
Electronic Theses and Dissertations, 2004-2019

2006

Text-image Restoration And Text Alignment For Multi-engine Optical Character Recognition Systems

Nikolai Kozlovski
University of Central Florida



Part of the [Electrical and Electronics Commons](#)

Find similar works at: <https://stars.library.ucf.edu/etd>

University of Central Florida Libraries <http://library.ucf.edu>

This Masters Thesis (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations, 2004-2019 by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation

Kozlovski, Nikolai, "Text-image Restoration And Text Alignment For Multi-engine Optical Character Recognition Systems" (2006). *Electronic Theses and Dissertations, 2004-2019*. 850.

<https://stars.library.ucf.edu/etd/850>



University of
Central
Florida

Showcase of Text, Archives, Research & Scholarship

STARS

TEXT-IMAGE RESTORATION AND TEXT ALIGNMENT
FOR MULTI-ENGINE OPTICAL CHARACTER RECOGNITION SYSTEMS

by

NIKOLAI KOZLOVSKI
B.S.Cp.E. University of Central Florida, 2004

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2005

© 2005 Nikolai Kozlovski

ABSTRACT

Previous research showed that combining three different optical character recognition (OCR) engines (ExperVision® OCR, Scansoft OCR, and Abbyy® OCR) results using voting algorithms will get higher accuracy rate than each of the engines individually. While a voting algorithm has been realized, several aspects to automate and improve the accuracy rate needed further research.

This thesis will focus on morphological image preprocessing and morphological text restoration that goes to OCR engines. This method is similar to the one used in restoration partial finger prints. Series of morphological dilating and eroding filters of various mask shapes and sizes were applied to text of different font sizes and types with various noises added. These images were then processed by the OCR engines, and based on these results successful combinations of text, noise, and filters were chosen.

The thesis will also deal with the problem of text alignment. Each OCR engine has its own way of dealing with noise and corrupted characters; as a result, the output texts of OCR engines have different lengths and number of words. This in turn, makes it impossible to use spaces a delimiter as a method to separate the words for processing by the voting part of the system. Text aligning determines, using various techniques, what is an extra word, what is supposed to be two or more words instead of one, which words are missing in one document compared to the other, etc. Alignment algorithm is made up of a series of shifts in the two texts to determine which parts are similar and which are not. Since errors made by OCR engines are due to visual misrecognition, in addition to simple character comparison (equal or not), a technique was developed that allows comparison of characters based on how they look.

Dedicated to Jane, mother, father, grandmother, and Jack.

ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. Arthur Weeks and Dr. Samuel Richie for academic support and supervision while working on this thesis. Also, I would like to thank Dr. Donald Malocha, for reading and correcting some of the earlier drafts.

TABLE OF CONTENTS

ABSTRACT.....	iii
ACKNOWLEDGMENTS	v
TABLE OF CONTENTS.....	vi
LIST OF FIGURES	viii
LIST OF TABLES.....	xii
LIST OF ACRONYMS/ABBREVIATIONS.....	xiv
CHAPTER ONE: INTRODUCTION.....	1
OCR	1
Previous Work	2
Problem Statement	6
Proposed Solution.....	6
CHAPTER TWO: MORPHOLOGICAL IMAGE PROCESSING.....	9
Introduction to Morphological Image Filters.....	9
Implementation of Morphological Filters for Binary Images.....	13
Morphological Filters And Image Restoration	14
Benchmarking Performance of Optical Character Recognition Engines	23
Relating Sizes and Shapes of Structuring Elements of Morphological Filters to Font Sizes and Types of Noise	25
CHAPTER THREE: TEXT ALIGNMENT	36
Introduction to Text Alignment	36
Text Alignment of Two Documents Using ASCII Comparison.....	38

Analysis of the Performance of Aligning Algorithm Based Only on Plain ASCII Comparison	49
Text Alignment of Two Documents Using Visual Comparison.....	51
Analysis of the Performance of Aligning Algorithm Based on Visual Character Comparison	66
Analysis of the Performance of Aligning Algorithm Based on Visual Character Comparison and Line Representation of Characters	74
Improving Algorithm for Finding Next Similar Part.....	75
Alignment of individual words of two texts	80
Alignment of individual words of three texts	88
Performance of Aligning Algorithm.....	92
CHAPTER FOUR: CONCLUSIONS.....	94
APPENDIX A: SOURCE CODE FOR MORPHOLOGICAL FILTERS	96
APPENDIX B: SOURCE CODE FOR TEXT ALIGNMENT ALGORITM	101
LIST OF REFERENCES	115

LIST OF FIGURES

Figure 1: Example of Distortion due to Filters Built-in into Copiers	6
Figure 2: Result of Change-tracking Feature of Microsoft® Word® 2000.	8
Figure 3: a) Example of Dilation Filter, b) Example of Erosion Filter.....	10
Figure 4: Example of Opening Filter (Erosion Followed by Dilation).....	11
Figure 5: Example of Closing Filter (Dilation Followed by Erosion).....	12
Figure 6: a) Broken Character; b) Restored Character by Closing Filter	12
Figure 7: Binary Image and Structuring Element	13
Figure 8: Summation of Overlapping Pixels	13
Figure 9: a) Dilation, and b) Erosion	14
Figure 10: Image that was used for benchmarking the performance of median and opening and closing morphological filters.	15
Figure 11: Portion of Image of Figure 10 That Will Be Used to Examine Various Noises and Effects of Filters.....	16
Figure 12: a) Image with Custom Noise, b) Applied to It Closing Filter 5x5	16
Figure 13: a) Image with Pepper Noise of probability 0.05, b) Applied to It Opening Filter 5x5	17
Figure 14: a) Image with Salt Noise of probability 0.05, b) Applied to It Closing Filter 5x5	18
Figure 15: a) Image with Salt and Pepper Noise of probabilities 0.10 each, b) Applied to It Median Filter 5x5.....	19
Figure 16: a) Image with Salt Noise of probabilities 0.4 each, b) Applied to It Closing Filter 5x5	20
Figure 17: Simulation of Suspect Characters and Words and Its Exponential Fit.....	24

Figure 18: One of the Eleven Images of Text of Font Size 10pt in This Case Scanned at 300dpi That Is Used Throughout This Section	26
Figure 19: Ratio of Image Difference before and after Application of Closing Filter versus Mask and Font Sizes	27
Figure 20: Ratio of Image Difference before and after Application of Opening Filter versus Mask and Font Sizes	28
Figure 21: Number of Good Characters Recognized by Abby’s OCR Engine versus Mask and Font Sizes	30
Figure 22: Number of Good Characters Recognized by Omni’s OCR Engine versus Mask and Font Sizes	31
Figure 23: Seven Shapes of Masks of Size 7x7	32
Figure 24: a) Image, b) Result of FineReader, c) Result of OmniPage, d) Result of Expervision	36
Figure 25: Algorithm for Text Aligning	39
Figure 26: Algorithm for Finding Next Similar Part of Text.....	44
Figure 27: Diagram for Algorithm for Finding Next Similar Part of Text.....	46
Figure 28: Diagram for Algorithm for Finding Next Similar Group of Characters	47
Figure 29: Example of Visual Character Comparison	52
Figure 30: a) Capital “W” of Font Times New Roma, b) Capital “W” of Font Courier New.....	52
Figure 31: Two characters that need to be aligned	53
Figure 32: Character padded with zeros	53
Figure 33: Correlation of two characters shown in Figure 31	54
Figure 34: Difference of aligned characters; in white shown overlapped parts and in grey shown parts that did not overlap.....	55

Figure 35: Difference of aligned characters; certain parts of unaligned pixels are de to difference in width of some features of characters	55
Figure 36: a) Unaligned pixels, b) Unaligned pixels after application of closing filter with structuring element of cross shape and size 3x3 pixels	56
Figure 37: Closed unaligned pixels added beck to aligned pixels	57
Figure 38: Graphical representation of amount of pixels of each of three colors.....	58
Figure 39: Graphical representation of equations 3.5, 3.6, and 3.7	60
Figure 40: Graphical representation of equations 3.6, 3.7, and 3.8	62
Figure 41: Example of Success of Visual Character Comparison.....	67
Figure 42: Example of Line Representation and Comparison of Characters	68
Figure 43: Line Representation of Characters “a” through “p”	69
Figure 44: Vertical Summation of Pixels of a) Character “H” and b) Character “h”	70
Figure 45: 3-level Threshold Applied to Figure 44	70
Figure 46: Characters of Font Impact	71
Figure 47: Algorithm for Detecting Next Similar Character Using Line Representation of Characters	72
Figure 48: Algorithm for Detecting Next Similar Character Using Line Representation of Characters in Action	73
Figure 49: Matrix-like Representation of All Possible Three-character Combinations within Two Six-character Long Strings	76
Figure 50: Shaded Regions Show Where Combination of Compared Characters Starts for a) Three-, b) Four-, c) Five-, and d) Six-Character Long Masks.....	76
Figure 51: Translating Matrix of Characters into Matrix of Difference of Characters	77

Figure 52: Masks a) [1 1 1], b) [1 0 1 1], and c) [1 1 0 1] Applied to Character Difference Matrix	78
Figure 53: a) Minimum Values of the Three Matrices, b) Number of Matrix to Which the Minimum Value Belongs	78
Figure 54: a) Minimum Values of the Three Matrices after Threshold Was Applied, b) Number of Matrix to Which the Minimum Value Belongs	79
Figure 55: Sum of Squares of Shifts versus Position in Difference Matrix in 3D and 2D views.	80
Figure 56: Example of Combination of cases 1.2 and 2.1	84
Figure 57: Example of Combination of cases 1.1 and 2.1	85
Figure 58: Example of Combination of cases 1.3 and 2.1	86
Figure 59: Algorithm for Aligning of Three Texts	89
Figure 60: Example of Aligning of Three Texts	90

LIST OF TABLES

Table 1 Examples of Errors Found in Multi-engine Environment	3
Table 2 Accuracy Improvements with Voting.....	5
Table 3 Abbreviations and Descriptions of Noises Used	21
Table 4 First Set of Images Processed by Abbyy	21
Table 5 First Set of Images Processed by Omni.....	22
Table 6 Results of Application of Closing Filter and ABBYY's OCR Processing.....	32
Table 7 Results of Application of Closing Filter and Omni's OCR Processing.....	33
Table 8 Results of ABBYY's OCR Processing of 375 Images with Various Font Sizes, Noise Densities, and Filtered by Closing Filter of Various Mask Sizes	34
Table 9 Result of Aligning Text Using Space as a Delimiter.....	37
Table 10 Example of Aligning Process Based on the Algorithm of Figure 25	41
Table 11 Sample Data Stored in One of Three Lists after First Part of Alignment.....	50
Table 12 Visually Similar Fragments of Texts That Were Not Flagged as Similar During Plain ASCII Comparison.....	51
Table 13 Fragment of a table for character similarity look-up	63
Table 14 Fragment of a table for character difference look-up	64
Table 15 Four Characters with Most Similar to Them Characters	65
Table 16 Misalignment due to High Thresholds.....	66
Table 17 Visually Similar Fragments of Texts That Were Not Flagged as Similar During Visual Character Comparison	67

Table 18 Groups of Characters That Were Marked as Similar by Adding Line Representation of Characters	74
Table 19 Examples of Masks and Their Applications	77
Table 20 Possible Combinations of Numbers of Spaces in Different Parts	81
Table 21 Possible Combinations of Ending and Beginning of Similar Parts	82
Table 22 Combinations of Cases of Table 20 and Table 21 and Corresponding to Them Actions	82
Table 23 Sample of Two Aligned Texts	86
Table 24 Examples of Merged Words after Aligned of Two Texts	87
Table 25 Sample of Three Aligned Texts	91

LIST OF ACRONYMS/ABBREVIATIONS

OCR	Optical Character Recognition
ASCII	American Standard Code for Information Interchange

CHAPTER ONE: INTRODUCTION

OCR

In this modern day and age it is hard to imagine printing a book or an article without first typesetting it on a computer. This not only gives the ability to easily edit and change the text but also distributes work electronically and search pages and pages of valuable information with single stroke of keyboard. It might come as a surprise that about 90% of all information that is available only as a hard copy [1]. This information is made up of old books and articles that are still of a value to the society but have never been transformed into digital format, publications for which digital copies are forever lost, or government related papers that have been typed using typewriters.

When somebody needs to access a certain article a request is sent out to an archive warehouse, where personnel then have to physically locate the document, make a copy, and then send it back, to the person who requested the document. In case of a small article it might not be such a big deal. If, on the other hand, a corporate lawyer who needs to see every legal case between years of 1981 and 2005 that had to do with guns, drugs, oil, and violence on TV, literally thousands of pages would be received. At this point there are several options; the lawyer could hire somebody to help find whatever it is he/she is looking for, the lawyer could hire somebody to enter everything on the computer and then search through the material using some kind of search engine, or he/she could ask a computer to convert everything from hard copy into digital format. While OCR technology is constantly being improved, the cost efficiency of the last option is constantly going down, and is becoming more and more popular.

The method by which scanned image of a hard copy becomes text entered on the computer is called optical character recognition (OCR). One big disadvantages of this method is inaccuracy. A lot of time and money have been spent on research so that computers can be able to recognize text as well as a human being [2].

Previous Work

There are several leading companies which are trying to develop a better OCR engine. Doculex is one of these companies. In addition to designing an OCR engine of their own and providing document conversion as a service, they are exploring a possibility of using two other OCR engines of their competitors to provide a better digital copy of document based on the results of the three OCR engines [1].

A first look at University of Central Florida at the associated statistical problem was completed by Mercedes McDonald where she showed that a voting scheme can be introduced based on the performance of the three OCR engines to improve overall accuracy [1]. Her calculations were based on a rather simple model. It was concluded that the accuracy of multi-engine OCR is better than the accuracy of each individual OCR engine. Mercedes ran OCR engines on the set of clean images of various fonts and styles. After examining errors made by those OCR engines, she classified them into several types. In order for her to determine whether multi-engine environment will be a success she focused on four types of errors shown in Table 1. Type 1 is the case when two words in some OCR processed engines were recognized as one word, and in another as two words. Even though, two engines recognized it as one word, if Doculex's OCR-It has greater accuracy when it comes to recognizing when one word ends and

another begins, the voting engine picks the correct Doculex's result. Type 2 error happens when a word had a dash and some engines ignored it or inserted a space after the dash. Given that Abbyy's FineReader has a greater accuracy recognizing dashes, the voting engine will again pick the correct answer. Type 3 error has to do with Omni's and Abbyy's misrecognizing comma and space delimited numbers. They often ignore the space. More commonly, when comma is between the digits as a part of formatting of a number, the voting engine cannot give more voting weight to Doculex's OCR-It; therefore, the wrong answer will be chosen. Type 4 error is considered to be the case when two out of three engines got the answer right and there is no special vote weighting for this particular case. In the benchmarks ran by McDonald, this category of errors resulted in correct voting.

Table 1
Examples of Errors Found in Multi-engine Environment

Types of Errors:	Type 1	Type 2	Type 3	Type 4 Only 1 engine displayed error
Original - Correct	chief elected	war-laden	April 3, 1959	
Omni	chiefelected	warladen	April 3,1959	
Doculex	chief elected	war- laden	April 3, 1959	
Abbyy	chiefelected	war-laden	April 3,1959	
Outcome w/ voting	CORRECT	CORRECT	INCORRECT	CORRECT

Mercedes created a look up table that connected the types of errors to the OCR engines and specific font sizes and styles. This look up table was used as a basis for a voting engine. Even though, she was not able to provide an approximation of how much more text can be

recovered correctly, she showed that it is possible to achieve greater accuracy with a multi-engine environment [3].

This algorithm was later realized in work done by Chris Sprague [3]. The multi OCR engine achieved 91.07% accuracy, which was 2.27% better than the most accurate single OCR engine of OmniPage Pro. Benchmarking the OCR engines individually versus a voting method for real-life documents, he achieved the results given in Table 2. In this table Sprague shows difference in inaccuracy percentage of each engine and inaccuracy percentage of multi-engine OCR voting systems for ten documents in columns one through three. The fourth column shows average of those differences. Some of the differences came out to be negative, which means that a particular engine performed better than the voting system applied to three of them. Overall, however, the voting scheme improved results by 10.03% [3].

Table 2
Accuracy Improvements with Voting

Accuracy Difference Table			Average Accuracy Difference
OCR-It	OmniPage	FineReader	
28.33%	2.12%	4.55%	11.67%
0.30%	0.00%	-1.19%	-0.30%
1.34%	0.67%	-0.17%	0.61%
2.22%	1.78%	0.89%	1.63%
36.84%	5.26%	52.63%	31.58%
12.15%	8.10%	3.64%	7.96%
16.84%	-0.51%	3.06%	6.46%
32.62%	4.23%	1.69%	12.85%
20.74%	0.74%	7.40%	9.63%
43.03%	0.31%	11.14%	18.16%
Average			10.03%

In developing the voting process, Sprague noted several issues could be addressed in further development of a multiengine OCR voting system. Some of the documents have hand writing, which is ignored by some OCR engines and somewhat processed by the others, and he also noted that some of the documents have characters that were broken apart as demonstrated in Figure 1 [3]. For example, in the word “amendment” of the first line of Figure 1, the letter “m” is broken up and might be potentially recognized as “n” and “i”. Also Letter “E” of word “Escrow” is broken up and could be misrecognized as “I” and symbol “;”. While the first

problem is hard to eliminate, the second problem can be partially solved using morphological filters.

2. AGREE that no amendment to these Escrow Instructions shall be of any effect until made in writing, signed by all parties and delivered to and accepted by the Escrow Agent. No notice or demand shall be of any effect unless made in writing, signed by the party making the notice or demand and presented to Escrow Agent. These Escrow Instructions and any amendments hereto, as provided above all constitute the entire agreement between the Escrow Agent and the parties hereto, notwithstanding the provisions of any purchase contract or other agreement between the parties, whether oral or written.

