.1. The perimeters of a circle and a square ....................... 33
Figure 3.3.1. Search is moving to the right ..................................... 34
Figure 3.3.2. Search is moving down .............................................. 35
Figure 3.3.3. Search is moving left ............................................... 35
Figure 3.3.4. Search is moving up ............................................... 36
Figure 3.3.5. Feature extraction & continuing to contour analysis ....... 37
Figure 4.0.1. Scan lines to look for a check .................................... 38
Figure 4.0.2. This shows the range of the scan lines ......................... 39
Figure 5.0.1. Actual sample output of checkbox detection algorithm ... 42
Figure 5.0.2. The output after the program finds the checkboxes ....... 43
Figure 5.0.3. The coordinate plain of the document ............................ 44
Figure 5.0.5. The layout of some checkboxes on a form ..................... 48
Figure 5.0.6. The checkboxes shown in a sorted list .......................... 49
Figure 5.0.7. These two checkboxes are on the same line .................... 50
Figure 5.0.8. Breakdown on how the checkboxes are grouped ............... 51
ACKNOWLEDGMENTS

I would like to thank Dr. Evangelos Yfantis who has been my mentor and one of the greatest teachers a student could have. I would also like to thank the rest of my advisory committee for getting me over this hurdle in my life, Dr. Regentova, Dr. Cooper, and Dr. Bein.

I would also like to send my thanks to the United States Department of Energy for their research grants who have funded this research. Also thanks to the UNLV Computer Graphics and Image Processing Laboratory for giving my first chance at real programming. I am very grateful of some of my friends and colleagues that I have worked with, Leonel Acoba, Jin Choe, Ramzi El-Khater, Ray Chavez, Oksana Kubushyna, Maria Teruel, and Christopher Visser.

I would like to thank my mom and dad for their encouragement through the years. I also would like to thank Pamela Mandelbaum for her support and for being there for me.
CHAPTER 1

INTRODUCTION

The ability to transfer a paper document to an electronic format is a growing need around the world. The University of Nevada, Las Vegas Computer Graphics and Image Processing Laboratory, part of the Center for Cybermedia Research under the School of Computer Science, has taken this challenge upon itself. The research was funded by the United States Department of Energy under the title, "Medical Records Knowledge Discovery and Information Management for Radiation Workers." One of the tasks of this project is to convert old medical record documents into electronic format. The fastest way to accomplish such a task on a large scale is to have an Optical Character Recognition (OCR) system, along with a handwriting recognition system, perform the conversion.

Most medical forms accomplish data collection by having patients or medical staff answer interrogational questions by entering a handwritten mark inside the boundaries of a check box. An algorithm had to be developed that could be utilized within a software application, to
recognize the condition of each checkbox it came across. More specifically, was there an intentional mark within the area of the box? In addition, the algorithm had to recognize a wide range of forms of varying quality. Due to the natural process of human handling, filing, storage, and time, not all forms were in good shape.

Therefore, the goal of this project is to process any type of form without prior knowledge of the characteristics or properties of said form. Our focus is to have a box detection algorithm that will detect boxes anywhere on the form without knowing the size, location or quantity of checkboxes.

The history of OCR research is relatively old in the area of pattern recognition [1]. The checkbox detection algorithm is an optical character recognition algorithm that can examine each character for a checkbox.

Optical mark recognition (OMR) is the process of taking scanned documents and reading predefined positions and records where marks are made on the form. Looking for the checkmark is the OMR part of the algorithm.
There are three major components for detecting checkboxes. The first one is the preprocessing. Preprocessing will prepare the document's characters to meet certain guidelines for the checkbox detection algorithm. The checkbox detection algorithm will determine if a character is a checkbox or not a checkbox. Lastly, if a checkbox is found, it will determine if some form of a checkmark is also present.
Figure 1.0.2. A breakdown on some of the procedures.

The first step is the preprocessing methods such as thinning and erosion. These methods make it easier to find boxes by reducing the number of pixels. The algorithm used to detect boxes is a right turn algorithm for detecting four corners. This will be the main factor in determining if a character is a checkbox or some other character since boxes have four corners.
Once a character passes the corner detection algorithm, it goes through a procedure that checks the perimeter to make sure that a box has been found, rather than the letter “O” which sometimes has four corners depending on the scan quality and the font.

The step-by-step procedure for detecting boxes will facilitate the conversion of the form to black and white, thin the pixels, perform erosion on the check marks, and locate characters that look like boxes. Once a checkbox has been located, the next step is to determine if it is checked or unchecked.
CHAPTER 2

PREPROCESSING

There are several obstacles to deal with before detecting boxes. The form has to be prepared and set up as specified in the checkbox detection algorithm.

There are smoothing and enhancing filters available to improve the quality of the document [2]. These filters are not required by the checkbox detection algorithm but will not hinder its affect and are likely to improve the recognition process.

![Smoothing low pass filter](image)

Figure 2.0.1. Smoothing filter.
There are filters that are greater than 3x3 windows available [3][4].

There are three main requirements that a form must meet before the checkbox detection algorithm is performed. For the first requirement, the form has to be in black and white where the black pixels represent typed to handwritten markings and the white is the background.

The second requirement is that the black pixels be no more than one pixel in width. It is easier to search characters when they only have a thickness of one pixel. The main search routine in the checkbox detection algorithm is a contour following method that runs through the characters. If the characters have a thickness greater than one pixel, this could greatly increase the probability of invalid results. There has been a lot of research done in contour following [5]. The one used in the
checkbox routine serves as two purposes, one as a pixel counter and the other as a corner locator.

The third requirement that the form has to go through is an erosion algorithm. This routine is specifically for checkboxes that are marked. Marked checkboxes often have handwritten marks that go beyond the boundary of a checkbox. It is difficult to run the contour following algorithm when there is a checkmark that runs beyond the border of the checkbox.

Figure 2.0.3. Samples of Checkboxes from several forms.
The erosion algorithm will remove the handwritten marks that go beyond the checkbox. Later on this thesis will explain that the preprocessing steps do not have to be done on the entire form, but just the individual characters right before they go through the checkbox detection algorithm.