Figure 1: Example of Distortion due to Filters Built-in into Copiers

Problem Statement

Most OCR engines use some form of image processing to restore a document image. The goal of preprocessing is to remove noise, and separate actual text from images and other non-text elements that are not to be processed by OCR engine. The goal of this thesis is to try to improve the document images before it is applied to an OCR engine using morphological filters. Another aspect that will be dealt in this research is text alignment between the text outputs that are produced by each OCR engine. Before a multi-engine voting can take place, the voting engine needs to know where each word is located in each processed text document.

Proposed Solution

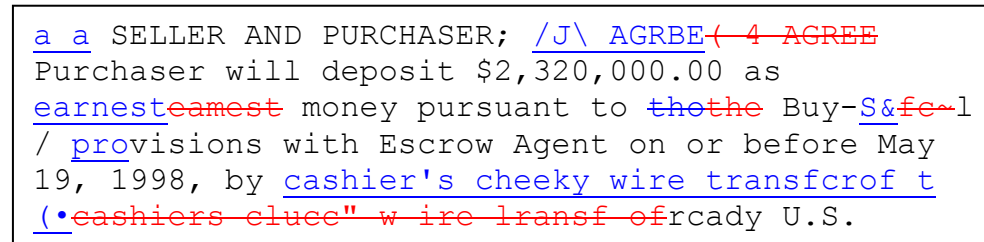
While Chris Sprague was implementing the voting system, several repeating problems with quality of scanned images were noticed. One of the major degradations has to do with the fact that the copies provided are not the copies of the original but the copies of the copies. Many

copiers now days have built in image filters. These filters can introduce breaks in the characters after making a copy of the copy many times [3]. Examples of such distortions are shown in Figure 1.

Distortion demonstrated in Figure 1 can be somewhat fixed using morphological filters, to be more specific a combination of erosion and dilation filters. These filters will be discussed further in Chapter 2. This technique is often used in preprocessing of finger prints. In case of additive noise, these filters can be used to reduce the noise and increase overall OCR accuracy.

While preprocessing can greatly improve the final OCR, outcome without knowing which word is where in each of the three documents, voting cannot take place at all. In previous work by Chris Sprague and Mercedes McDonald, the possibility of using already existing documents to compare functions were discussed. Linux users are probably familiar with DocDiff [1]. There are also programs available for windows such as Document Compare feature of Microsoft Word and DiffDoc by SoftInterface© [4]. Such programs are designed for the case when existing text was modified and not acquired from an alternate source. They seek out portions of text that did not match, given that some of the text will be exactly the same, which is not the case in this application. Applying change tracking software to OCR processed files would often result in flagging of large sections of text which makes voting impossible. Figure 2 shows Word's® 2000 results, where a large portion of the text at the end of the sample was marked as different. Since the voting system accepts single words only, it would be impossible to pick the correct answer. This kind of software is designed to keep track of changes not typos and errors, and has not proven to be efficient for this particular problem. Alignment of three texts produced by three OCR engines required more of a unique solution, which is designed to compare two or more texts that essentially are the same but have differences in letters with reasonable frequency.

Perhaps a visual comparison of the text would perform better than the plain ASCII comparison. Character degradation needs also be taken in consideration to allow the alignment engine to see where OCR programs could have made a mistake.



a a SELLER AND PURCHASER; /J\ AGRBE(-4-AGREE
Purchaser will deposit \$2,320,000.00 as
earne~~st~~~~eame~~st money pursuant to ~~the~~~~the~~ Buy-S&fe-l
/ provisions with Escrow Agent on or before May
19, 1998, by cashier's cheeky wire transfcrof t
(~~.cashiers-cluce" wire-transf-ofrcady~~ U.S.

Figure 2: Result of Change-tracking Feature of Microsoft® Word® 2000.

Sprague and Mercedes have developed means by three separate OCR engines can be brought together to form a multi-engine OCR system using voting scheme. While voting algorithm is an essential part of the system, the complete system in order to become autonomous and more reliable, image preprocessing and text aligning algorithms have to be developed. Chapter two of this thesis discusses the concepts of morphological filters. Erosion and dilation are discussed in great detail since these filters are the basis of document restoration. Chapter three discusses ways to align words of the three documents that are produced by three OCR engines after scanning the same page. Chapter four summarizes the content and discusses possible future work.

CHAPTER TWO: MORPHOLOGICAL IMAGE PROCESSING

Introduction to Morphological Image Filters

Morphological operators come from set theory [5]. The two basic morphological filters are erosion and dilation, and the most common applications of these filters involve binary images. In image processing the erosion filter tends to reduce size of bright spots, while dilation tends to do the inverse. Figure 3.a gives an example of dilation and Figure 3.b gives an example of erosion. It is commonly considered that the object that is being dilated or eroded consists of high values (white). For the problem covered in this chapter black ink will be considered a high value or object value. In Figure 3.a, light rectangle is the object that being dilated and the dark rounded rectangle around it is what was dilated onto the original object. The circle (in this particular example) is called the structuring element or sometimes in image processing referred to as the mask. In this particular case, it is solid and shown hallow for demonstration purposes. Basically, structuring element defines what shapes of the object will be preserved and what shapes of the object will be discarded. Dilation can be expressed with equation 2.1. It basically means that set A dilated by set B is a union of sets $A + b$ where b is an element of all elements of set A with all elements of set B .

$$\text{Dilation of } A \text{ by } B = A \oplus B = \{\cup(A+b) | b \in B\} \quad (2.1)$$

where:

A is an object that is dilated,

B is a structuring element, and

b is an element of B

Erosion is expressed by the equation 2.2. This equation implies that the set A eroded by set B is composed of elements p such that all of the elements from set B added to p belong to set A.

$$\text{Erosion of } A \text{ by } B = A \ominus B = \{p | (B+p) \subseteq A\} \quad (2.2)$$

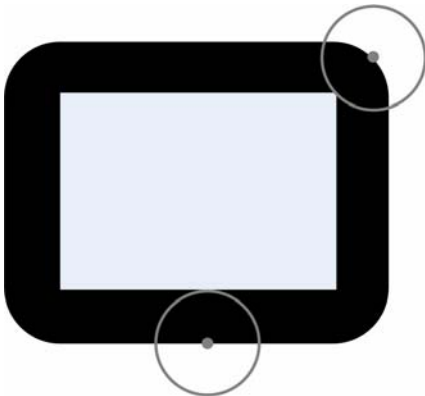
where:

A is an object that is eroded,

B is a structuring element, and

p is an element of $A \ominus B$

a)



b)

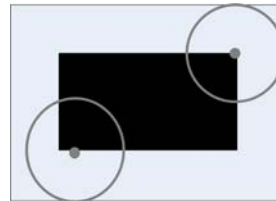


Figure 3: a) Example of Dilation Filter, b) Example of Erosion Filter

Dilation filter in case of a circle can be thought of as a wheel that is being rolled on the perimeter outside of the object. The center of the wheel, then, describes the perimeter of the dilated object. Erosion, on the other hand would be similar to rolling the wheel on the perimeter inside the object.

The opening filter is erosion followed by dilation. This filter can be used to remove small specs and noise that are small enough to fit inside the structuring. An opening filter is demonstrated in Figure 4 and expressed as

$$A \circ B = (A \ominus B) \oplus B \quad (2.3)$$

A closing filter, on the other hand, connects elements that are closer than the size of the mask and removes the hole that can fit inside the mask. Results a of closing filter are demonstrated in Figure 5 and expressed by as

$$A \bullet B = (A \oplus B) \ominus B \quad (2.4)$$

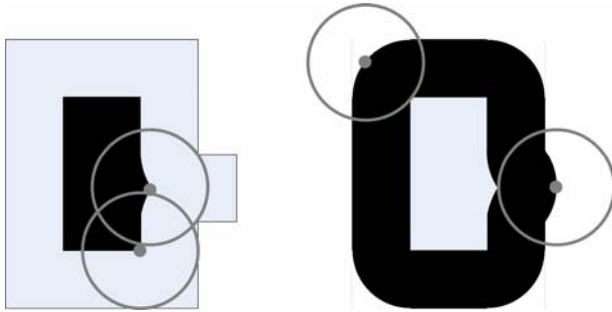


Figure 4: Example of Opening Filter (Erosion Followed by Dilation)

The original object is the light grey portion of the left image in Figure 4. The dilated object then is formed by tracing the circular structuring element about the inside of object shown on the left of Figure 4. As the circle is rotated about object shown in gray it traces a new contour as shown in black. Similarly, erosion of the dilated object is shown on the right of Figure 4, this type the circular structuring element traces the eroded object outside. Because the structuring element cannot fit into the bump on the object, the bump will be smoothed out. After applying the dilation filter, since most of the information about the bump was discarded, it will come back as a significantly smaller bump.

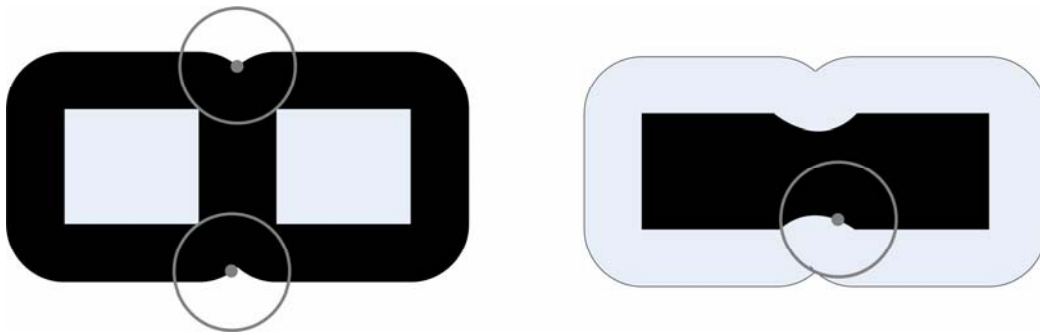


Figure 5: Example of Closing Filter (Dilation Followed by Erosion)

In the case of a closing filter, the structuring element is too big for the gap between the two original objects shown in light grey. This will cause the structuring element to cover a larger area. Since the information about the gap has now been lost, after applying the erosion filter, the two objects will now be connected to each other.



Figure 6: a) Broken Character; b) Restored Character by Closing Filter

Figure 6 demonstrates how a closing filter can connect broken apart characters. The size of the structuring element has to be bigger than the gap. However, if the structuring element is too big, the closing filter can not only distort the character but also make it completely unrecognizable. In this particular example the gap was 5 pixels wide, and a 7x7 round mask was applied. Even

with this small size of the mask, there is slight distortion that can be seen in the upper right corner inside the letter.

Implementation of Morphological Filters for Binary Images

Discussed in this section, is one of the simpler ways to implement erosion and dilation filters for binary images.

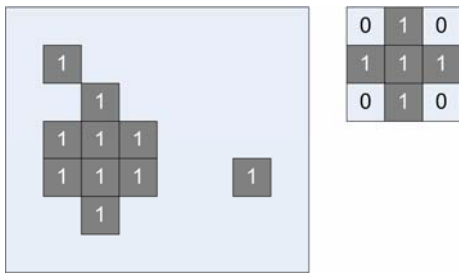


Figure 7: Binary Image and Structuring Element

Figure 7 shows a binary image on the left that will be subjected to erosion and dilation filters of structuring element shown in the same figure on the right.

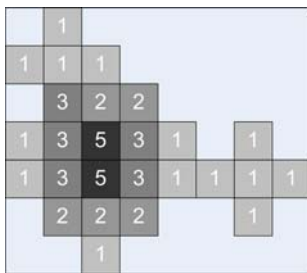


Figure 8: Summation of Overlapping Pixels

Figure 8 shows an intermediate state of applying the two morphological filters to the original image. The structuring element is then moved across the image. For each placement of structuring element a sum of values of pixels that are covered by the structuring element is

computed. Since the structuring element consists of 5 pixels, the maximum sum would be five and the minimum sum would be zero.

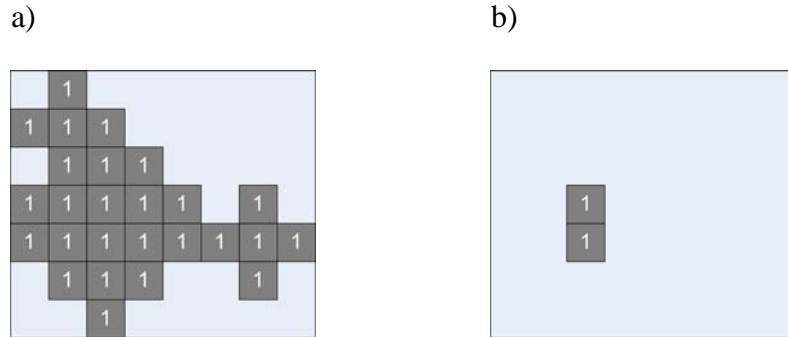


Figure 9: a) Dilation, and b) Erosion

After the sums have been computed a threshold is applied to the set of sums that will produce the binary image. If the filter is dilation the threshold is always zero, if the filter is erosion the threshold is the maximum possible sum, which in this case is five. From Figure 9.a it can be concluded that the output image will take on the new features similar to the shape of the structuring element.

Morphological Filters And Image Restoration

In order to establish where to start with examining of morphological filters and their effects on images with various fonts, sizes, and added noises, test image was created. This image contained by printing to an image at 300dpi (a value recommended by developers of most OCR engines) of a document that contained several types of fonts, such as Time New Romans and Courier New, sizes, such as 8pt, 10pt, 12pt, and 14pt, of both normal and bold weights, and italicized style. This image was subjected to impulse (salt and pepper) noise of probability of 0.005 for both salt and pepper and a custom made filter that introduced horizontal gaps in

characters of width half the thickness of characters of 12pt size as shown in Figure 10 and Figure

11.

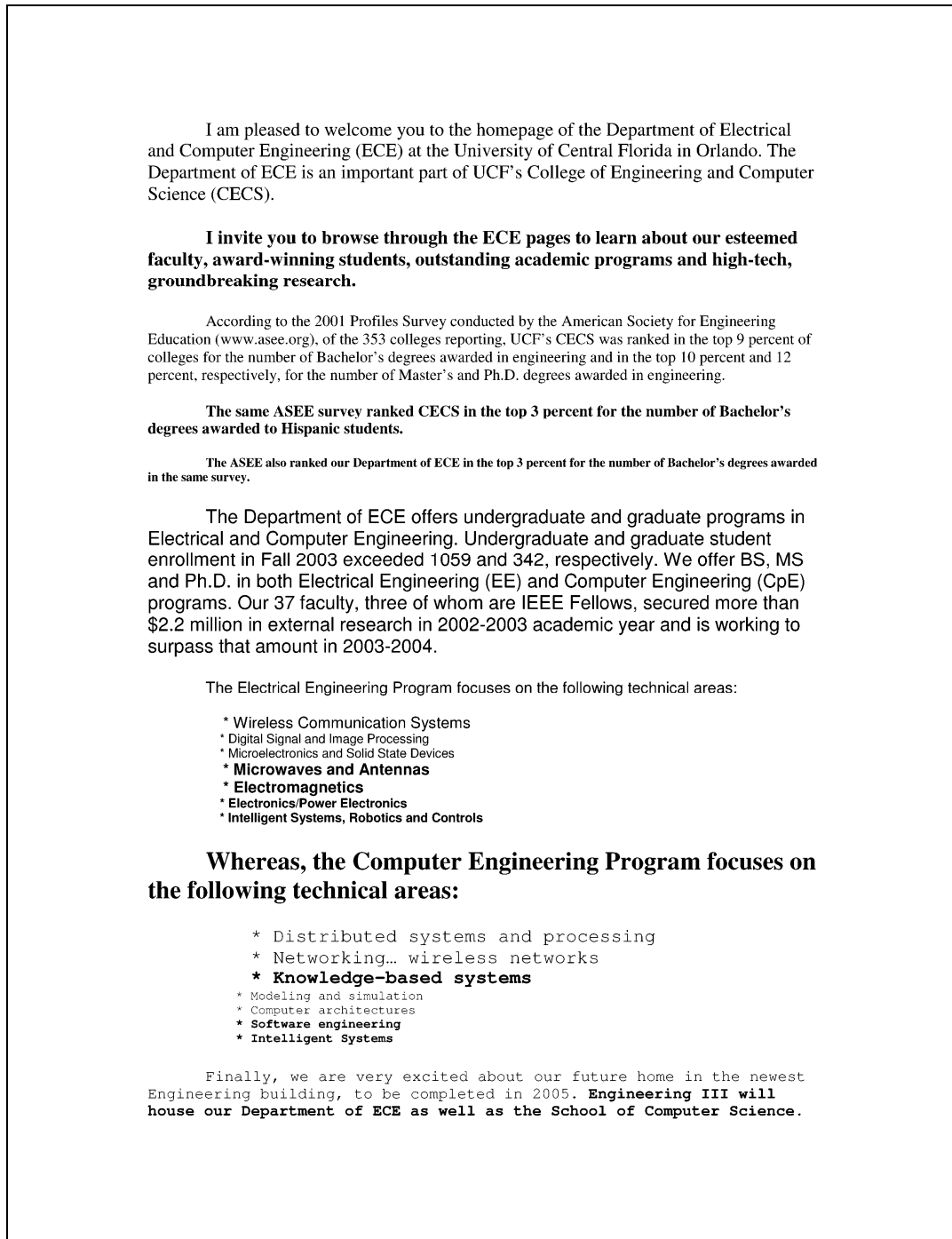


Figure 10: Image that was used for benchmarking the performance of median and opening and closing morphological filters.

Opening, closing, and median filters were then applied to those images.

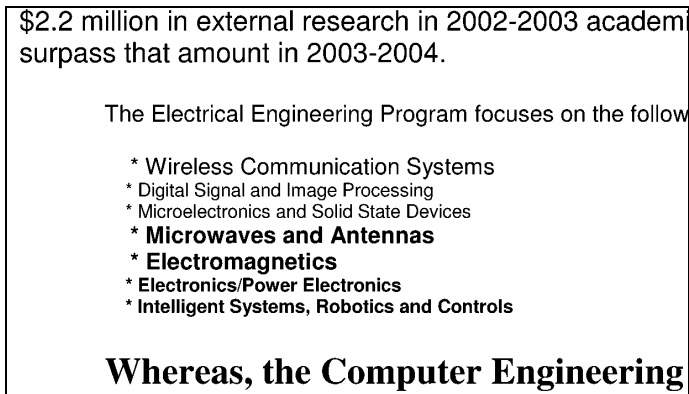
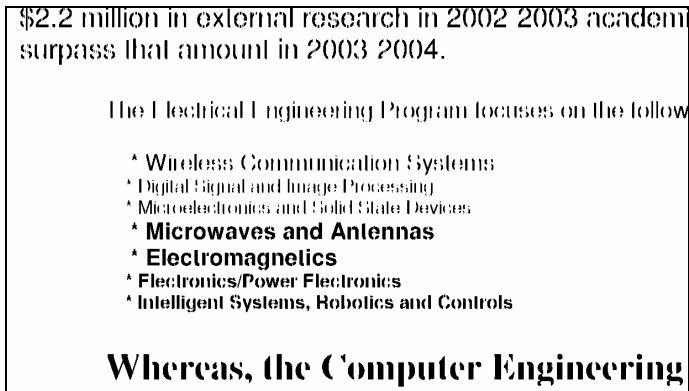


Figure 11: Portion of Image of Figure 10 That Will Be Used to Examine Various Noises and Effects of Filters

a)



b)

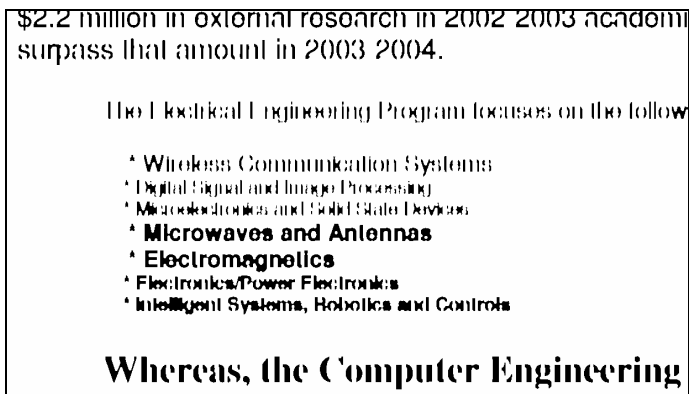
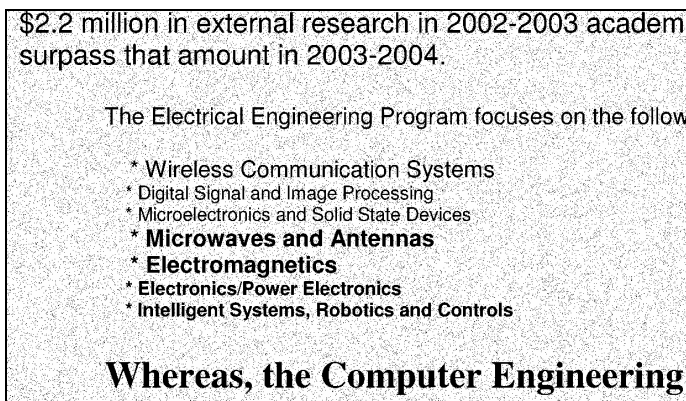


Figure 12: a) Image with Custom Noise, b) Applied to It Closing Filter 5x5

In Figure 12 closing filter was able to somewhat restore the gaps produced by custom filter noise. However, that was the case only for smaller fonts. In case when the font was too small, the closing filter actually connected the parts of the characters that were not supposed to be connected.

a)



b)

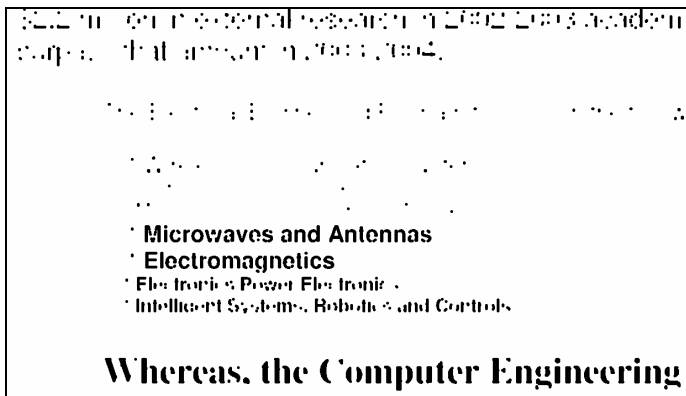
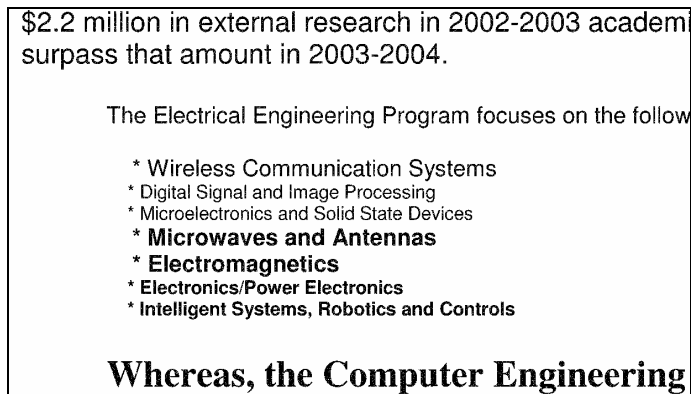


Figure 13: a) Image with Pepper Noise of probability 0.05, b) Applied to It Opening Filter 5x5

Opening filter successfully removed the of pepper noise in the Figure 13 only where the page was supposed to be blank. However, it has done significant amount of damage to the actual

text as can be seen in the text with the smaller font. By the preserved text, one can note that size of the mask of opening filter must be smaller than the features of the text present in the image.

a)



b)

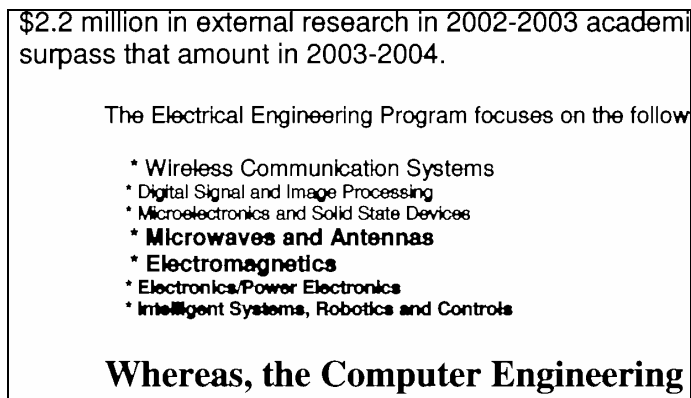
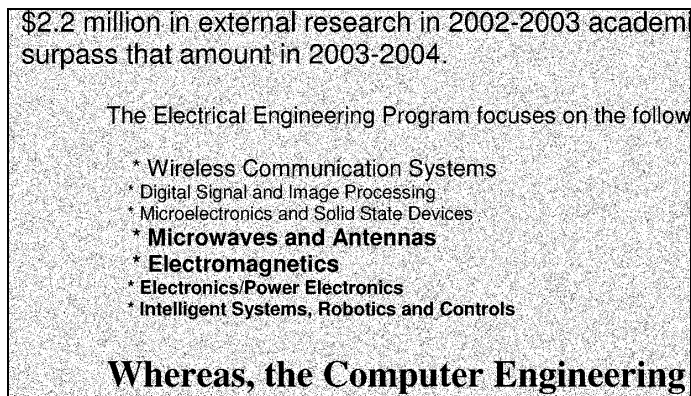


Figure 14: a) Image with Salt Noise of probability 0.05, b) Applied to It Closing Filter 5x5

In case of Figure 14, the closing filter was able to successfully restore the text. It can be noted that in the case of smaller text, where gap between the characters and features within characters was smaller than the size of the structuring element used, the filter connected non-desired sections of text.

a)



b)

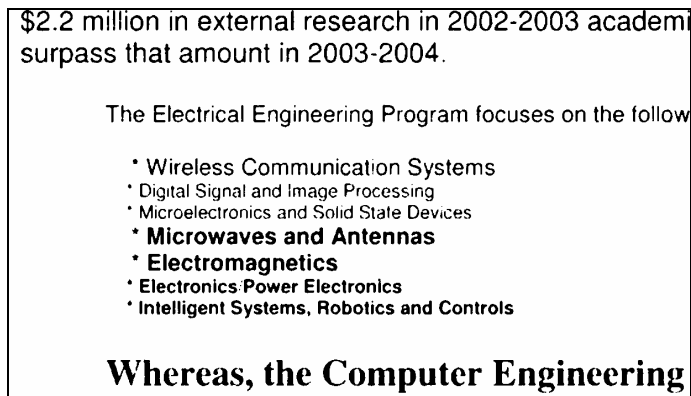
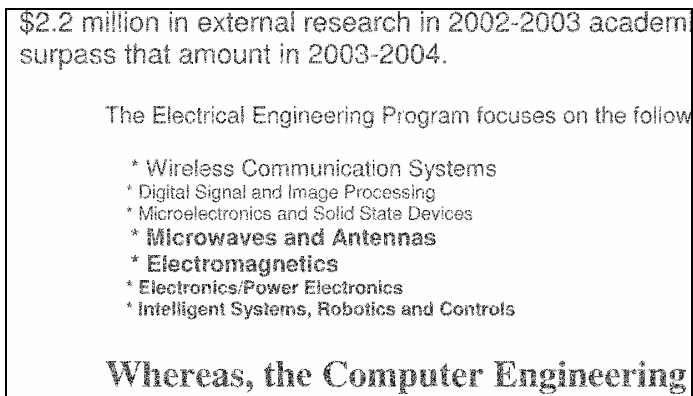


Figure 15: a) Image with Salt and Pepper Noise of probabilities 0.10 each, b) Applied to It Median Filter 5x5

The median filter was successful in removing pepper noise and restoring the pixels. However, this particular filter is known to work great on impulse noise. While it might not be suitable for this particular application, its numerous modifications must be kept in mind.

a)



b)

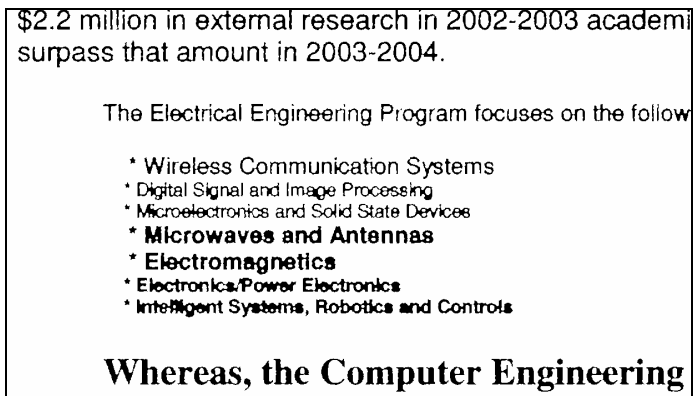


Figure 16: a) Image with Salt Noise of probabilities 0.4 each, b) Applied to It Closing Filter 5x5

Similarly to results of Figure 14, in case of Figure 16, the closing filter restored most of the text, and only added a slight distortion to the smaller text.

Since the closing filter delivered the most remarkable results, it was then examined more closely using additional types of noises listed in Table 3.

Table 3
Abbreviations and Descriptions of Noises Used

Abbreviation	Description
cus	Custom filter that generates gaps in horizontal features of characters comparable to the thickness of characters of 12pt font scanned at 300dpi
ds01	Dust and Spec filter of Adobe Photoshop® 6.0 with mask size 1
ds03	Dust and Spec filter of Adobe Photoshop® 6.0 with mask size 3
sp005005	Salt and pepper noise with probabilities 0.005

Table 4
First Set of Images Processed by Abbyy

Image	Suspect characters	Total characters
cus	451	2610
cus_out	910	2691
ds01	32	2316
ds01_out	507	2382
ds03	702	2335
ds03_out	813	2419
original	7	2306
original_out	676	2412
sp005005	12	2306
sp005005_out	916	2404

Note: File names with postfix "out" are processed with closing filter.

Table 5
 First Set of Images Processed by Omni

<u>Image</u>	<u>Suspect words</u>	<u>Total words</u>
cus	187	390
cus_out	184	358
ds01	14	371
ds01_out	119	356
ds02	104	333
ds02_out	136	383
ds03	134	324
ds03_out	157	395
original	1	354
original_out	126	350
sp005005	15	355
<u>sp005005_out</u>	<u>183</u>	<u>361</u>

Note: File names with postfix "out" are processed with closing filter.

Square mask 3x3 was used in erosion and dilation filters. According to results of Table 4 and Table 5, only Omni was able to perform better after applying opening and closing filters in the case of noise from the custom filter which simulated the effect of characters breaking up after the page has been copied over and over again.

Based on the previous results the following changes to the experiment can be done: the noise model can be improved since it has been shown not to be very effective, the shape of the mask of morphological filter can be changed to preserve more rounded features of the font, and

an additional filter can be used, such as median filter, which is also a morphological filter in case of binary images, to eliminate salt and pepper (impulse, in general) noise. Opening/closing filter did not perform well in the case of pepper noise. Closing/opening filter; on the other hand did not perform well in the presence of salt noise. This has limited the conclusion that could be derived from this experiment. Even though; presence of various font sizes and styles might have seemed like a good idea, having this variety hurt statistical outputs of OCR engines.

While these experiments were not systematic, they helped to establish several important things about closing and opening filters and their application. Effects of these filters greatly depend on the size and the shape of structuring element. Also, selection of size of structuring element must be done accordingly to the font size and resolution of the image. From these conducted experiments it can also be concluded that the noise model that needs to be considered is salt noise of various densities.

Benchmarking Performance of Optical Character Recognition Engines

Two out of three OCR engines (Abbyy's and Omni's) provide qualitative description of the output that they produce. Abbyy's OCR engine as a feed back provides number of words found and number of suspect words. Suspect words are those words that have one or more characters that the OCR engine was not able to recognize exactly, meaning that the OCR engine had several characters in its database that looked similar to what it saw in the image. Omni's OCR engine provides the number of suspect characters. While it is hard to connect the two values together its possible to relate them for a known fragment of the text. In benchmarking the performance of morphological filters the same text file will be used. Since the

average length of words and total number of words is known, it can be easily correlated which values of suspect characters correspond to suspect words. By creating a two-dimensional array filled with zeros of height equal to number of words and width equal to the average number of characters in the word and setting cells at random with value one will provide the relationship between the number of suspect characters (number of ones introduced) and number of suspect words (number of rows that have at least one cell with value one).

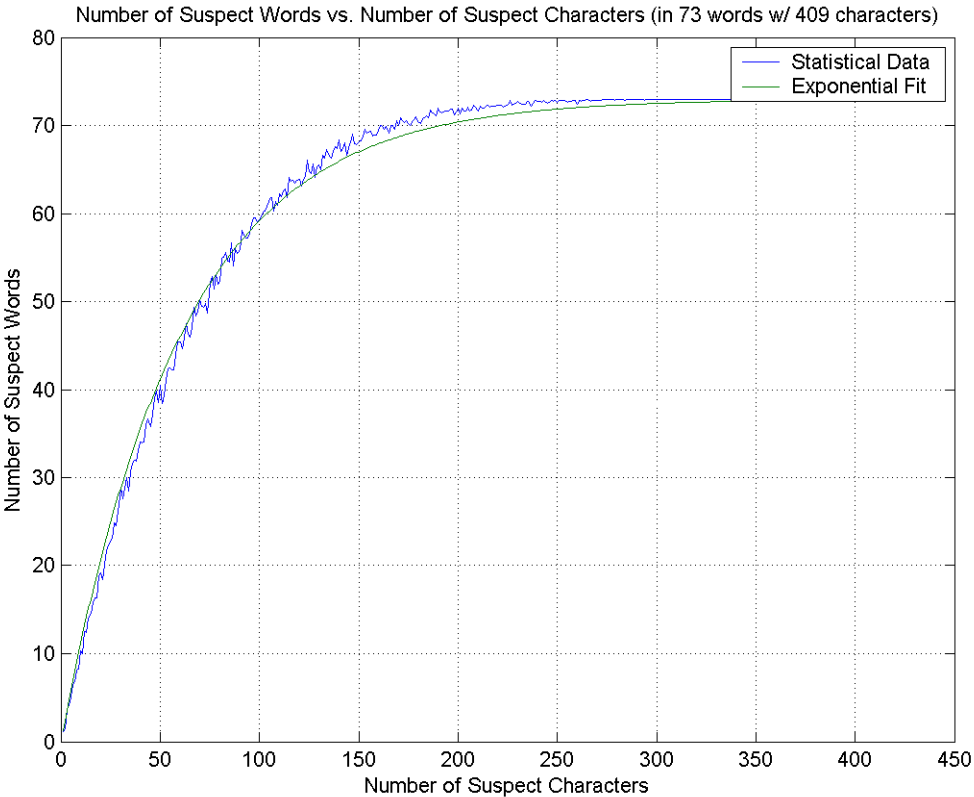


Figure 17: Simulation of Suspect Characters and Words and Its Exponential Fit

Figure 17 shows the simulated data and its exponential fit. At first, when the number of suspect characters is low, the probability that one word will have two of them is very low; therefore, it behaves as a linear function in the beginning, which means that for the most suspect

character will result in one suspect word. When the number of suspect characters is high, the probability that each word has suspect character is very high; therefore, at a certain point the number of suspect characters stops increasing and approaches an asymptote. The exponential fit is described by equation 2.5.

$$w = W \cdot \left(1 - e^{-\frac{n}{60}} \right) \quad (2.5)$$

Where:

W – total number of words in the text,

w – number of suspect words, and

n – number of suspect characters.