2.1 Color to Gray Scale

It is possible that the form is scanned in color, gray scale, or black and white. To get around this, the form is converted to black and white. We do this because it is easier to detect edges when there is a definitive difference in finding an edge when comparing a white pixel to a black pixel. Also, it may not be known whether the form was scanned in gray scale or as a color image. Keep in mind, written text and typed text will be black and the background will be white. Every pixel on the form and determine if it should be converted to a black pixel or a white pixel. This can be done by measuring the red, green, and blue (RGB) values to YIQ where \( Y \) is the luminance and \( I \) and \( Q \) are the chrominance [6]. If the RGB values are high indicating a light background, then changes are made to the RGB values to equal white. Otherwise the pixel is dark and therefore the RGB values will be set to equal black. The matrix used in the transformation is given by the following [7]:

\[
\begin{pmatrix}
0.2989 & 0.5870 & 0.1140 \\
0.5960 & -0.2745 & -0.3214 \\
0.2115 & -0.5294 & 0.3112
\end{pmatrix}
\]
Figure 2.1.1 The RGB to YIQ transformation.

\[
\begin{bmatrix}
Y \\
I \\
Q
\end{bmatrix}
= 
\begin{bmatrix}
0.2989 & 0.5866 & 0.1144 \\
0.5959 & -0.2741 & -0.3218 \\
0.2113 & -0.5227 & 0.3113
\end{bmatrix}
\begin{bmatrix}
R \\
G \\
B
\end{bmatrix}
\]

The gray scale intensity can be determine by \( L = 0.299R + 0.587G + 0.114B \), where \( L \) is the luminance which is also known as the \( Y \) component in YIQ [8]. This will give a range of gray scale values from 0-255, where the smaller numbers represent dark pixels and the large values represent lighter pixels. After the gray scale conversion, the image gets converted to binary, black and white. By using a cut off value from the gray scale, all intensities below the cut off value are black and all the values above the cut off value are white.

2.2 Character Separation

Before getting to some of the preprocessing methods used on characters, the characters need to be identified and separated from the rest of the document. It would be a waste of processing time to apply the thinning and erosion algorithm on the entire document, even though it
would not have any effect on some the white space. The thinning and erosion do a multi-pass through the document until it is no longer needed. Since most of the documents consist of white space, separating the characters is necessary.

![Diagram of a number 4 with arrows pointing to its top, left, right, and bottom]

Figure 2.2.1. Four points that encapsulates itself with in a border.

When characters are touching the system must deal with it because it could affect the performance of interpreting characters [9]. There are a variety of techniques to separate characters [10].

At this point, the document has a binary representation since it was converted to black and white. The processes of separating the characters
will exam each pixel and do a Depth First Search on the black pixels. The search is looking for a boundary to mark off a character.

The boundary around the character is a rectangular box that encapsulates the pixels by finding the extremes: the extreme left, extreme right, extreme top, and extreme bottom. When the boundary is found, the character is ready for the next step in preprocessing.

2.3 Filling Gaps

Once the image is in black and white, the next step is to thin the pixels so that the edges of the characters have a width of one pixel. In some cases, there is a small chance that there are broken edges or gaps in the characters. This will be a problem if there is a broken edge in a checkbox because the final step in preprocessing is erosion, which removes all endpoints. To avoid this problem, a filling algorithm is applied.

By running multiple passes of a fattening or coating method, the edges of the characters increase in size [11]. When the edges increase, the gaps get smaller and merge together. Combining the morphological operations of dilation and erosion eliminates gaps in characters [12]. The dilation process will make the edges of the characters larger to fill in the gaps. After the dilation process, an erosion algorithm will shrink the width of the character down to one pixel in width [13]. The combination
of these two operations will work well with one of the requirements for the checkbox detection algorithm of having the image thinned down to one pixel in width and without having gaps.

2.4 Thinning

Some forms may use thick, large fonts when creating boxes. By running a two-pass thinning algorithm on the form, all thick edges are shrunk down to one pixel in width and at the same time maintaining the proper shape of the character [14]. There are many different varieties of thinning algorithms that keep a more accurate shape of the characters [15]. Preprocessing is an important step in OCR. It is the beginning process to prepare the form before running this algorithm [16].

The width of each character may be larger than one pixel; the thinning algorithm is used to shrink it down to one pixel in width. A thinning algorithm is defined as an algorithm where the output is an image where the lines are one pixel in width. Common thinning algorithms will have several runs or passes where each pass will delete a pixel if it is removable. A pixel is removable if it does not break a connection, is not an endpoint, and, is a black pixel. The algorithms only consider binary images where the pixel can either be black or white. This method is a self-stabilizing parallel thinning algorithm that uses a
3x3 window of neighbors. Using a window size of 3x3 is common among parallel thinning algorithms [17][18][19].

For the image to be stable, the image must not have any removable pixels. A non-stable image will be one that has removable pixels. For this algorithm to be parallel, think of each pixel as a process. Each pixel will have a status: unknown, stable, waiting, or delete. When a pixel is unknown, it will have to run a routine to check its neighbors to determine its status. An image has all of its pixels unknown before this algorithm runs. When a pixel is stable, it is either white or not removable. A pixel is waiting if at least one of its neighbors is unknown. A pixel's status is set to delete if none of its neighbors are unknown, and if its current status is unknown. The purpose of these statuses is to ensure that if two pixels are deleted that a connection will not be broken.

There may also be a case where the current pixel is waiting and all of its neighbor's are either waiting or stable. This is a possible deadlock state. In order to get around this, a rank is given to the pixels to determine which pixel would be set to delete. This will insure mutual exclusion so that only one pixel will be deleted.

2.4.1 Parallel Thinning States

The following will set the states for the current pixel's status:

UNKNOWN:
If this pixel is white or Black non-removable,
    Then status := STABLE
If this pixel is black and at least one neighbor is unknown
    Then status := WAITING
Else this pixel is removable
    Then status := DELETE
WAITING:
    If all of this pixel's neighbors are either WAITING or STABLE and its rank is higher than the other WAITING pixels (at least one neighbor is waiting)
        Then status := DELETE
    If all of this pixel's neighbors are STABLE (and none are waiting)
        Then status := DELETE
DELETE:
    Delete this pixel and set its status := STABLE
    All of its neighbors' statuses := UNKNOWN
    The image will stabilize when all of the statuses are set to STABLE.

2.4.2 Parallel Thinning Rules

The unknown state has three possible moves. It first checks to see if it is a white pixel. If it is, then it is stable and will never have to make
another move again. If it is black and not removable, its status is also set to stable.

The rules for being removable will now be addressed. If it is a removable pixel, its neighbors' statuses are checked. If at least one of its neighbor's statuses is unknown, then the current pixel is set to waiting because it needs to wait on the unknown pixels before doing anything. If this pixel is removable and none of its neighbors have an unknown status, then this means that its neighbors' statuses are either waiting or stable. It will not have a neighbor whose status is set to delete because the current pixel is set to unknown.

The rules for deleting require that none of its neighbors are unknown. Therefore this pixel whose status is currently set as unknown will be set as delete. When a pixel's status is set to delete, that pixel is deleted and all of its neighbors' statuses are changed to unknown. They are changed to unknown so that they can re-evaluate themselves in case they were waiting, or were black pixels that were stable and could now be removable.

A pixel's status is waiting because it is waiting on its neighbors that are unknown. There is a possible deadlock situation that could occur since this algorithm is done in parallel. That situation is when the pixel is waiting and all of its neighbors' statuses are waiting or stable. In this situation, a decision has to be made as to which pixel that is still waiting gets to be deleted. The pixel that gets deleted is the one with the lowest
rank (or lowest ID). Each pixel is assigned a rank or ID based on its location on the image. The rank can be calculated as follows:

\[ r = n \times y + x, \text{ where} \]

- \( r \) = rank,
- \( n \) = number of columns in the image,
- \( y \) = row number, and
- \( x \) = column number.

There could only be one pixel with the lowest rank that is waiting, that will be the pixel whose status will be set to delete.

Up to this point, the only thing left that needs to be defined is a "removable" pixel. A pixel is removable if it is black and satisfies the following:

\[ P[x][y] \text{ is the current, where} \]

- \( x \) = the column, and
- \( y \) = the row.

(3 neighbors are black)

1. \( P[x-1][y-1] \land P[x][y-1] \land P[x+1][y-1] \) are black all others are white
2. \( P[x-1][y+1] \land P[x][y+1] \land P[x+1][y+1] \) are black all others are white
3. \( P[x-1][y-1] \land P[x-1][y] \land P[x-1][y+1] \) are black all others are white
4. \( P[x+1][y-1] \land P[x+1][y] \land P[x+1][y+1] \) are black all others are white

(4 neighbors are black)

1. \( P[x+1][y-1] \land P[x+1][y] \land P[x+1][y+1] \land P[x][y-1] \) are black all others are white
2. \( P[x-1][y-1] \land P[x][y-1] \land P[x+1][y-1] \land P[x+1][y] \) are black all others are white
3. \( P[x+1][y-1] \land P[x+1][y] \land P[x+1][y+1] \land P[x][y+1] \) are black all others are white
4. \( P[x-1][y+1] \land P[x][y+1] \land P[x+1][y+1] \land P[x+1][y] \) are black all others are white
5. \( P[x-1][y+1] \land P[x][y+1] \land P[x+1][y+1] \land P[x-1][y] \) are black all others are white
6. \( P[x-1][y-1] \land P[x-1][y] \land P[x-1][y+1] \land P[x][y+1] \) are black all others are white
7. \( P[x-1][y-1] \land P[x][y-1] \land P[x-1][y+1] \land P[x][y-1] \) are black all others are white
8. \( P[x-1][y-1] \land P[x][y-1] \land P[x+1][y-1] \land P[x-1][y] \) are black all others are white
9. \( P[x-1][y-1] \land P[x][y-1] \land P[x+1][y] \land P[x+1][y+1] \) are black all other are white
10. \( P[x+1][y-1] \land P[x][y-1] \land P[x+1][y] \land P[x-1][y-1] \) are black all other are white
(5 neighbors are black)

1. \( P[x-l][y-l] \land P[x][y-1] \land P[x+l][y-l] \land P[x+l][y-l] \land P[x+l][y+l] \) are black all others are white

2. \( P[x+1][y-1] \land P[x+1][y] \land P[x+1][y+1] \land P[x][y-1] \land P[x][y+1] \) are black all others are white

3. \( P[x+1][y-1] \land P[x+1][y] \land P[x+1][y+1] \land P[x-1][y+1] \land P[x+1][y+1] \) are black all others are white

4. \( P[x-1][y+1] \land P[x][y+1] \land P[x+1][y+1] \land P[x-1][y] \land P[x+1][y] \) are black all others are white

5. \( P[x-1][y+1] \land P[x][y+1] \land P[x+1][y+1] \land P[x-1][y] \land P[x-1][y-1] \) are black all others are white

6. \( P[x-1][y-1] \land P[x-1][y] \land P[x-1][y+1] \land P[x][y-1] \land P[x][y+1] \) are black all others are white

7. \( P[x-1][y-1] \land P[x-1][y] \land P[x-1][y+1] \land P[x][y-1] \land P[x-1][y-1] \) are black all others are white

8. \( P[x-1][y-1] \land P[x][y-1] \land P[x+1][y-1] \land P[x-1][y] \land P[x+1][y] \) are black all others are white

(6 neighbors are black)

1. \( P[x-1][y-1] \land P[x][y-1] \land P[x+1][y-1] \land P[x+1][y-1] \land P[x+1][y+1] \land P[x][y+1] \) are black all others are white

2. \( P[x+1][y-1] \land P[x+1][y] \land P[x+1][y+1] \land P[x][y-1] \land P[x][y+1] \land P[x-1][y+1] \) are black all others are white
3. \(P[x+1][y-1] \land P[x+1][y] \land P[x+1][y+1] \land P[x-1][y+1] \land P[x-1][y] \land P[x+1][y+1] \land P[x-1][y]\) are black all others are white

4. \(P[x-1][y+1] \land P[x][y+1] \land P[x+1][y+1] \land P[x-1][y] \land P[x+1][y] \land P[x-1][y-1]\) are black all others are white

5. \(P[x-1][y+1] \land P[x][y+1] \land P[x+1][y+1] \land P[x-1][y] \land P[x-1][y-1] \land P[x][y-1]\) are black all others are white

6. \(P[x-1][y-1] \land P[x-1][y] \land P[x-1][y+1] \land P[x][y-1] \land P[x][y+1] \land P[x+1][y-1]\) are black all others are white

7. \(P[x-1][y-1] \land P[x-1][y] \land P[x-1][y+1] \land P[x][y-1] \land P[x][y-1] \land P[x+1][y]\) are black all others are white

8. \(P[x-1][y-1] \land P[x][y-1] \land P[x+1][y-1] \land P[x-1][y] \land P[x+1][y] \land P[x+1][y+1]\) are black all others are white

All other conditions will make the status of the pixel as stable.