Value 60 was determined experimentally; however, it most likely is dependant on W and N (total number of characters in the text). Equation 2.6, is the inverse to equation 2.5, shows how to obtain the number of suspect characters knowing the number of suspect words.

$$n = -60 \cdot \ln \left(1 - \frac{w}{W} \right) \quad (2.6)$$

Relating Sizes and Shapes of Structuring Elements of Morphological Filters to Font Sizes and Types of Noise

As it was noted earlier, the size of the masks used needs to agree with the font size of the text and resolution at which it was scanned. It is recommended by all three developers of OCR

engines used in this thesis that the documents are scanned at 300dpi. This is the resolution that will be used throughout this chapter of the thesis.

To be able to relate the results and draw conclusions, the same text was used. Eleven images were created that would have the same text with eleven different font sizes and constant font size within each image. Fonts used are 10pt, 11pt, 12pt, 13pt, 14pt, 15pt, 16pt, 17pt, 18pt, 19pt, and 20pt.

The Department of ECE offers undergraduate and graduate programs in Electrical and Computer Engineering. Undergraduate and graduate student enrollment in Fall 2003 exceeded 1059 and 342, respectively. We offer BS, MS and Ph.D. in both Electrical Engineering (EE) and Computer Engineering (CpE) programs. Our 37 faculty, three of whom are IEEE Fellows, secured more than \$2.2 million in external research in 2002-2003 academic year and is working to surpass that amount in 2003-2004
--

Figure 18: One of the Eleven Images of Text of Font Size 10pt in This Case Scanned at 300dpi That Is Used Throughout This Section

First thing that needs to be determined is the relation ship between the font size and the mask size such that the clean text does not get distorted. To each of the 11 images opening and closing morphological filters of mask sizes 3x3, 5x5, 7x7, 9x9, and 11x11 were applied. A circular shape was used for all of the masks. The outputs of this procedure were 110 images. One way to describe the effect of the filter on the image is to calculate the difference of the two images. Since images have different font sizes, in order to be able to compare the results, a ratio of image difference and number of black pixels will be considered as image degradation value. In case of no degradation this value will be zero, and will increase with the level of degradation.

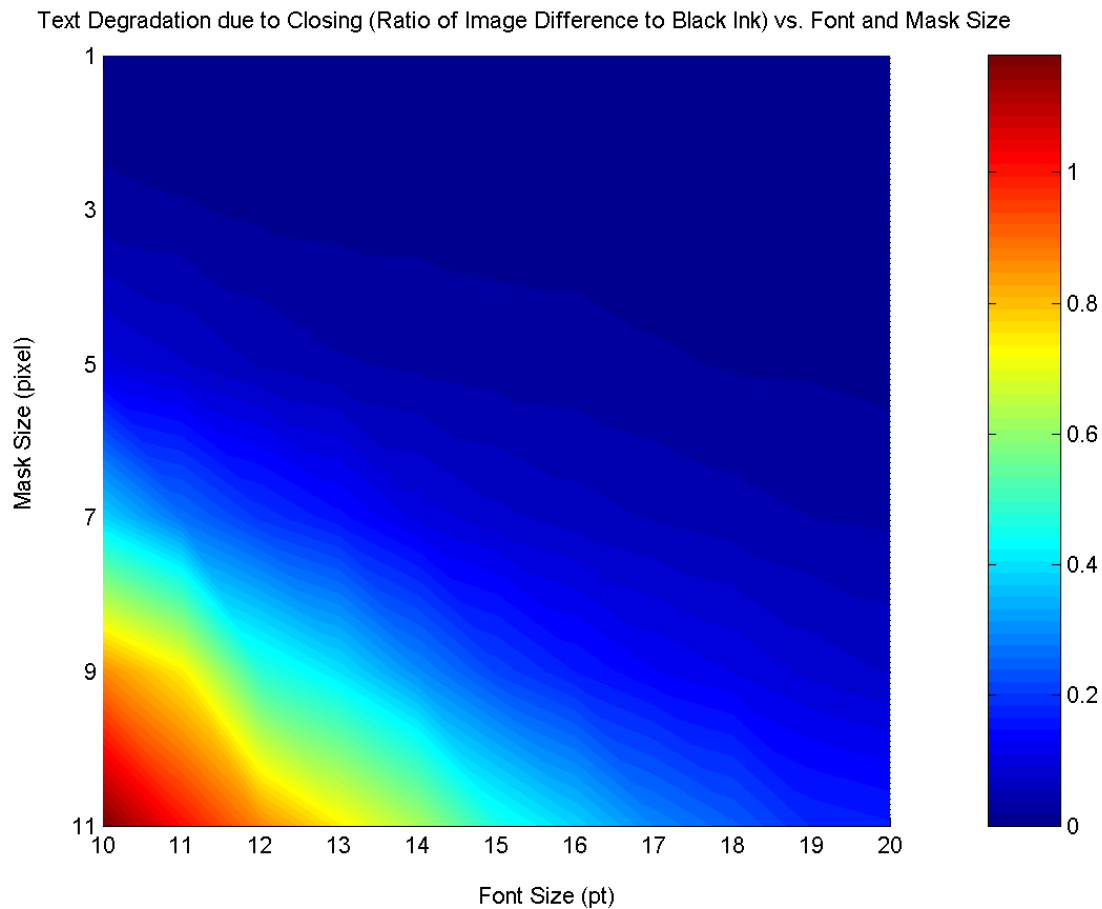


Figure 19: Ratio of Image Difference before and after Application of Closing Filter versus Mask and Font Sizes

Figure 19 shows a graph of image degradation of images versus font size and size of the applied filters. Since mask sizes of interest are the ones that affect the image but not too much, from this particular results in can be concluded that the mask size and font size can be related by equation 2.7.

$$MaskSize = 0.6 \cdot FontSize - 1 \quad (2.7)$$

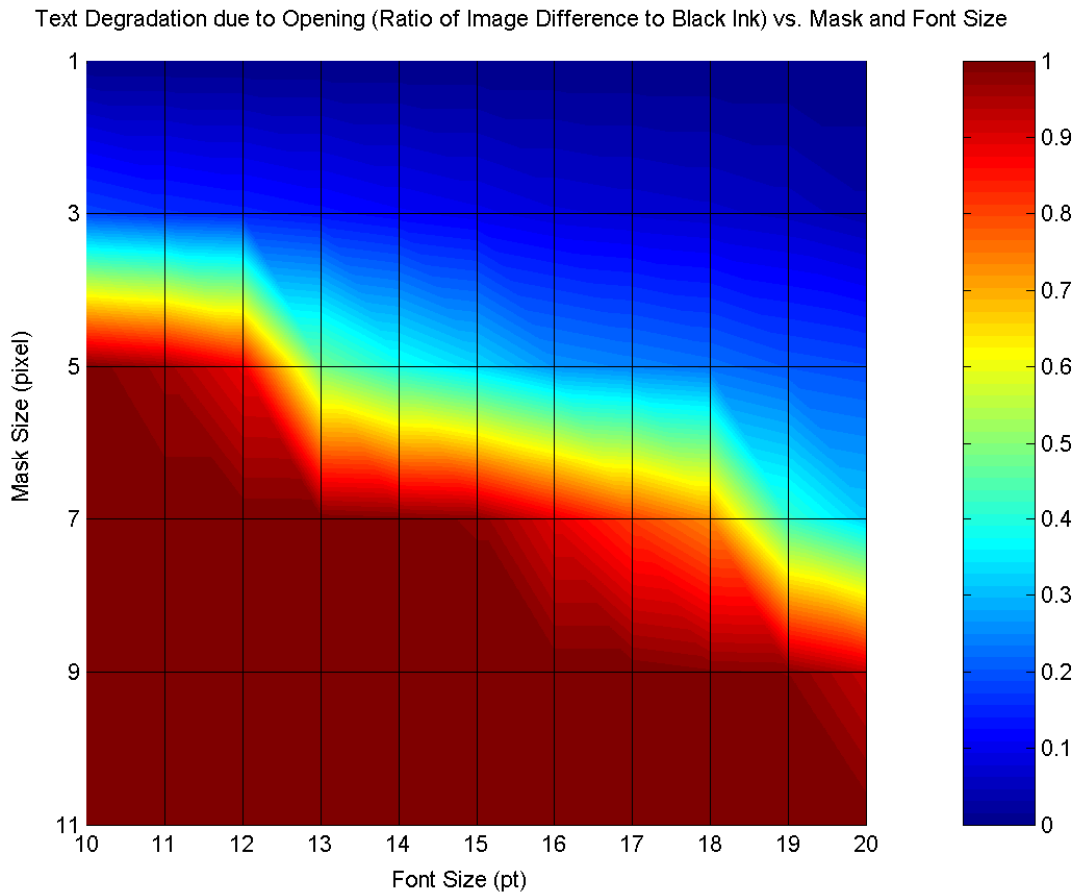


Figure 20: Ratio of Image Difference before and after Application of Opening Filter versus Mask and Font Sizes

Figure 20 shows a graph of image degradation of images versus font size and size of the applied opening filters. Unlike a closing filter, distortion due to an opening filter occurs at significantly small mask sizes. Relationship of mask size to font size is described by equation 2.8.

$$MaskSize = 0.2 \cdot FontSize + 1 \quad (2.8)$$

This suggests that in case of salt and pepper noise or any noise that introduces gaps opening filter will distort an image beyond recognition. The only time this filter can be used

successfully is when only pepper noise is present. Since this thesis deals with restoration of characters by eliminating gaps, this filter is beyond the scope of the thesis.

The focus now falls on the closing filters. Next step in analysis of effects of closing filter on image is to look at amount of characters recognized by OCR engines and number of characters flagged as suspect characters by these engines. The 55 images, acquired by applying a closing filter of five different mask sizes to images of eleven different font sizes, have then been processed by Abby's and Omni's OCR engines. Abby's reported suspect word values have then been converted to a number of suspect characters. For each of the images, each OCR engine returns number of characters and number of suspect characters. To represent these two values as a single graph their differences were taken. This difference represents number of character that each OCR engine claims to have recognized correctly. This value will be referred to as number of good characters. In the ideal case, when all characters were recognized and none of them were marked as suspect characters the output would be 475 characters (409 not counting white spaces).

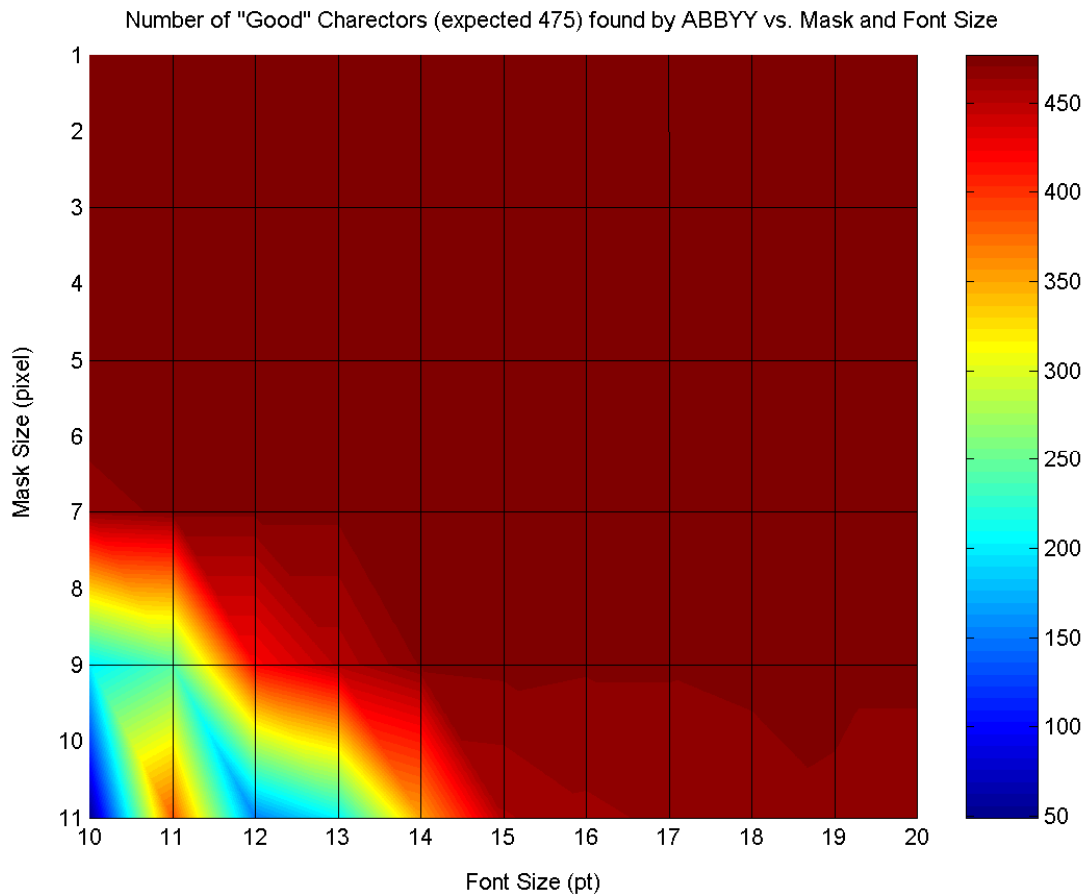


Figure 21: Number of Good Characters Recognized by Abby’s OCR Engine versus Mask and Font Sizes

Figure 21 shows the number of good characters found by ABBYY’s OCR engine. In this case higher values are better. The distribution of results follows closely the once predicted in Figure 19. In fact the relationship between the mask size and the font size according to results of ABBYY’s OCR engine also follow equation 2.7. Selecting mask sizes according to this equation will correspond to the point of the graph where number of good characters is maximum.

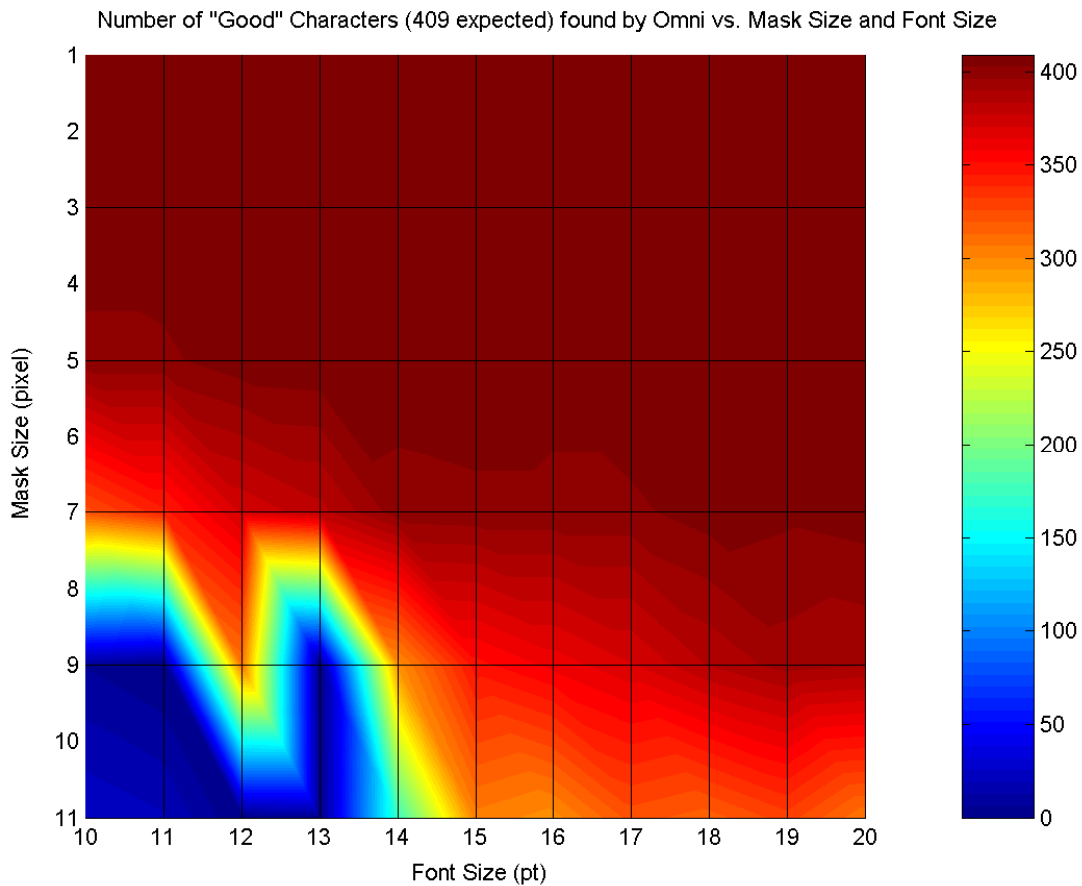


Figure 22: Number of Good Characters Recognized by Omni's OCR Engine versus Mask and Font Sizes

Figure 22 shows the number of good characters found by Omni's OCR engine. This distribution of results also follows closely the once predicted in Figure 19, however slightly skewed. In fact the relationship between the mask size and the font size according to results of ABBYY's OCR engine also is better described by equation 2.9. Selecting mask sizes according to this equation will correspond to the point of the graph where number of good characters is maximum.

$$MaskSize = 0.4 \cdot FontSize + 1 \quad (2.9)$$

Equations 2.6, 2.7, 2.8, and 2.9 provide maximum size of the mask that can be used for a certain font size. It does not mean that the smaller mask cannot be chosen.

Next step is to determine whether the shape of the mask is important or not, and if it is, which shape will lead to better results.