2.5 Endpoint Erosion

After the thinning algorithm performs its tasks, an endpoint erosion algorithm starts its runs through the image removing lines that do not make a connection by deleting pixels that are at the end of a line.

There are many ways for a person to place a check on a checkbox and it is very common for the check to be drawn outside the borders of the box. This will present a problem because the character does not look like a box when it has lines going through it.
Figure 2.5.1. How endpoint erosion works on the checkmarks.

The method used in dealing with this problem is erosion, which eliminates the lines that are drawn outside the borders of the checkbox.
The erosion transformation of a binary image is based on the successive morphological erosions on the image [20][21]. This algorithm looks at a 3x3 window of an arbitrary pixel and checks to see if it is part of a connection. To do this, the eight pixels around the center pixel are counted. If there is more than one black pixel then we say the center pixel is part of a connection. Otherwise there is no more than one pixel around the center pixel meaning that the center pixel is part of an end point on a line, therefore that pixel is deleted.
Endpoints always have at most one neighbor

Figure 2.5.3. The 3x3 window for endpoint erosion.

This procedure is critical in allowing the checkbox to be easily traced through by the contour following algorithm for checkbox detection.
Figure 2.5.4. How endpoint erosion removes the checkmark.

There is no rule that says people need to make their checkmarks completely within the boundary of the checkbox. This algorithm clearly helps by erasing the checkmarks that go beyond the border while preserving the checkbox boundary.
CHAPTER 3

CHECKBOX RECOGNITION

The checkbox recognition algorithm is the heart of this entire procedure. After the preprocessing is complete, it is assured that the character being examined in this step is ready to run through the algorithm to determine if it is a checkbox or some other character.

At this point, it is given that the character is thinned down to one pixel in width, and an erosion algorithm eliminated handwritten marks that go beyond the boundary.

Figure 3.1 shows the box structure after the preparation algorithms run through the image. You will notice that the edges are not straight, so there will have to be a way to trace the checkbox border by searching around the neighboring pixels. The quality of the scanner and the form has a large impact on the quality of the box after form preparation. If the form and the scanner are good quality then chances are that the box could show perfect edges.
Figure 3.0.1. Sample of an empty checkbox of poor quality.

3.1 Contour Following

Corners are detected using a right turn algorithm. The box detection algorithm will scan the image from the upper right corner going from left to right, and top to bottom. With this method, we will encounter a pixel that may be part of the box. This pixel will be the starting pixel. The starting pixel will always be on the top side of the box and most likely towards the left part of the topside.

The starting pixel, as shown in figure 3.2, is the upper leftmost pixel. From there we will start our search going from left to right across the top of the box.
Figure 3.1.1. The starting pixel is located at the top left.

When we are scanning the sides of the box, we cannot just go across to the next pixel because the sides are not assumed to be smooth. We will need to search the neighboring pixels while maintaining the proper direction of the search.
Figure 3.1.2 illustrates the procedure for searching. The dark shaded box is the current pixel where we start our search. A priority is placed on which pixel to look at. The order is based on where the next pixel should be on the box. It is ordered by the most probable next pixel.

Suppose P (x, y) is the current pixel, where the origin of the image, P (0, 0), is on the bottom left. The following is the ordering of the search for each direction:

RIGHT:

P (x+1, y)
P (x+1, y+1)
P (x+1, y-1)
P (x, y-1)
\[ P(x-1, y-1) \]

**DOWN:**

\[ P(x, y-1) \]
\[ P(x+1, y-1) \]
\[ P(x-1, y-1) \]
\[ P(x-1, y) \]
\[ P(x-1, y+1) \]

**LEFT:**

\[ P(x-1, y) \]
\[ P(x-1, y-1) \]
\[ P(x-1, y+1) \]
\[ P(x, y+1) \]
\[ P(x+1, y+1) \]

**UP:**

\[ P(x, y+1) \]
\[ P(x-1, y+1) \]
\[ P(x+1, y+1) \]
\[ P(x+1, y) \]
\[ P(x+1, y-1) \]

Notice each directional search consists of five out of eight pixels in the 3x3 window (the ninth pixel is the current pixel). The remaining three
pixels do not need to be visited because that will move the search in the opposite direction.

Since we are looking for a box, we will have four right turns since we are going clockwise around the box. The order of the search will be to go right across the top, down the right side, left across the bottom, up the left side, and right across the top to look for the starting pixel. When the starting pixel is found, then the box is found. At every location we will check to see if a right turn took place. Right turns indicate when the direction of the search has changed and a corner was found.

To check for a right turn, we look back a certain amount of pixels and we subtract to find the difference in the coordinates. Since the height and width of the character is known, the number of pixels to look back is $\frac{1}{4}$ times the smaller value between the height and width. If the difference is greater than the turning tolerance, then a right turn took place and the direction of the search changes. This all can be summarized by the following equations.

\[
\begin{align*}
  n & = \text{number of pixels to look back} \\
  t & = \text{turning tolerance} \\
  s & = w \text{ if } (w < h) \text{ otherwise } s = h, \text{ where } w = \text{width and } h = \text{height} \\
  n & = \frac{1}{4} (s) \\
  1 & < t < n 
\end{align*}
\]
The turning tolerance is a small number depending on how many pixels back you are comparing to. The tolerance has to be less than the look back value. If you were subtracting the difference from four pixels back, then a good tolerance would be three. If you decide to subtract from seven pixels back, then a good tolerance can range from four to six depending on how tight you want to make it.

For poor quality documents, it is best to look back more pixels because the edges won't be as straight. If the document is in perfect condition, then looking back one pixel will find boxes if the edges of the box are straight. The turning tolerance can never be greater than the number of pixels to look back. Since it is a measure of distance, having a greater tolerance is not possible.

Starting with the start pixel, we search across the top going right and following the path of the pixels. The first place we will look is the right adjacent pixel. If that pixel is black then we set that pixel as the current pixel and continue the search. If that pixel is white, then try the upper right pixel. This step is repeated on all pixel search locations until a white pixel is found. If a black pixel is not found then this character is not a box.

After every search, we check for a right turn. Since we are going across the top and moving right, we are looking for a right turn to change the search pattern to go downward.
After the third right turn, the search pattern is moving up along the left side of the box looking for either the fourth right turn, or the starting pixel. A fourth right turn will change the search pattern back to the rightward search. Once the starting pixel is found, then we conclude that a box shaped character is found.

3.2 Roundness

We may come across situations where we find a character that will pass through that have the characteristics of a box when they really are not boxes. The letter “O” looks like a checkbox since its features have four corners, depending on the scan quality and font. To get around this problem, we will measure the perimeter.

The perimeter of a circle inscribed in a square is smaller than the perimeter of a square. We know the outline of the character by measuring the minimum and maximum of the X-coordinate and the Y-coordinate. We also know the perimeter of the character by counting the pixels during the search that was used to determine if it was a box. While running through the contour following algorithm, the number of pixels that run through it are counted. If the perimeter of the character is significantly smaller than the perimeter of the outline, then the character is not a box and is a circle or the letter “O.” If the perimeter of the character is about the same as the perimeter of the outline then it is
a box. The largest x-value, smallest x-value, largest y-value, and smallest y-value are recorded during the contour following. Using these four values, a rectangular perimeter is formed. This rectangular perimeter will be used to match it with the character to see how round or rectangular it is.

![Diagram of a circle and a square with labels for width, height, and radius.](image)

There are more pixels on the perimeter of the square than there are in the circle.

Figure 3.2.1. The perimeters of a circle and a square.
It is easy to prove that this is true. Assume the letter "O" is a circle with diameter \(d\) and assume that a checkbox is square with length \(l\) and width \(w\). The perimeter of a circle is \(\pi d\) and the perimeter of the checkbox is \(2(w + h)\). Since the checkbox is a square then \(w = h\). Also, the diameter of the circle is equal to the height and width of the square so \(d = h = w\). So, \(2(w + h) = 2(d + d) = 2(2d) = 4d\), and \(4 > \pi\).

3.3 Algorithm Outline

The following is an overview of the algorithm that was just covered:

1. Locate the start pixel. Keep in mind the starting pixel will be on the top part of box since the document is assumed to have started the search for each box from the upper left corner.

2. Search each pixel going right.

![Figure 3.3.1. Search is moving to the right.](image)

3. Check for a right turn. If a right turn took place then proceed to step 4 otherwise repeat step 2.
4. Search each pixel going down.

\[ \begin{array}{ccc}
  &  & \\
 5 & 4 & \\
 3 & 1 & 2 \\
\end{array} \]

Figure 3.3.2. Search is moving down.

5. Check for a right turn. If a right turn took place then proceed to step 6 otherwise repeat step 4.

6. Search each pixel going left.

\[ \begin{array}{ccc}
  &  & \\
 3 & 4 & 5 \\
 1 & \\
 2 & \\
\end{array} \]

Figure 3.3.3. Search is moving left.

7. Check for a right turn. If a right turn took place then proceed to step 8 otherwise repeat step 6.

8. Search each pixel going up.
9. Check to see if the current pixel is the starting pixel. If it is, then go to step 12. If not then check to see if a right turn took place. If a right took place then proceed to step 10 otherwise repeat step 8.

10. Search each pixel going right (See figure 3.3.1).

11. Look for the starting pixel. If the starting pixel is found then go to step 12 otherwise repeat step 10.

12. Examine the perimeters of the outline to the box. If the perimeters are similar in value then this is a checkbox otherwise it is not a checkbox.

This box detection algorithm will work on any arbitrary form and has the ability to check characters to see if they are boxes even when the image of the boxes is not smooth. It is important to prepare the form before checking for boxes because people have different ways of making check marks on boxes, and some check marks are so exaggerated that the box can lose characteristics of a box making it difficult for the
algorithm to detect it. This algorithm works well even with poor quality boxes with edges that are not straight.

This algorithm does a remarkable job at telling the difference from the letter “O” and checkboxes. This was a problem since the Letter “O” and even the letter “Q” have similar features to checkboxes.

By maintaining a pixel count during the contour following procedure, the numbers of pixels that are in the checkbox have been crucial in determining the squaring of a checkbox. This method does not have to be normalized since if it was the letter “O”, the algorithm assumed it was a checkbox and compared the pixel count to what the value of the perimeter should be.
CHAPTER 4

OPTICAL MARK RECOGNITION

Once a box is found, the next step is to determine if it is checked or unchecked. This is a simple and fast search that requires you to vertically and horizontally search the box, counting the number of black pixels. When searching vertically, start the search from above the upper boundary of the box and go down below the lower boundary of the box.

When you start the search, you will come across a pixel at the top of the box and below the box. So if the box is not checked, you will find 2 pixels in each search.

![Scan Line Intersects](image)

Figure 4.0.1. Scan lines to look for a check.
Figure 4.0.2. This shows the range of the scan lines.

If you count more than two black pixels, then the box is checked because you will come across at most two pixels if the box is unchecked since the sides of the box contain black pixels. By scanning from left to right vertically, you will be able to come across a check mark if it exists.

As you can see in Figure 4.1, the box on the left is unchecked and each scan line only detects two pixels. The box on the right is checked and each scan line detects more than two pixels. When a scan line encounters three or more pixels, then the scan line has found a possible check mark.