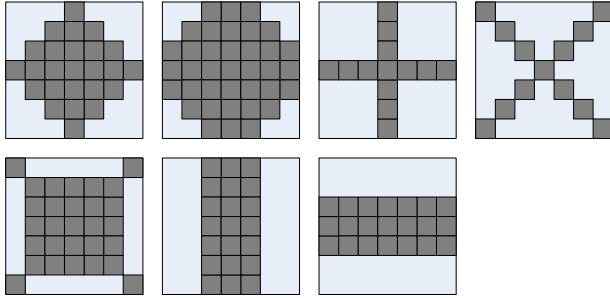


Figure 23: Seven Shapes of Masks of Size 7x7

Shapes that were chosen for next series of experiments are shown in Figure 23.

According to equation 2.9 the closing filter that used these shapes was applied to the text image file that had text with a font size 15pt.

Table 6
Results of Application of Closing Filter and ABBYY's OCR Processing

Shape	Image Difference Ratio	Suspect Chars	Total Chars	Good Chars
Circle	0.104	0	475	475
Vertical cross	0.14	0	475	475
Diagonal cross	0.252	13	475	462
Diamond	0.124	0	475	475
Diagonal cross w/ circ.	0.243	6	473	467
Vertical bar	0.045	0	475	475
Horizontal bar	0.128	9	475	466

Table 7
Results of Application of Closing Filter and Omni's OCR Processing

Shape	Image Difference Ratio	Suspect Chars	Total Chars	Good Chars
Original	0	0	409	409
Circle	0.104	9	409	400
Vertical cross	0.14	16	409	393
Diagonal cross	0.252	64	377	313
Diamond	0.124	6	409	403
Diagonal cross w/ circ.	0.243	29	420	391
Vertical bar	0.045	2	409	407
Horizontal bar	0.128	51	352	301

Filters that distorted the image the least was the filter with masks shaped like a vertical bar. The most damaging filters turned out to be the two filters shaped like diagonal cross and diagonal cross with a circle. It can be concluded that the filters do less damage if their shapes resemble features common to characters. Since this is the case, it means that horizontal bar could potentially restore breaks in characters without distorting their vertical features. However, according to results of OCR processing of images there was a significant damage done by the filters with masks shaped like horizontal bar. Perhaps a shape that resembles areas of characters that are broken would deliver better results.

Next step is to determine the effects of closing filter on various noises. The simplest noise model is salt noise. Eleven pictures of different font sizes were subjected to salt noise of 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, and 0.5 salt densities. Each of these images were then processed

by closing filters with circular shapes of sizes 3x3, 5x5, 7x7, 9x9, and 11x11. As a result, 385 images were generated. Since results of Omni's and ABBYY's OCR engines are loosely correlated, the images were processed only by ABBYY's OCR engine. The results were stored in 3D array, which is hard to represent graphically or in a table. Since at this point any improvement is of an interest, instead of using values, one could use an indication that there is or there is not any improvement. The results are shown in Table 8.

Table 8
Results of ABBYY's OCR Processing of 375 Images with Various Font Sizes, Noise Densities, and Filtered by Closing Filter of Various Mask Sizes

Salt Density	Mask Size	Font Size											
		10pt	11pt	12pt	13pt	14pt	15pt	16pt	17pt	18pt	19pt	20pt	
0.005	3x3	-	-	-	-	-	-	-	-	-	-	-	-
	5x5	-	-	-	-	-	-	-	-	-	-	-	-
	7x7	-	-	-	-	-	-	-	-	-	-	-	-
	9x9	-	-	-	-	-	-	-	-	-	-	-	-
	11x11	-	-	-	-	-	-	-	-	-	-	-	-
0.01	3x3	-	-	-	-	-	-	-	-	-	-	-	-
	5x5	-	-	-	-	-	-	-	-	-	-	-	-
	7x7	-	-	-	-	-	-	-	-	-	-	-	-
	9x9	-	-	-	-	-	-	-	-	-	-	-	-
	11x11	-	-	-	-	-	-	-	-	-	-	-	-
0.02	3x3	-	-	-	-	-	-	-	-	-	-	-	-
	5x5	-	-	-	-	-	-	-	-	-	-	-	-
	7x7	-	-	-	-	-	-	-	-	-	-	-	-
	9x9	-	-	-	-	-	-	-	-	-	-	-	-
	11x11	-	-	-	-	-	-	-	-	-	-	-	-
0.05	3x3	-	-	-	-	-	-	-	-	-	-	-	-
	5x5	-	-	-	-	-	-	-	-	-	-	-	-
	7x7	-	-	-	-	-	-	-	-	-	-	-	-
	9x9	-	-	-	-	-	-	-	-	-	-	-	-
	11x11	-	-	-	-	-	-	-	-	-	-	-	-
0.1	3x3	-	-	-	-	-	-	-	-	-	-	-	-
	5x5	-	-	-	-	-	-	-	-	-	-	-	-
	7x7	-	-	-	-	-	-	-	-	-	-	-	-
	9x9	-	-	-	-	-	-	-	-	-	-	-	-
	11x11	-	-	-	-	-	-	-	-	-	-	-	-
0.2	3x3	x	-	-	-	x	x	-	x	x	x	x	x
	5x5	x	-	-	-	x	x	x	x	x	x	x	x

Table 8

Results of ABBYY’s OCR Processing of 375 Images with Various Font Sizes, Noise Densities, and Filtered by Closing Filter of Various Mask Sizes

Salt Density	Mask Size	Font Size										
		10pt	11pt	12pt	13pt	14pt	15pt	16pt	17pt	18pt	19pt	20pt
	7x7	-	-	-	-	x	x	x	x	x	x	x
	9x9	-	-	-	-	-	x	-	x	x	x	x
	11x11	-	-	-	-	-	-	-	-	-	x	-
0.5	3x3	x	x	x	x	x	x	x	x	x	x	x
	5x5	x	x	x	x	x	x	x	x	x	x	x
	7x7	-	x	x	x	x	x	x	x	x	x	x
	9x9	-	-	-	x	x	x	x	x	x	x	x
	11x11	-	-	-	-	-	-	-	x	x	x	x

Note: “-“indicates that there was no improvement in output of OCR engine after application of closing filter compared to OCR results of image with noise, “x” indicates that there was at least some improvement

With pepper noise density lower than 0.01 OCR engines did not have any problems recognizing the text. However, for salt densities 0.2 and 0.5, number of recognized characters went down and number of suspect characters went up for non filtered images. Relationship between font size and the mask size still follows the same pattern which is described by equations 2.8 and 2.9.

Results shown in Table 8, Table 7, Table 6, Figure 21, and Figure 22 suggest that irrelevant of noise the best choice for mask size is dependent on the size of the text. In order to proceed with exploring of possibilities of implementing morphological filters into a multi-engine OCR system, a method which allows detection font size of the text in a particular region of an image is needed, which is beyond the scope of this thesis.

CHAPTER THREE: TEXT ALIGNMENT

Introduction to Text Alignment

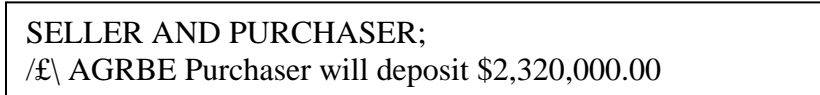
Once the document is processed by three OCR engines and before voting can take place, text needs to be aligned. An alignment of two or more text documents that were generated by two or more OCR engines is defined as a process of matching words that came from the same section of the image of that word among the text documents acquired by OCR engines.

a)

The image shows a scanned document snippet with a black border. The text is in a bold, serif font. The first line reads "SELLER AND PURCHASER:". The second line starts with a circled number "1." followed by "AGREE Purchaser will deposit \$2,320,000.00". There is a small, illegible mark to the left of the "1.".


SELLER AND PURCHASER:
1. AGREE Purchaser will deposit \$2,320,000.00

b)

The image shows the OCR result from FineReader, enclosed in a black border. The text is clean and matches the original document's content.

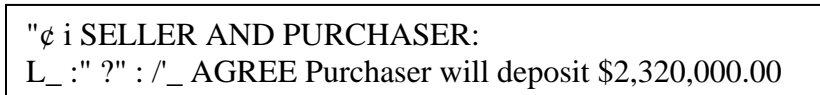
SELLER AND PURCHASER;
/£\ AGRBE Purchaser will deposit \$2,320,000.00

c)

The image shows the OCR result from OmniPage, enclosed in a black border. The text is clean and matches the original document's content.

a a SELLER AND PURCHASER;
(4 AGREE Purchaser will deposit \$2,320,000.00

d)

The image shows the OCR result from Expervision, enclosed in a black border. The text is clean and matches the original document's content.

"ç i SELLER AND PURCHASER:
L_ : " ? " : / _ AGREE Purchaser will deposit \$2,320,000.00

Figure 24: a) Image, b) Result of FineReader, c) Result of OmniPage, d) Result of Expervision

Results of an OCR engine is a single text file. A voting engine requires a single word from each of the documents as an input. If the only problem in OCR processing would have been misrecognized characters, one could separate words using space as a delimiter and feed that

into the voting engine. Unfortunately, as demonstrated in Figure 24, in addition to misrecognized characters there are also parts of the graphics or noise that were not filtered out and recognized as group of symbols. Also some OCR engines might insert an extra space between two characters of the same word, or opposite of that, recognize two separate words as one.

Table 9
Result of Aligning Text Using Space as a Delimiter

<u>OmniPage</u>	<u>FineReader</u>	<u>Expervision</u>
SELLER	a	"ç
AND	a	i
PURCHASER;	SELLER	SELLER
/£\	AND	AND
AGRBE	PURCHASER;	PURCHASER:
Purchaser	(L_
will	4	:"
deposit	AGREE	?"
\$2,320,000.00	Purchaser	:
	will	/'_
	deposit	AGREE
	\$2,320,000.00	Purchaser
		will
		deposit
		\$2,320,000.00.

Because of that, simply using spaces as word separators is not possible. Table 9 demonstrates the input that goes into the voting algorithm that was produced by splitting text into words using space character as a delimiter without any kind of alignment. Multi-engine OCR system will only be able to correctly vote on the first five sets of three words of Table 9, since results of FineReader and Expervision are in sync for the first five words. What is needed is some kind of aligning algorithm that will go through the text and find identical or at least similar parts. To make the problem easier, first we will align only two documents together using ASCII comparison.

Text Alignment of Two Documents Using ASCII Comparison

The aligning process of the three text documents from each OCR engine can be broken down in three parts. First part would take each one of the three possible pairs of the documents and search for similar and different parts. This step will produce three lists. Elements of each one of these lists will contain parts of sentences or words. Elements of those lists are then separated into single using space character as a delimiter. Outcome of this step will give three tables that will contain alignment between the two documents of word pair. Finally the three tables will be combined together to form a single list that will contain three aligned columns of words.

The goal of the first part of text alignment algorithm is to find where is the nearest similar parts of text in the two documents. These two parts should have to be about 3 to 6 characters long, since smaller text segments would not necessarily guarantee same point in original text and bigger text segments is harder to compare to each other. Once a starting text segment of text is

found using various methods listed later in this section it will be determined whether following text is similar or not. In case when following text becomes significantly different, next chunk of similar text will need to be found. Graphical representation of the algorithm is shown in Figure 25.

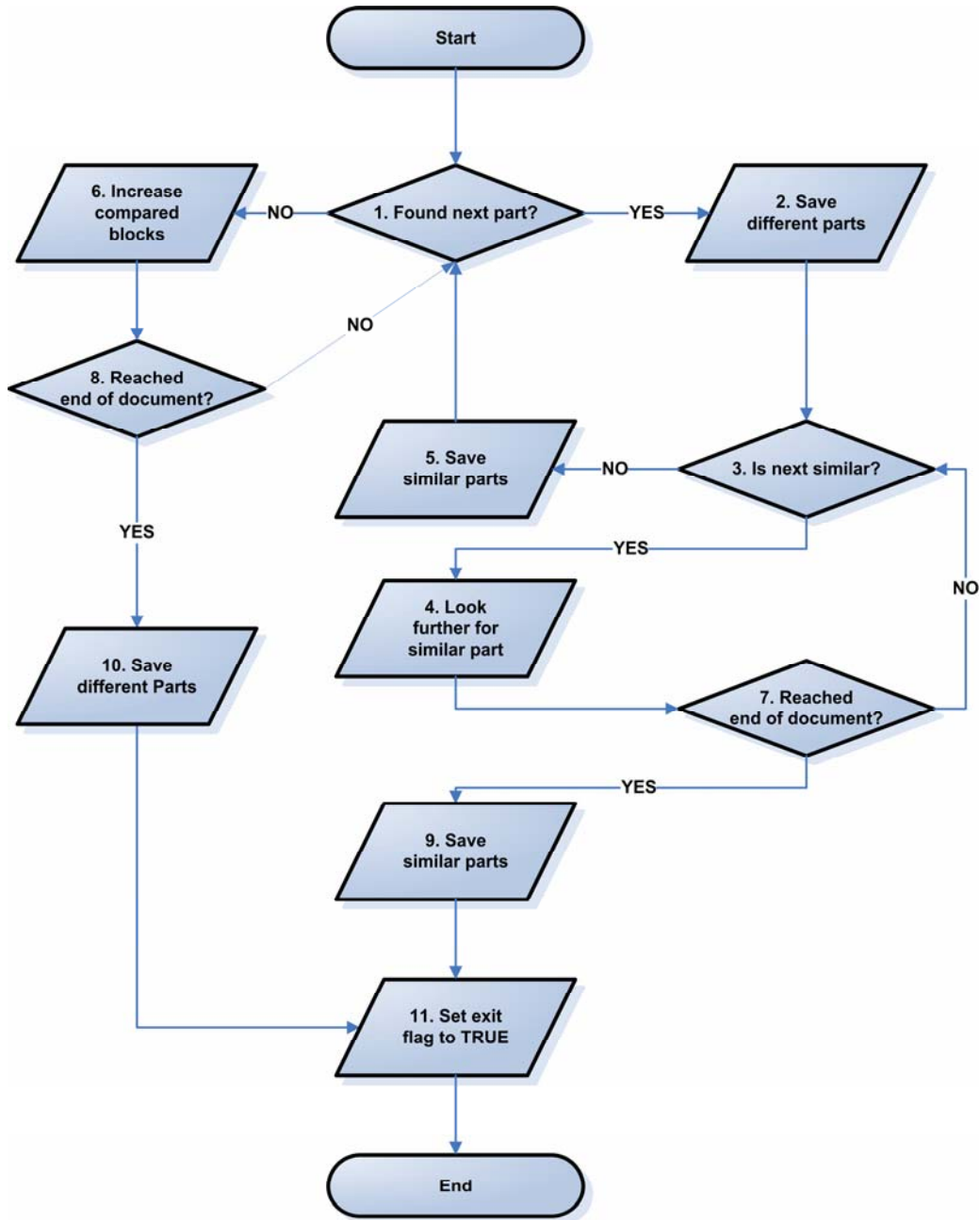


Figure 25: Algorithm for Text Aligning

The algorithm can be divided into eleven blocks and put into a loop that can be terminated by one or more states within the loop. When the algorithm starts it is set to state one, where it searches for the first similar part of the text within the starting amount of characters. If no similar parts were found within given amount of characters, the algorithm goes to state six where the window size in which it searches is increased. Following that, the system goes to state eight and checks if the window is within the document. If the window is bigger than the rest of one of the documents, then there is no reason to search for similar parts since all possible combinations has already been looked at in the previous iteration with the smaller window and the system goes to state ten where it saves the text as not similar. From state ten the algorithm goes into state eleven where the exit flag for the loop is set to an appropriate value to terminate the main loop. If the window still lays within both of the documents the system goes to state one, looking for next similar part within, now extended in state six, range of characters.

Once a similar part is found, the position where it starts in both of the documents is recorded and the system goes to state two where it records the last different parts. From state two the algorithm goes to state three where it checks whether the next group of characters is similar or not. Theoretically, this step could be eliminated and the system could go back to state one. However, when looking for similar characters following already found similar parts the criterion by which the similarity is determined does not have to be as strict, since chances that the characters will be similar are higher than the chances that the first few characters within the default window are going to be similar. If in the state three, it was indeed determined that the characters are similar, system goes into state four, where it goes out further from the last similar part found in state one. Following that, comes state seven that makes sure that the end of any of

the documents hasn't been reached yet. If the search for the next similar group of characters goes beyond one of the documents, the algorithm then goes to state nine where it saves similar parts. From state nine the system goes to state eleven where the exit flag for the loop is set to an appropriate value to terminate the main loop. If in state three, it was determined that the following characters are not similar, the algorithm goes to state five, where it saves similar parts, followed by state one, where it searches for next similar part.

The heart of the algorithm lies in the two most important functions which are "Found next part?" and "Are next characters similar?" The most basic approach is to compare ASCII values of the characters. As it was mentioned before, the first chunk of similar text needs to be about 3 or 4 characters long. As shown in Figure 24, the beginning of the documents is not always similar, which means that several combinations will need to be tried. The algorithm for finding similar parts is shown in Figure 26.

Table 10 demonstrates this algorithm in action. The two texts that need to be aligned are "xxxabczzooo" and "yyabczzppp".

Table 10
Example of Aligning Process Based on the Algorithm of Figure 25

Step	State	Input to state	Output/Action	Memory
1	1	xxx yya	no	Empty
2	6	Size of the window	Size of the window increased by one	Empty
3	8	Current position within each text and current size of the window	no	Empty
4	1	xxxa yyab	no	Empty

Table 10
 Example of Aligning Process Based on the Algorithm of Figure 25

Step	State	Input to state	Output/Action	Memory
5	6	Size of the window	Size of the window increased by one	Empty
6	8	Current position within each text and current size of the window	no	Empty
7	1	xxxab yyabc	no	Empty
8	6	Size of the window	Size of the window increased by one	Empty
9	8	Current position within each text and current size of the window	no	Empty
10	1	xxx abc yy abcz	Yes, current size mask and current position	Empty
11	2	Position where last similar parts ended, current size mask and current position	Current position increased by the positions of found similar parts within window	list(1).diff1="xxx" list(1).diff2="yy"
12	3	zz zz	Yes	list(1).diff1="xxx" list(1).diff2="yy"
13	4	Current position	Increase current position by number of successfully matched characters	list(1).diff1="xxx" list(1).diff2="yy"
14	7	Current position	No	list(1).diff1="xxx" list(1).diff2="yy"
15	3	zx zy	Yes	list(1).diff1="xxx" list(1).diff2="yy"
16	4	Current position	Increase current position by number of successfully	list(1).diff1="xxx" list(1).diff2="yy"

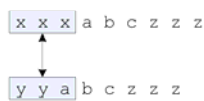
Table 10
 Example of Aligning Process Based on the Algorithm of Figure 25

Step	State	Input to state	Output/Action matched characters	Memory
17	7	Current position	No	list(1).diff1="xxx" list(1).diff2="yy"
18	3	xx yy	No	list(1).diff1="xxx" list(1).diff2="yy"
19	5	Start of current similar part	-	list(1).diff1="xxx" list(1).diff2="yy" list(1).sim1="zz" list(1).sim2="zz"
20	1	ooo ppp	no	list(1).diff1="xxx" list(1).diff2="yy" list(1).sim1="zz" list(1).sim2="zz"
21	6	Size of the window	Size of the window increased by one	list(1).diff1="xxx" list(1).diff2="yy" list(1).sim1="zz" list(1).sim2="zz"
22	8	Current position within each text and current size of the window	yes	list(1).diff1="xxx" list(1).diff2="yy" list(1).sim1="zz" list(1).sim2="zz"
23	10	End of last similar part	-	list(1).diff1="xxx" list(1).diff2="yy" list(1).sim1="zz" list(1).sim2="zz" list(2).diff1="ooo" list(2).diff2="ppp" list(2).sim1="" list(2).sim2=""

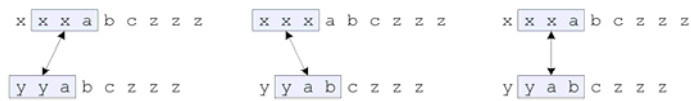
Table 10
 Example of Aligning Process Based on the Algorithm of Figure 25

Step	State	Input to state	Output/Action	Memory
24	11		Set exit flag to TRUE	list(1).diff1="xxx" list(1).diff2="yy" list(1).sim1="zz" list(1).sim2="zz" list(2).diff1="ooo" list(2).diff2="ppp" list(2).sim1="" list(2).sim2=""

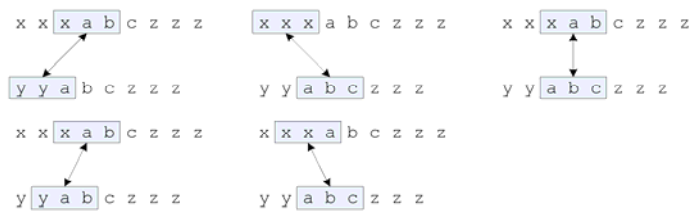
Step 1: compare next 3 characters in the text



Step 2: compare 3-character combinations within next 4 characters in the text



Step 3: compare 3-character combinations within next 5 characters in the text



Step 4: compare 3-character combinations within next 6 characters in the text

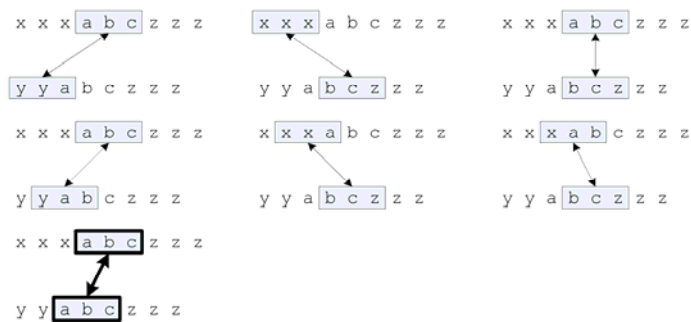


Figure 26: Algorithm for Finding Next Similar Part of Text

In Figure 26 three-character windows were used to find similar segments of text. In this particular example, the non similar part would begin at 0 and end at 2 for the first document, and begin at 0 and end at 1 for the second document. Similarly, the located similar part would begin at 3 for the first document, and at 2 for the second document. Search for the next similar character would begin at positions 6 and 5 respectively. Benchmarking will help to determine whether 3 is an adequate number of characters to compare or, perhaps, longer string of characters will be required. One of the major disadvantages of this method is that number of comparison operations is square of number of steps. With a large number of misrecognized text, this method could be very time consuming. This can be improved by using less strict rules when comparing the strings; for example, using 4-character long strings and calling them equal if at least 3 of 4 characters are the same would flag “ABBY” and “AEBY” as similar text instead of going out further to find an exact match. It is important, however, that the characters that are ignored are not space characters. If there is a disagreement between the two OCR engines whether there should or should not be a space character, it should be left in the unmatched part and dealt with along with other unmatched parts. The diagram for the algorithm of finding next similar part is shown in Figure 27.

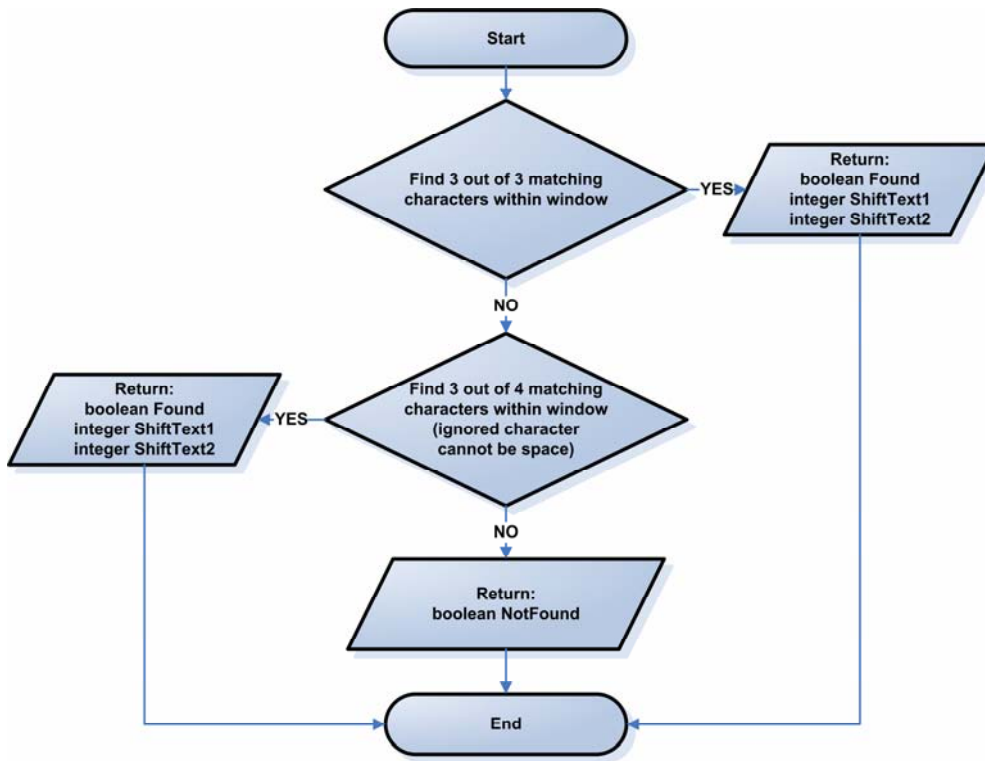


Figure 27: Diagram for Algorithm for Finding Next Similar Part of Text

Since three equal characters in the row is stricter than three equal characters out of four, the algorithm will test for that first. In case such a combination is found, it will return to the main algorithm flag indicating success set to true and positions in each of the documents where similar part begins. If no parts of three characters were found, it'll search for three out of four equal characters. Again in the case of success, it will return a flag indicating success set to true and positions in each of the documents where similar part begins. In case of no combinations were found, the algorithm will return a flag indicating failure set to true.

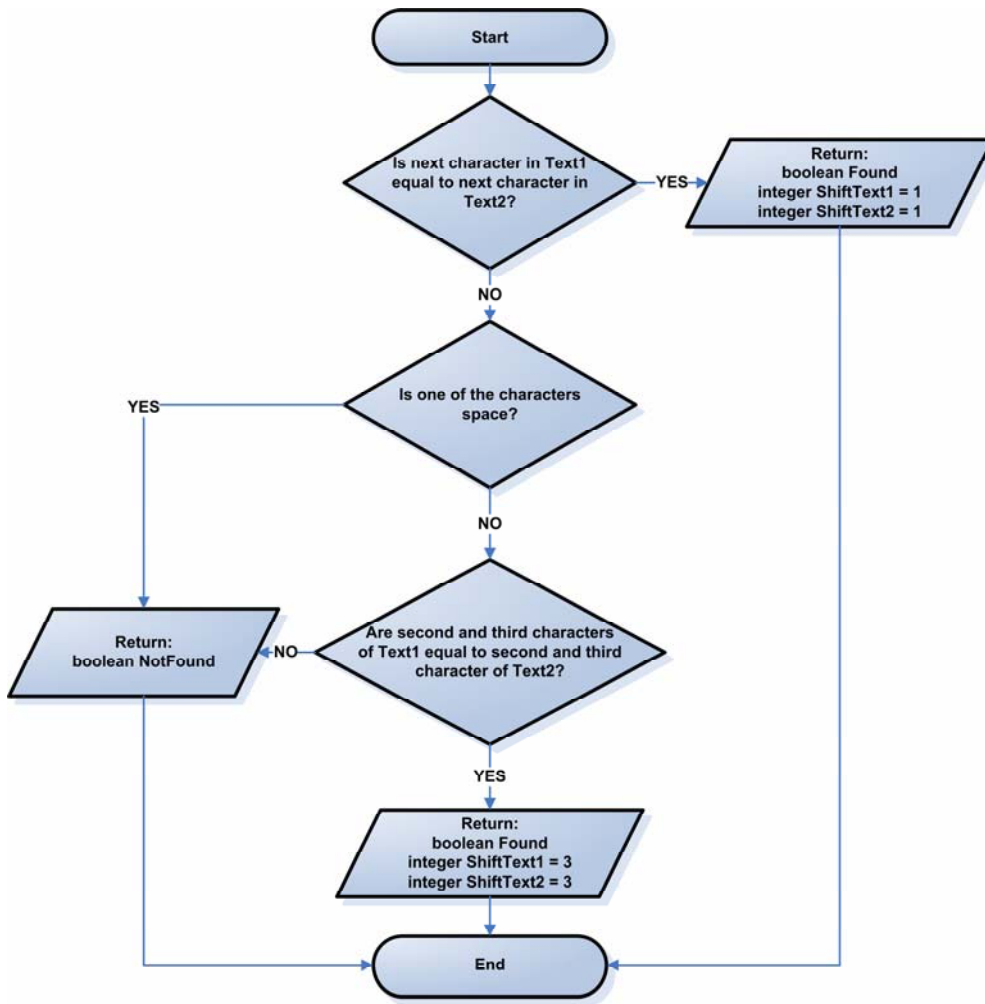


Figure 28: Diagram for Algorithm for Finding Next Similar Group of Characters

The first algorithm checks if the next character of text are equal to another. If they are equal, the algorithm returns a flag indicating success and positions by which next window of comparison needs to be shifted. In this particular case it is always going to be one. If the characters are not the same, in order to preserve space as a delimiting element and before the algorithm will check for characters further ahead, it checks whether one of the characters is a space. If one of the characters is a space (it cannot be both since that would have been flagged as similar characters by the first comparison), the system will exit to main the algorithm with the

failure flag set to true. If, on the other hand, it is not true, the system will compare if second and third character of first text are respectively equal to the second and third characters of second string. If they are all equal, the system will return flag indicating success and positions by which next window of comparison needs to be shifted. In this particular case it is always going to be three. In case they are not equal, the system will exit to main algorithm returning the failure flag set to true.

Since different parts are always followed by similar parts, the most intuitive way to store aligned text at this point is to store it in the list that is defined as follows:

```
listText( i ).stringDifferentPartText1
```

```
listText( i ).stringDifferentPartText2
```

```
listText( i ).stringSimilarPartText1
```

```
listText( i ).stringSimilarPartText2
```

Using the text given as an example in Figure 26 the list containing the aligned part would look as follows:

```
listText( 1 ).stringDifferentPartText1 = 'xxx'
```

```
listText( 1 ).stringDifferentPartText2 = 'yy'
```

```
listText( 1 ).stringSimilarPartText1 = 'abczzz'
```

```
listText( 1 ).stringSimilarPartText2 = 'abczzz'
```

Analysis of the Performance of Aligning Algorithm Based Only on Plain ASCII Comparison

There are two parameters on which the performance of the first part of the aligning algorithm is evaluated. The first parameter is the number of similar characters that were found. It can easily be determined by summing lengths of similar parts. This, however, does not tell us continuity of found similar parts. The second parameter that is important when talking about the performance is the number of similar parts found. Low number of similar parts found along with high total number of similar characters found suggests that there is good continuity. Using results of Doculex's and Abbyy's OCR engines run on a complete image, part of which is shown in Figure 1, the following results were obtained: the number of similar parts found is 147, the total numbers of similar characters length of similar parts for each document are 4682 and 4682, and the total numbers of unmatched characters for both of the documents are 508 and 533. This means that roughly 10% of the text was not recognized as similar. Sample data stored in one of the three lists is shown in Table 11.

Table 11
Sample Data Stored in One of Three Lists after First Part of Alignment

	Element and Property	Value
1	listText(21).stringDifferentPartText1	" .., ' _ ' _to)dsi [red in /gt_ _ "
2	listText(21).stringDifferentPartText2	""
3	listText(21).stringSimilarPartText1	"additional m"
4	listText(21).stringSimilarPartText2	"additional m"
5	listText(22).stringDifferentPartText1	"mt"
6	listText(22).stringDifferentPartText2	"on"
7	listText(22).stringSimilarPartText1	"ey payable pursu0nt to the "
8	listText(22).stringSimilarPartText2	"ey payable pursuant to the "
9	listText(23).stringDifferentPartText1	"Ra"
10	listText(23).stringDifferentPartText2	"Bu"
11	listText(23).stringSimilarPartText1	"y-S"
12	listText(23).stringSimilarPartText2	"y-S"

Looking at first two rows of Table 11, one can conclude that in this particular case the first OCR engine tried to represent some kind of graphics or noise on the page as string of characters, while second OCR engine ignored it. Fifth and sixth lines of the table demonstrate the case when two characters were misrecognized by one or both OCR engines. Rows seven and eight showed that allowing for error digit 0 and corresponding to it in the second text letter “a” were marked as similar. Ninth and tenth rows again show misrecognized characters by one or both OCR engines. In this particular case, however, what draws attention is the fact that the

two fragments of text “Ra” and “Bu” look very similar, perhaps using some kind visual comparison would be more appropriate in this case.

Text Alignment of Two Documents Using Visual Comparison

Further examining parts of the documents that were not marked as similar, it can be noted that some of those parts even though have different ASCII values look very similar. Some of those fragments are shown in

Table 12
Visually Similar Fragments of Texts That Were Not Flagged as Similar During Plain ASCII Comparison

<u>OmniPage</u>	<u>Omni</u>
“a”	“e”
“pah”	“pub”
“cca”	“een”
“al”	“ni”
“Ra”	“Bu”
<u>“0Q0”</u>	<u>“000”</u>

Figure 29 demonstrates how visual character comparison could mark different characters as similar.

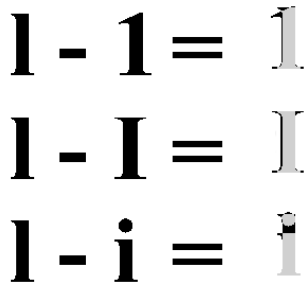


Figure 29: Example of Visual Character Comparison

In Figure 29 the lowercase letter “l” is compared to capital “I”, digit “1”, and the lowercase letter “i”. In grey shown parts of the characters that are common to both the compared characters. Parts of characters that are not common to both characters are shown in black. The amount of black essentially tells how much the characters are different from one another.

Since OCR is based mostly on feature extraction, different widths of different parts of the same character do not have to be accounted. For example, in Times New Roman font, capital letter “W” has different widths depending on the direction of a particular feature, while same letter of font Courier New has the same width for every part of the letter as shown in Figure 30.



Figure 30: a) Capital “W” of Font Times New Roma, b) Capital “W” of Font Courier New

Since essentially visual character comparison is based on image subtraction, better results could be achieved if each character has constant width, which makes characters of font Courier

New a better choice. Before character subtraction can take place, each pair of characters must be aligned in the best possible way. While some characters can be aligned easily manually, such as lowercase letter “l” and upper case letter “I”, other characters are not that easy to align manually; for example digit “9” and symbol “\$” shown in Figure 31.



Figure 31: Two characters that need to be aligned

One of the ways to do this automatically is to use image correlation, mathematical form of which is given in the equation 3.1.

$$c(x, y) = \sum_s \sum_t f(s, t) w(x + s, y + t) \quad (3.1)$$

where:

c – Result of correlation

f, w – correlated images



Figure 32: Character padded with zeros

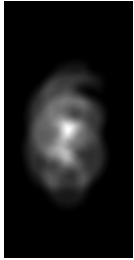


Figure 33: Correlation of two characters shown in Figure 31

In this case images that need to be correlated are of equal size. One of the images w is padded with zeros, as shown in Figure 32, so that its new dimensions are three times bigger than the original dimensions. Image f is then moved through image w and at each step it is multiplied with current overlapping part of image w . In case of binary image, high values will only remain where high values of both images overlap. Those values are then summed together, and this value is set to image c at position corresponding to current position of image f within image w . Correlation of the two characters is shown in Figure 33. The highest value then represents the point where pixels with high value overlapped the most [6]. This point dictates how one character needs to be shifted with respect to another so that the difference between the two would be minimal. This point might not be necessarily unique, but mathematically there is no difference between the points where correlation is at maximum, and any of these points can be chosen. Both overlapped and not overlapped parts of the characters are of an interest. In order to preserve this information, different parts can be stored in the difference image as grey, and overlapped parts can be stored as white as demonstrated in Figure 34.