Since it is possible to have some noise in the box rather than a checkmark, all the scan lines are polled to determine if a significant amount of scan lines have encountered a checkmark.
It is important to scan the box vertically and horizontally. There are so many ways a person can check a box since there is no standard on the way a checked box is marked. A vertical scan may not be able to pick up a check mark if the check is small and off to one side since the vertical line scan does not search near the sides of the box. If the vertical line scan is too close to the sides of the box then it will detect the side of the box and will encounter more than two pixels. To avoid this problem, the horizontal scan will search the box the same way the vertical line scan. So if the box is checked and the vertical scan does not pick up the check, the horizontal scan will pick it up. The same would apply if the horizontal scan does not pick up the check mark, then the vertical scan will.
CHAPTER 5

LOCATION

An OCR algorithm is meaningless without a method to determine its location in the document. Most forms have multiple checkboxes so it is important to determine the location of the checkboxes with respect to each other on the form. A more advanced method would determine the location of the checkboxes with respect to the form.

As stated before, there are two ways to process checkboxes. One way is to feed the form through an OCR routine to find the numbers and letters to string them together to formulate the words through a dictionary. When an unknown character is found it runs through the checkbox recognition algorithm. In this case, if a checkbox is found then the location of the checkbox is already known based on the processed words. However in the other use of this algorithm in a stand-alone checkbox routine, only checkboxes are recognized. This is the type of routine where the user would only need to process a form to get feedback based on checkboxes such as multiple-choice exams. In this case the user does not need to know what is stated in the document but needs to
get handwritten input from the checkboxes. A stand-alone procedure
would need to identify the checkboxes in the order that they appear.

For this type of procedure, the input would be a scanned form and the
output would be a list of box id's along with an identifier that states
whether the box is checked or unchecked.

Please check the following that apply.

1) The color of the moon is  Blue □  Red □

2) Where are you?       Here □  Earth □

3) A pound of feathers is less than a pound of salt.
   TRUE □
   FALSE □

4) Please leave these checkboxes blank. □  □

Figure 5.0.1. Actual sample output of checkbox detection algorithm.

The program outlines the check marked boxes in red and the blank
checkboxes in green.
Figure 5.0.2. The output after the program finds the checkboxes.

First, there would have to be a coordinate system for this type of identification. Assume that the document is in the image coordinate system with the top left corner at (0,0) and the units are measured in pixels.

- Let $x$ be the number of pixels from the left column
- Let $y$ be the number of pixels from the top row
- The coordinates will then be $(x, y)$
It takes two sets of coordinates to define the location of a checkbox. One set of coordinates will define the upper left corner and the second set of coordinates will define the bottom right corner. Let \((x_1, y_1)\) be the coordinates of the upper left corner and let \((x_2, y_2)\) be the coordinates for the bottom right. Then the following is true:

\[
\begin{align*}
x_1 &< x_2 \\
y_1 &< y_2
\end{align*}
\]
In order to know which checkbox comes first, a rank number is used to order the checkboxes. Since documents are read top to bottom, left to right, the checkboxes will also be ordered that way. A checkbox that is closer to the top of the form will be first to any checkboxes below it. Obviously, if two checkboxes are on the same line, then the checkbox to the left will rank above the one to the right.
The coordinates to determine the location of a checkbox will be the center of the checkbox. Recall that \((x_1,y_1)\) are the coordinates for the top left corner and \((x_2,y_2)\) are the coordinates for the bottom right corner. Then the center of the checkbox is

\[
\left( \frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2} \right)
\]

Let \(X\) be the center \(x\)-coordinate of the checkbox and \(Y\) be the center \(y\)-coordinate of the checkbox.

\[
X = \frac{x_1 + x_2}{2}
\]
\[
Y = \frac{y_1 + y_2}{2}
\]

The center coordinates will represent the location of a checkbox, hence the coordinates of a checkbox will be \((X,Y)\).

Not all forms get scanned properly and can have some distortion that could cause each line on the form not be parallel to the image space coordinates. This could affect the ordering of the checkboxes, so a method has to be in place to correct the problem. Even if the scan was good there could still be a small distortion even if it is a one-pixel difference.

The checkboxes are to be ordered by each line that they are on. A sorting routine will be used to determine which line each checkbox goes
on. The lines will be ranked going from the top to the bottom of the document. Once it is determined which line each checkbox is associated with, they will be ordered with respected to each other checkbox in the line starting with the leftmost checkbox.

The first step is to sort the checkboxes in increasing y-values with respect to their coordinates, (X, Y). This will order the checkboxes from the closest to the top of the document to the bottom. If there are more than one equal y-values, it will not matter which one is above the other since the second step will classify them as being on the same line on the document.

The second step will determine which checkboxes are on the same line even if the document is not aligned properly. Starting with the first checkbox on the sorted list, exam the y-values of each checkbox. Recall that (x1, y1) is the top left corner and (x2, y2) is the bottom right. Let Ci be a checkbox in the sorted list and Ci+1 be the next checkbox in the sorted list. So the coordinates for Ci is ( (Ci x1, Ci y1), (Ci x2, Ci y2) ) and the coordinates for Ci+1 is ( (Ci+1 x1, Ci+1 y1), (Ci+1 x2, Ci+1 y2) ).
Figure 5.0.5. The layout of some checkboxes on a form.

*Note: checkbox 8 is not aligned properly
There are two possible outcomes. One outcome is both checkboxes are on the same line as each other. The other outcome is if the two checkboxes are completely on a different line from each other on the document. To be classified as being on the same line $C_i y_2$ has to be greater than $C_{i+1} y_1$.  

Figure 5.0.6. The checkboxes shown in a sorted list.
Figure 5.0.7. These two checkboxes are on the same line.

The third and final step once the checkboxes are in their proper line number with respect to the document is to sort them within their line in ascending x-value. This will make it so the checkboxes are arranged from top to bottom and left to right.
Figure 5.0.8. Breakdown on how the checkboxes are grouped.
CHAPTER 6

CONCLUSION

Having the ability to have an OCR algorithm that treats checkboxes as characters can make transferring documents from paper to electronic simple and efficient. This algorithm can be a useful tool for people who use checkboxes on forms who need to store their hard copy data electronically.

Not all documents are preserved in good quality. This algorithm was designed with the intent to interpret some of the oldest paper documents that have been damaged by age.