Figure 34: Difference of aligned characters; in white shown overlapped parts and in grey shown parts that did not overlap

Since the amount of black can vary depending on how much zero padding was added during correlation and during image created, to make amount of black comparable to amount of grey and white, we can calculate what is the lowest amount of black pixels present in all of the pictures, and subtract it from the rest of the pictures. For this particular set of images, this value is 17562 with total number of pixels 18432 each.

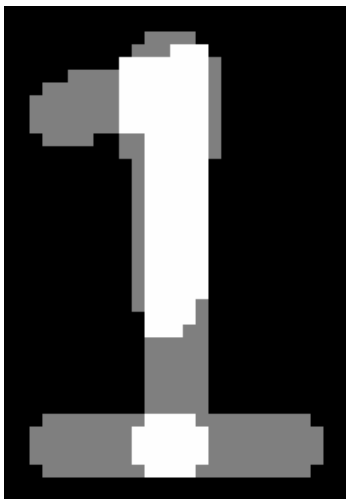


Figure 35: Difference of aligned characters; certain parts of unaligned pixels are de to difference in width of some features of characters

Figure 35 shows that sometimes for certain characters unaligned pixels can show up because of the difference in widths of certain features of the characters. While those parts are perfectly aligned, presence of those unaligned pixels can through off similarity value. One way to approach this problem is to apply opening (dilation followed by erosion) filter to the grey part of the image.

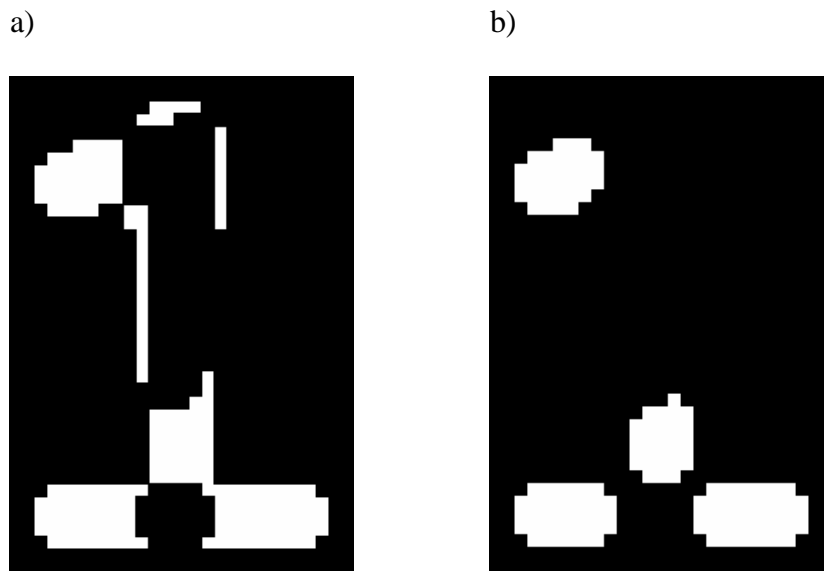


Figure 36: a) Unaligned pixels, b) Unaligned pixels after application of closing filter with structuring element of cross shape and size 3x3 pixels

Figure 36.b shows the unaligned pixels after closing filter was applied. Size of the structuring element in this case was 3x3. Since the width of features of the characters is on average 6 pixels, applying filter of the size bigger than the half of the character would lead to loss of unmatched pixels that are important in calculation of similarity value of the two characters. Because most of the features of characters of Courier New have rounded edges, cross shape, closest to round shape in case of 3x3 structuring element, was chosen.

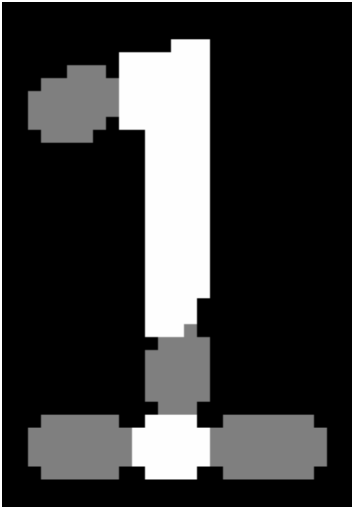


Figure 37: Closed unaligned pixels added back to aligned pixels

Figure 37 was acquired by disregarding grey pixels of image Figure 35 and adding to it image shown in Figure 36.b as grey pixels. As a result, the difference of the two characters does not include false unaligned pixels.

Now that a good representation of aligned and unaligned pixels is acquired, some kind of mechanism by which a similarity value of the two characters can be determined. Since the number of pixels is fixed, numbers of white, black and grey pixels can be represented in the form of equation of a plane as shown in equation 3.2.

$$b_p + w_p + g_p = S \quad (3.2)$$

where:

b_p , w_p , and g_p – numbers of black, white, and grey pixels

S – total number of pixels

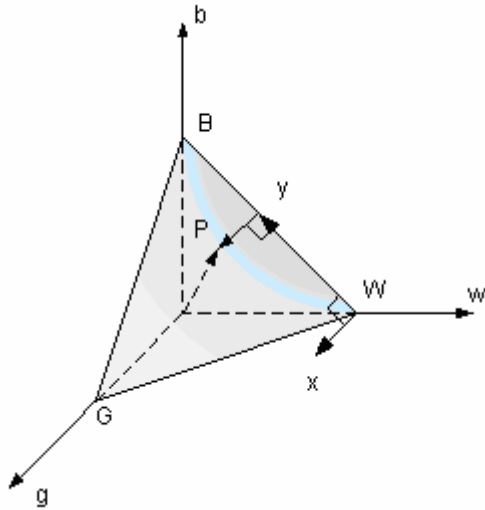


Figure 38: Graphical representation of amount of pixels of each of three colors.

Figure 38 demonstrates 3D representation of possible combinations of colors in the image. Depending on where a particular image falls, the combination of colors, the value of similarity of the two characters will be assigned. First, transformation from 3D coordinates (bwg) to coordinates in the plane (xy) is needed as shown in Figure 38. When P falls on W it means that all pixels aligned and that it is the perfect match, which means that the similarity will be set to 1. When P falls on B, it means that there were no pixels and two space characters are compared and their similarity is also 1. If, on the other hand, P falls on G, it means that there was nothing similar about the two characters and that their similarity will be set to 0. To simplify further calculations it is better to normalize the coordinates in xy coordinate system, meaning that both x and y will have range [0, 1]. Using geometry it can be shown that coordinates of point P(b, e, g) can be related to coordinates (x,y) in plane BWG by equation 3.3.

$$\begin{aligned}
 x_p &= \frac{g_p}{S} \\
 y_p &= \frac{b_p - w_p + S}{2 \cdot S}
 \end{aligned}
 \tag{3.3}$$

Depending on where particular color combination falls value associated with similarity will be chosen. If image is all black or all white, similarity value should be one; if however, image is all gray, similarity value should be zero. The first obvious choice is to discard y_p , and express similarity value as shown in equation 3.4.

$$SimVal(x_p) = 1 - x_p
 \tag{3.4}$$

However, examining closely what happens along edges WG and BG the following can be noted. While along WG edge similarity between the two characters can still be expressed as linear function, along BG edge things are not as straight forward. For larger amounts of grey pixels and no white pixels similarity value needs to be zero; however, smaller amounts of grey pixels and no white pixels indicate presence of small characters that could potentially be punctuation marks or noise. While some OCR engines could recognize them as comma or a dot, others could have simply discarded them. In this case an exponentially decaying or piecewise constant function with similar properties would be appropriate. Since most of the time combinations of grey, white and black colors will fall inside the triangle, an interpolated value of the two functions can be taken. Equation 3.5 demonstrates such an approach.

$$SimVal(x_p, y_p) = f_{WG}(x_p) \cdot y_p + f_{BG}(x_p) \cdot (1 - y_p) \quad (3.5)$$

$$f_{WG}(x_p) = 1 - x_p \quad (3.6)$$

$$f_{BG}(x_p) = \begin{cases} 1 - \frac{x_p}{2} & 0 \leq x_p \leq 0.1 \\ 1.1875 - 2.375 \cdot x_p & 0.1 < x_p \leq 0.5 \\ 0 & 0.5 < x_p \leq 1 \end{cases} \quad (3.7)$$

Equation 3.6 defines linear function along WG side, and equation 3.7 defines piecewise linear function along BG side.

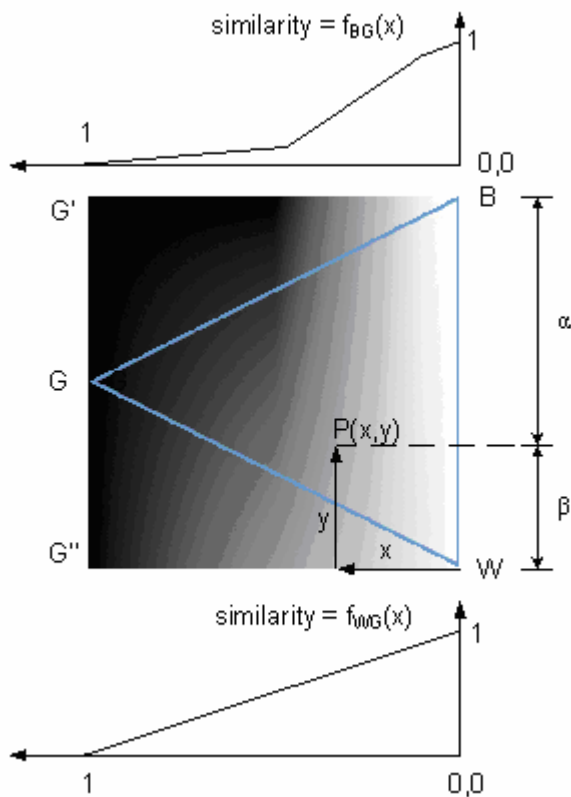


Figure 39: Graphical representation of equations 3.5, 3.6, and 3.7

Figure 40 is a graphical demonstration of equations 3.5 through 3.7. It can be noted that the functions f_{BG} and f_{WG} do not really go along BG and WG sides respectively, in fact they are going along BG' and WG'', which distorts the desired out put. Equation 3.8 is an improved *SimVal* function that takes this fact into consideration.

$$SimVal(x_p, y_p) = \begin{cases} f_{WG}(x_p) \cdot \frac{y_p - \frac{x_p}{2}}{1 - x_p} + f_{BG}(x_p) \cdot \frac{1 - y_p - \frac{x_p}{2}}{1 - x_p} & x_p \neq 1 \\ 0 & x_p = 1 \end{cases} \quad (3.8)$$

At glance, it might seem that there is a discontinuity at x_p equal to one and that the values could go to infinity. Examining closely, it turns out that for both fractions both denominator and numerator are linearly approaching zero, which means that, even though, there is a discontinuity, the limit of both fractions as x_p approaches one is equal to a constant value. Since the values of x_p are discrete and they will not be close enough to one, this problem can be overcome by simply equating the function to a value of zero.

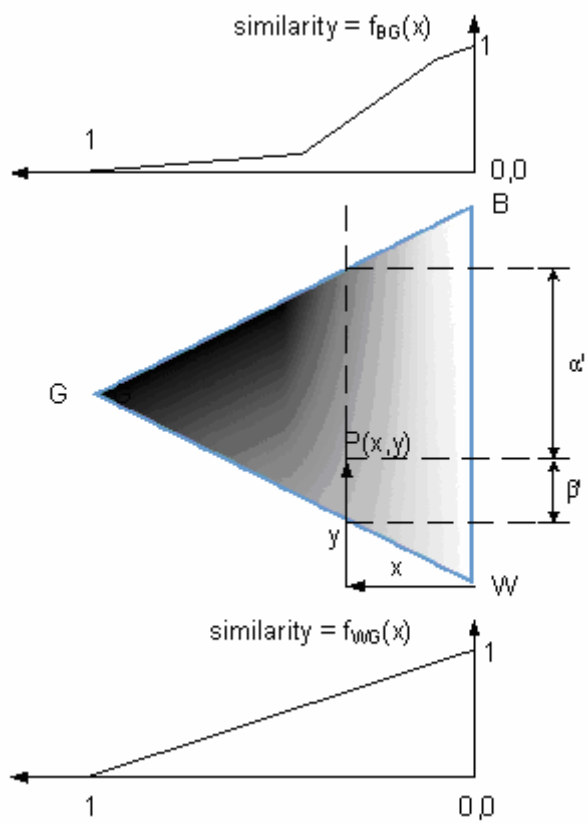


Figure 40: Graphical representation of equations 3.6, 3.7, and 3.8

As shown in Figure 40, functions f_{BG} and f_{WG} this time really do go along BG and WG sides of triangle.

Now that all tools for comparing characters have been acquired, table with similarity values can be built.

Table 13
 Fragment of a table for character similarity look-up

	a	b	c	d	e	f	g	h	i	j
a	1.000	0.694	0.830	0.734	0.964	0.615	0.577	0.851	0.806	0.494
b	0.694	1.000	0.867	0.644	0.795	0.655	0.571	1.000	0.646	0.418
c	0.830	0.867	1.000	0.921	0.925	0.694	0.800	0.885	0.718	0.534
d	0.734	0.644	0.921	1.000	0.804	0.658	0.711	0.807	0.747	0.634
e	0.964	0.795	0.925	0.804	1.000	0.624	0.694	0.833	0.648	0.465
f	0.615	0.655	0.694	0.658	0.624	1.000	0.549	0.653	0.829	0.486
g	0.577	0.571	0.800	0.711	0.694	0.549	1.000	0.631	0.461	0.701
h	0.851	1.000	0.885	0.807	0.833	0.653	0.631	1.000	0.609	0.489
i	0.806	0.646	0.718	0.747	0.648	0.829	0.461	0.609	1.000	0.747
j	0.494	0.418	0.534	0.634	0.465	0.486	0.701	0.489	0.747	1.000

Table 13 is a part of lookup table for character similarity. For i equal to j , where i is row index and j is column index, the value in the table is one, because character is exactly equal to itself. Since similarity of character "a" to character "b" is the same as similarity of character "b" to character "a", the table is also symmetric about i equal to j . Highlighted in yellow is a great example of two characters, "b" and "h", similar to each other. In reality this value is less than one; however, rounded up to three digits of precision it came out to be one. In certain cases, intuitively, similarity value came out to be too high. For example, characters "d" and "c" highlighted in red in the table. In Courier New font most of the weight of characters is in lower portion, which means that the upper part of the character "d" might not have had enough weight

to make similarity value lower. This problem can be partially solved by choosing a different font and benchmarking the overall performance of the multi-engine OCR system; however this is out of the scope of this thesis.

In practice, when coding the alignment engine, in order to simplify expressions that will need to be calculated, it will be easier to use difference value instead of similarity. A difference value of two characters is simply one less similarity value of the two characters.

Table 14
Fragment of a table for character difference look-up

	a	b	c	d	e	f	g	h	i	j
a	0.000	0.306	0.170	0.266	0.036	0.385	0.423	0.149	0.194	0.506
b	0.306	0.000	0.133	0.356	0.205	0.345	0.429	0.000	0.354	0.582
c	0.170	0.133	0.000	0.079	0.075	0.306	0.200	0.115	0.282	0.466
d	0.266	0.356	0.079	0.000	0.196	0.342	0.289	0.193	0.253	0.366
e	0.036	0.205	0.075	0.196	0.000	0.376	0.306	0.167	0.352	0.535
f	0.385	0.345	0.306	0.342	0.376	0.000	0.451	0.347	0.171	0.514
g	0.423	0.429	0.200	0.289	0.306	0.451	0.000	0.369	0.539	0.299
h	0.149	0.000	0.115	0.193	0.167	0.347	0.369	0.000	0.391	0.511
i	0.194	0.354	0.282	0.253	0.352	0.171	0.539	0.391	0.000	0.253
j	0.506	0.582	0.466	0.366	0.535	0.514	0.299	0.511	0.253	0.000

Table 14 shows a fragment of look-up table with character difference values. Just like Table 13, this table is also symmetric about its diagonal, and value of zero now represents identical characters.

Table 15
Four Characters with Most Similar to Them Characters

Character “1”		Character “8”		Character “I”		Character “[“	
l	0.004807	S	0.049867	l	0.044152	{	0.025746
I	0.046466	B	0.063210	l	0.046466	(0.045318
i	0.089090	6	0.075887	T	0.075402		0.055896
L	0.112948	3	0.118504	L	0.076215	!	0.114276
j	0.148680	0	0.124118	i	0.168751]	0.132074
!	0.166219	9	0.159772	f	0.196556	}	0.149722
T	0.171959	H	0.167910	J	0.199795	1	0.194807

Table 15 shows four characters and the characters similar with difference values next to them. When performing text alignment a threshold value will be chosen to determine how big the difference can be in order for the two characters to be considered similar. Depending on how big this value is, some characters will have more similarity with some characters than the others. For example, setting the maximum difference value to 0.005 out of four characters in Table 15 only character “1” will have a similar to it character. In the plain ASCII comparison algorithm similar parts were determined only when one character in one text file was equal to the character in the other text file. Since the lookup table does include comparison of a character to itself, and the difference value is zero, it will always be flagged as similar independent of what the threshold is. This means that the plain ASCII comparison can be replaced by the visual comparison.