Some of the keys to this checkbox detection algorithm are its preprocessing methods. The algorithm will not be able to function properly without first running through a thinning algorithm and then an endpoint erosion algorithm to remove checkmarks that stretch beyond the borders.

A contour following procedure with a prioritized search pattern is the main reason for the success to trace around a checkbox without accidentally tracing a checkmark. This particular search pattern is one
of the key contributions to allowing an algorithm successfully detect checkboxes, with or without checkmarks in them.

There were some challenges along the way in the development of this algorithm. One of them was that this algorithm used to classify the letter "O", "Q", and the letter "D" as checkboxes. Instead of doing multiple passes through the characters to eliminate this problem, tracing through the character in the contour following procedure and keeping a count on the total number of pixels helped eliminate misclassifications. By keeping a pixel count and the possible boundary of a checkbox, they are compared to see how close the match is.

This algorithm was designed with the intent for it to be used with or without another OCR method that can detect characters. When working with an OCR method that detects characters, that OCR system would come across unknown characters that could potentially be checkboxes. The OCR system would pass the unknown character to the checkbox detection algorithm to determine if it is a checkbox. The checkbox detection algorithm could also work on its own by scanning an entire document looking for checkboxes. When checkboxes are found, their coordinates are stored. A localization procedure determines the order in which the checkboxes are in to accurately output its results.

An OMR algorithm determines if predefined areas are marked. Once the checkbox detection algorithm finds a checkbox, the area inside the checkbox is searched to see if it has a mark.
The checkbox detection algorithm plays an important role in document conversions to digital. In the not too distant past, it was only a dream to think that a computer could be programmed to read text and interpret its meaning. That dream has become a reality, and this checkbox detection algorithm makes it more promising.
APPENDIX

SOURCE CODE

BoxDetection(CImageObject* image)
//PURPOSE: Find check boxes
//IN: CImageObject *image – the document in image format
{
    int length,width;
    float ratio1,ratio2,match;
    int x,y,perimeter;
    BoxInfo *box_info_head = NULL;
    bool firsttime = true;
    int cx1, cy1, cx2, cy2, pix_cnt;
    int imageWidth = image->GetWidth();
    int imageHeight = image->GetHeight();

    for(y=5; y < imageHeight-5; y++)
        for(x=5;x<imageWidth-5; x++)
            if (image->GetPixel(x,y) == 0)
                if (IsBox(image,true,x,y,cx1, cy1, cx2, cy2, pix_cnt))
                    {
                        length = cx2-cx1;
                        width = cy2-cy1;
                        perimeter = 2*(length + width);
                        ratio1 = float(length)/float(width);
                        ratio2 = float(width)/float(length);
                        match = float(perimeter)/float(pix_cnt);

                        // Check to see if there is another box inside
                        // perimeter-15
                        if ( (ratio1 < 1.150) 
                            && (ratio2 < 1.150) 
                            && (perimeter > 40) && (match < 1.250) )
First time, set up the head of the list
if (firsttime)
{
    firsttime = false;
    m_box_info = new BoxInfo;
    box_info_head = m_box_info;
}
else
{
    m_box_info->next = new BoxInfo;
    m_box_info = m_box_info->next;
}

m_box_info->x1 = cx1;
m_box_info->x2 = cx2;
m_box_info->y1 = cy1;
m_box_info->y2 = cy2;
m_box_info->next = NULL;

// Check for check marks
GetOriginalImage(image);
Thinning(image);

// Scan the boxes horizontally
m_box_info = box_info_head;
while (m_box_info != NULL)
{
    for (y = abs(int(float(m_box_info->y1 - m_box_info->y2)*0.10)) + m_box_info->y1; 
         y < abs(int(float(m_box_info->y1 - m_box_info->y2)*0.90)) + m_box_info->y1; y++)
    {
        pix_cnt = 0;
        for (x = abs(int(float(m_box_info->x1 - m_box_info->x2)*0.10)) + m_box_info->x1; 
             x < abs(int(float(m_box_info->x1 - m_box_info->x2)*0.90)) + m_box_info->x1; x++)
        {

    }

}
if (image->GetPixel(x,y)==0)
{
    pix_cnt++;
}

if (pix_cnt > 2)
    m_box_info->isChecked = true;
else
    m_box_info->isChecked = false;

m_box_info= m_box_info->next;
}

////////////////////////////////////////////////////////////////////////////////
// Scan the boxes vertically
////////////////////////////////////////////////////////////////////////////////

m_box_info = box_info_head;

while(m_box_info != NULL)
{     
    for(x=abs(int(float(m_box_info->x1-m_box_info->x2)*0.1))+m_box_info->x1; x < abs(int(float(m_box_info->x1-m_box_info->x2)*0.90))+m_box_info->x1; x++)
    {
        pix_cnt = 0;
        for(y=abs(int(float(m_box_info->y1-m_box_info->y2)*0.1))+m_box_info->y1; y < abs(int(float(m_box_info->y1-m_box_info->y2)*0.90))+m_box_info->y1; y++)
        {
            if (image->GetPixel(x,y)==0)
            {
                pix_cnt++;
            }
            if (pix_cnt > 2)
                m_box_info->isChecked = true;
        }
    m_box_info= m_box_info->next;
}

////////////////////////////////////////////////////////////////////////////////
GetOriginalImage(image);

m_box_info = box_info_head;
while(m_box_info != NULL)
{
    if (m_box_info->isChecked)
        DrawOnImage(image,
                     m_box_info->x1-2,
                     m_box_info->y1-2,
                     m_box_info->x2+2,
                     m_box_info->y2+2,
                     255,0,0);
    else
        DrawOnImage(image,
                     m_box_info->x1-2,
                     m_box_info->y1-2,
                     m_box_info->x2+2,
                     m_box_info->y2+2,
                     0,200,0);

    m_box_info= m_box_info->next;
}

bool CFPData::IsBox(CImageObject *image,bool pixel_color,int start_x, int
                     start_y,int &cx1,int &cy1, int &cx2,int &cy2,int &count)
//PURPOSE: Box Detection Algorithm, follows the edge to determine if
//it's a box
//IN:
//    bool pixel_color = true will set the color of visited pixels
//    start_x = starting x pixel
//    start_y = starting y pixel
//OUT:
//    cx1 = upperleft corner for the x
//    cy1 = upperleft corner for the y
//    cx2 = lowerright corner for the x

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


// cy2 = lower right corner for the y
// count = number of pixels counted that describes the box

{ int x,y,turnX[8],turnY[8];

  count=0;
  x = cx1 = cx2 = start_x;
  y = cy1 = cy2 = start_y;
  turnX[0]=start_x;
  turnY[0]=start_y;

  //Color the first pixel
  if (pixel_color)
  {
    image->SetPixel(x,y,RGB(1,0,0));
  }

  //Set the direction
  bool go_right=true,go_down=false,go_left=false,go_up=false;

  int turn_count=0; // number of pixels after a turn;

  do
  {

    //Check boundary conditions on the image
    if ((x+5) > image->GetWidth()) return false;
    if ((y+5) > image->GetHeight()) return false;

    count++;
    //Going RIGHT
    if (go_right)
    {
      if (image->GetPixel(x+1,y) ==0)
        x++;
      else if(image->GetPixel(x+1,y-1) ==0)
      {
        x++;y--;}
      else if(image->GetPixel(x+1,y+1) ==0)
      {
        x++;y++;
      }
      else if (image->GetPixel(x,y+1) ==0)
        y++;

    }

  } while (count < count_limit);

  image->SetPixel(x,y,RGB(1,0,0));

  return true;

}
else if(image->GetPixel(x-1,y+1) ==0)
{
    y++;x-;
}
else // Not a box
{
    return false;
}

turnX[turn_count%8]=x;
turnY[turn_count%8]=y;
turn_count++; 

//Check for a right turn
if ((turn_count > 4) && ((y-turnY[(turn_count-4)%8])>2))
{
    go_right=false;
go_down=true;
turn_count = 0;
}

//Going DOWN
else if (go_down)
{
    if (image->GetPixel(x,y+1) ==0)
        y++;
    else if(image->GetPixel(x+1,y+1) ==0)
    {
        x++;y++;
    }
    else if(image->GetPixel(x-1,y+1) ==0)
    {
        y++;x-;
    }
    else if (image->GetPixel(x-1,y) ==0)
        x--; 
    else if(image->GetPixel(x-1,y-1)==0)
    {
        y--;x--;
    }
    else // Not a box
        return false;

    turnX[turn_count%8]=x;
turnY[turn_count%8]=y;

turn_count++;
//Check for a right turn
if ((turn_count > 4) && ((x-turnX[(turn_count-4)%8])<-2))
{
    go_left=true;
    go_down=false;

    turn_count = 0;
}
}
//Going LEFT
else if (go_left)
{
    if (image->GetPixel(x-1,y) ==0)
        x--;
    else if(image->GetPixel(x-1,y+1) ==0)
        {x--;y++;}
    else if(image->GetPixel(x-1,y-1) ==0)
        {x--;y--;}
    else if (image->GetPixel(x,y-1) ==0)
        y--;;
    else if(image->GetPixel(x+1,y-1)==0)
        {x++;y--;}
    else // Not a box
        return false;

    turnX[turn_count%8]=x;
    turnY[turn_count%8]=y;

    turn_count++;
//Check for a right turn
if ((turn_count > 4) && ((y-turnY[(turn_count-4)%8])<-2))
{
    go_left=false;
    go_up=true;
    turn_count = 0;
}
}
/Going UP
else if (go_up)
{
    if (image->GetPixel(x,y-1) ==0)
        y--; 
    else if(image->GetPixel(x-1,y-1) ==0)
    {
        x--;y--; 
    }
    else if(image->GetPixel(x+1,y-1) ==0)
    {
        y--;x++; 
    }
    else if (image->GetPixel(x+1,y) ==0)
        x++; 
    else if(image->GetPixel(y+1,x+1)==0)
    {
        x++;y++; 
    }
    else // Not a box
        return false;

    turnX[turn_count%8]=x;
    turnY[turn_count%8]=y;

    turn_count++;

    //Check for a right turn
    if ((turn_count > 4) && ((x-turnX[(turn_count-4)%8])>2))
    {
        go_left=false;
        turn_count = 0;
    }

    //Find the starting pixels
    else
    {
        if (image->GetPixel(x+1,y) ==0)
            x++; 
        else if(image->GetPixel(x+1,y+1) ==0)
        {
            x++;y++; 
        }
        else if(image->GetPixel(x+1,y-1) ==0)
        {
            x++;y--; 
        }
    }
else if (image->GetPixel(x,y+1) == 0)
    y++;
else if (image->GetPixel(x-1,y+1) == 0)
    {
        y++;x--;
    }
else // Not a box
    {
        return false;
    }
tumX[tum_count%8]=x;
tumY[tum_count%8]=y;
tum_count++;

    //Return TRUE if the box is found
    if ((x==start_x) && (y==start_y))
        return true;
}

//MARK IT VISITED with Deep Blue
if (pixel_color)
    {
        image->SetPixel(x,y,RGB(0,0,1));
    }

//Limit the number of searches
//In other words, no rediculously large boxes!
if (tum_count > 4024) return false;

//Find out where to draw the green box
if (x<cx1)cx1=x;
if (y<cy1)cy1=y;
if (x>cx2)cx2=x;
if (y>cy2)cy2=y;

}while (((x != start_x) || (y != start_y)));

return true;
}
BIBLIOGRAPHY


VITA

Graduate College
University of Nevada, Las Vegas

John Michael Istle

Home Address:
11243 Tribiani Ave
Las Vegas, NV 89138

Degrees:
Bachelor of Science, Computer Science, 2003
University of Nevada, Las Vegas

Bachelor of Science, Mathematics, 2003
University of Nevada, Las Vegas

Publications:


Thesis Title: Optical Character Recognition for Checkbox Detection

Thesis Examination Committee:
Chairperson, Dr. Evangelos Yfantis, Ph. D.
Committee Member, Dr. Nina Cooper, Ph. D.
Committee Member, Dr. Wolfgang Bein, Ph. D.
Graduate Faculty Representative, Dr. Emma Regentova, Ph. D.