**Analysis of the Performance of
Aligning Algorithm Based on Visual Character Comparison**

Outcome of this algorithm can also be characterized by the same parameters as the outcome of algorithm based only on plain ASCII comparison. Using value of 0.15 for threshold in finding the next similar word and 0.2 in finding the next similar character, number characters found similar went up from 4682 to 4823. Number of characters marked as different went down for both documents from 508 and 533 to 367 and 392. This is a 30% improvement, and while it is relatively small compared to the total number of characters in the document, this small difference can make great impact on the further aligning. Total number of similar parts also went down from 147 to 133. Part of the algorithm that checks if next pair of characters is similar once the start of similar part is found also is now based on visual character comparison. Decrease in number of similar parts indicates that when plain ASCII comparison failed to continue marking similar parts as similar, visual character comparison succeeded. Values used for threshold can greatly impact the outcome. Increasing the thresholds can improve the results; however, increasing them too much can introduce misalignment.

Table 16
Misalignment due to High Thresholds

Unmatched Text 1	Unmatched Text 2	Matched Text 1	Matched Text 2
"n"	""	" c"	" h"
"ompl"	""	"ied "	"ave "
"with,"	"been"	" a"	" c"
"m"	"omplie"	"i w"	"d w"

Table 16 demonstrates that setting threshold for finding next similar parts at 0.3 and next similar character at 0.1 resulted in text misalignment. Second line of unmatched text one actually matches fourth line of unmatched text two.

<pre>y rights to which it or t1)e Till y rights to which it or rile 1'il</pre>
--

Figure 41: Example of Success of Visual Character Comparison

Figure 41 shows example of when plain ASCII comparison would fail. From the beginning of second to last word to the end of the last word out of nine characters only three are exactly the same. Plain ASCII comparison would flag this part as different, while visual comparison saw where the OCR engines made errors and flagged those parts as similar.

Table 17
Visually Similar Fragments of Texts That Were Not Flagged as Similar During Visual Character Comparison

<u>OmniPage</u>	<u>Abby</u>
“m”	“in”
“ii”	“u”
“li”	“h”
“am)”	“and”
“rn”	“m”
“nd”	“iul”

Table 17 demonstrates some of the unmatched by visual character comparison fragments of text that visually are similar. The reason why visual comparison failed in these cases is that

the characters got broken apart or merged together. As Sprague noted before, due to the fact that some of the documents were copied over and over again some of the characters lost their least significant features [3]. In case of most characters vertical features are more important than the connecting horizontal ones. This resulted in characters breaking apart. For example character “m” seems to be often misrecognized by Abbyy’s OCR engine as “in” or “rn”. One of the ways to approach this problem is to include a error lookup table for visual comparison not only single characters but also combination of characters. Unfortunately, there are too many combinations that need to be accounted for, which could result in long computational times. Since most of the time vertical features are preserved, another approach is to represent each character as series of vertical lines as shown in Figure 42.

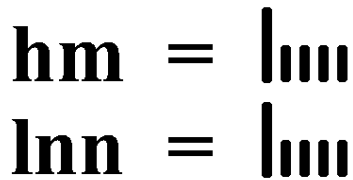


Figure 42: Example of Line Representation and Comparison of Characters

Figure 42 shows an example, where two characters are compared to three characters. Examining unmatched parts and most common fonts in general it can be noted that there are two most common heights of vertical features of characters.

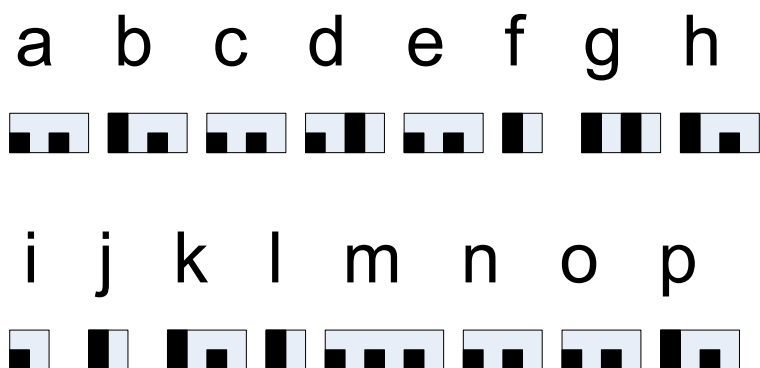


Figure 43: Line Representation of Characters “a” through “p”

Figure 43 shows line representation of characters “a” through “p”. Characters “a”, “c”, “e”, “n”, and “o” have the same line representation, which limits application of this method of character comparison. When using this method looking for next similar character or group of characters once the start of similar part has been found, there is a good chance that the following characters are indeed the same. On the other hand, when looking for the start of similar parts of the two texts, this method can introduce false results. There are several ways how this line representation of characters can be generated. One is to generate this list manually based on unmatched text. The other method is to generate these values automatically by generating images of characters, summing pixels vertically, and then applying thresholds.

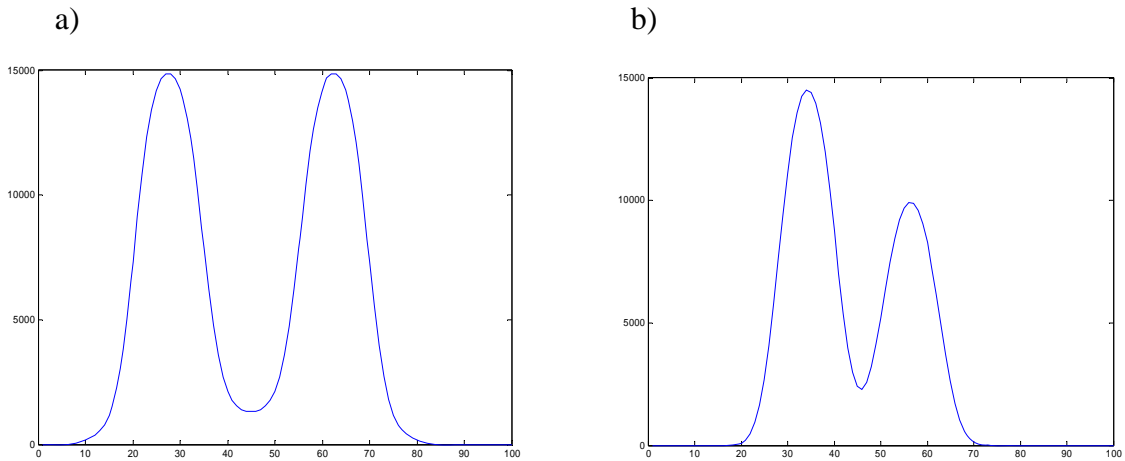


Figure 44: Vertical Summation of Pixels of a) Character “H” and b) Character “h”

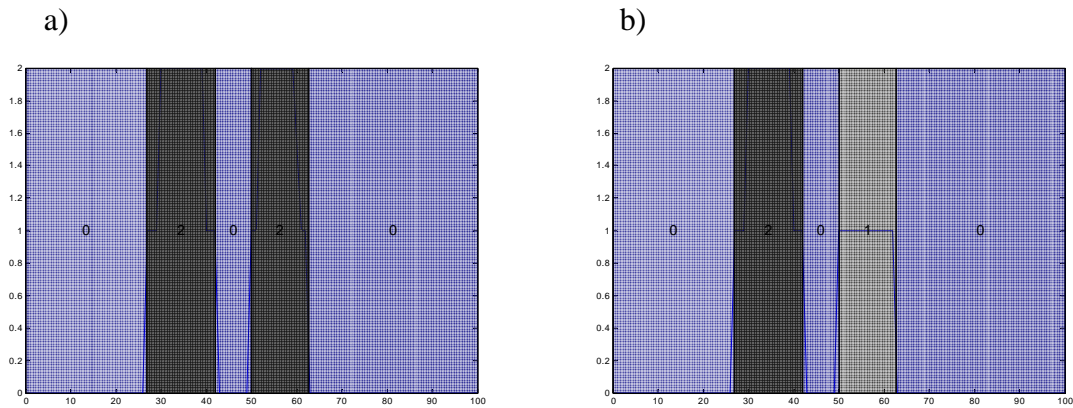


Figure 45: 3-level Threshold Applied to Figure 44

Figure 44 and Figure 45 show the two steps of the process that assigns line representation of characters on the example of characters “H” and “h”. For better results, the Impact font can be chosen. Characters of this font have vertical features enhanced, which will reduce amount on levels after threshold, such as shown in Figure 45. Instead of going from level zero to level two directly, level one appears in between. This can be reduced by using Impact font. To completely

eliminate this intermediate level, its width needs to be compared against previous and next levels widths. If it is considerably smaller, then it can be discarded.

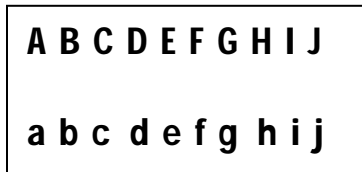


Figure 46: Characters of Font Impact

Figure 46 demonstrates characters of the Impact font. This font is better for automatic line representation of characters because vertical features are more enhanced compared to horizontal ones, and they also have the same widths for all characters. A leading zero is not necessary and can be discarded. Line representation of characters “H” and “h” can be stored as follows:

```
listLineChars(1).line(1) = 2
```

```
listLineChars(1).line(2) = 0
```

```
listLineChars(1).line(3) = 2
```

```
listLineChars(1).line(4) = 0
```

```
listLineChars(2).line(1) = 2
```

```
listLineChars(2).line(2) = 0
```

```
listLineChars(2).line(3) = 1
```

```
listLineChars(2).line(4) = 0
```

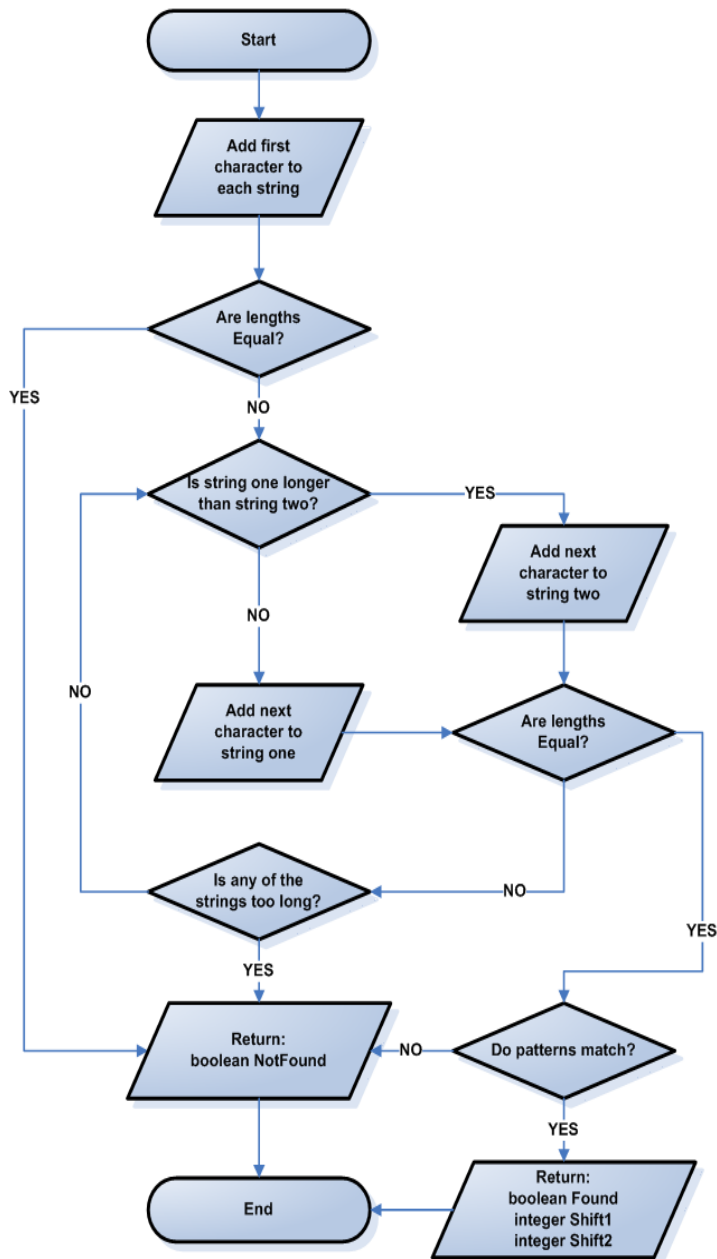


Figure 47: Algorithm for Detecting Next Similar Character Using Line Representation of Characters

Figure 47 shows an algorithm that uses line representation of characters to group character-wise uneven groups and mark them as similar. Figure 48 demonstrates application of this algorithm on two strings “lnn” and “hm”, which have different lengths yet similar looks.

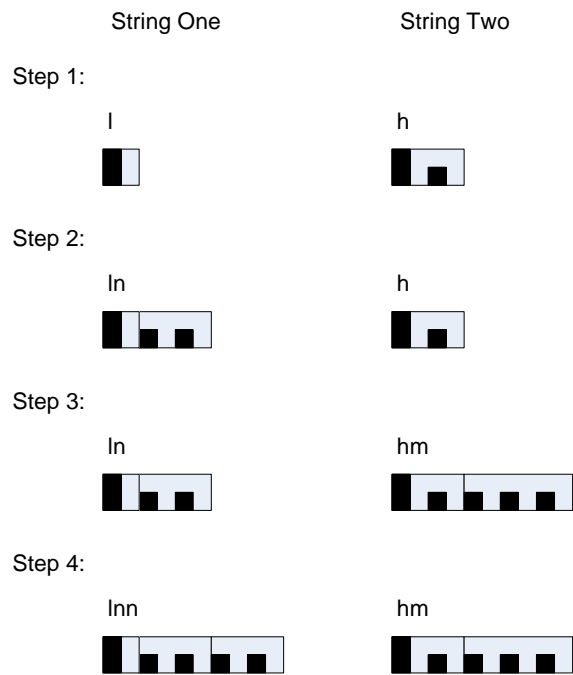


Figure 48: Algorithm for Detecting Next Similar Character Using Line Representation of Characters in Action

Once line-wise length of the two strings is equal, their patterns are compared. If they are identical, these two groups of characters are marked as similar. Looking at previous results of alignment using visual character comparison, it can be noted that these fragments would not exceed lengths of five characters. Once the length of one of the strings exceeds five characters, the algorithm returns to the main program indicating end of a similar part. To avoid uneven number of space characters within similar part it is also being omitted. If one of the added characters is a space, the algorithm also exits indicating end of a similar part.

Analysis of the Performance of Aligning Algorithm Based on Visual Character Comparison and Line Representation of Characters

Just like in previous analyses, there are three parameters that affect the performance of the alignment. Number of similar part went down from 133 to 106. This indicated that the continuity has increased. Number of similar characters went up from 4682 for both documents to 4912 for Abby's OCR engine and to 4906 for Omni's OCR engine. Note that before these two numbers were the same for both engines. This time line representation of characters made it possible to match uneven number of characters and mark these parts as similar. From the difference in the number of similar characters it can be concluded that Abby's OCR tends to split characters, while Omni's OCR tends to merge them.

Table 18
Groups of Characters That Were Marked as Similar by Adding Line Representation of Characters

<u>OmniPage</u>	<u>Abby</u>
h	li
ll	U
ha	lw
a	ii
ro	m
E	li
in	m
u	ii
mad	nuul
<u>ru</u>	<u>m</u>

Table 18 demonstrates some more examples in addition to the ones shown in Table 17, where line representation of characters made successful detection of similar parts.

Improving Algorithm for Finding Next Similar Part

While the algorithm for finding the next similar part of the text has shown to be efficient, it has a few major drawbacks. It has a rather complex implementation, which in turn causes it to be difficult to adjust for certain cases and applications. Matrices are known for their ability to simplify complex equations and some times even series of complex equations. A matrix-like approach can be taken in the algorithm for finding similar parts. Two strings “xxxabczzz” and “yyabczzz” will be used as an example.

First of all, a representation of all possible combinations is needed. Since the smallest number of characters that are used to flag text as a beginning of similar part is three, number of possible combinations for strings of three characters long is one, four character-long strings is four, etc.

x	x	x	a	b	c						
	x	x	x	a	b	c					
		x	x	x	a	b	c				
			x	x	x	a	b	c			
				x	x	x	a	b	c		
					x	x	x	a	b	c	
						x	x	x	a	b	c
			y	y	a	b	c	z			

Figure 49: Matrix-like Representation of All Possible Three-character Combinations within Two Six-character Long Strings

Figure 49 demonstrates how the matrix for finding next similar part can be found. Picking each non empty cell, above a dashed line, and such that the number of character to the left is at least two will give a unique three-character combination. In case of four-character combinations, number of character to the left must be at least four. In general, number of characters to the left must be length of compared characters (character within the mask) less one, as demonstrated in Figure 50.

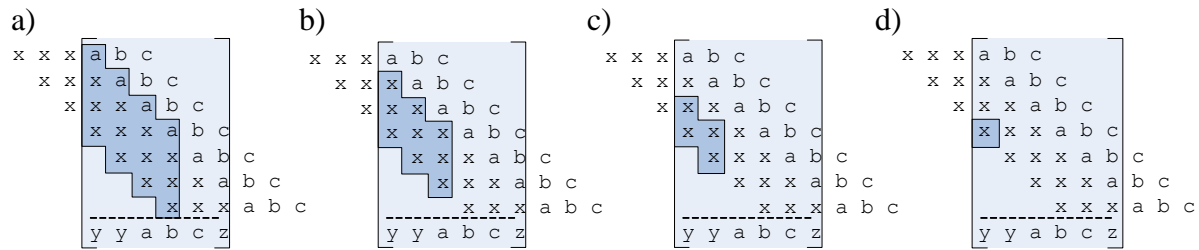


Figure 50: Shaded Regions Show Where Combination of Compared Characters Starts for a) Three-, b) Four-, c) Five-, and d) Six-Character Long Masks

Next step is to replace characters above the dashed line with a difference value that corresponds to the character below the dash line. Empty cells can be replaced with values of one. This way they will not be accounted as similar parts in further calculations.

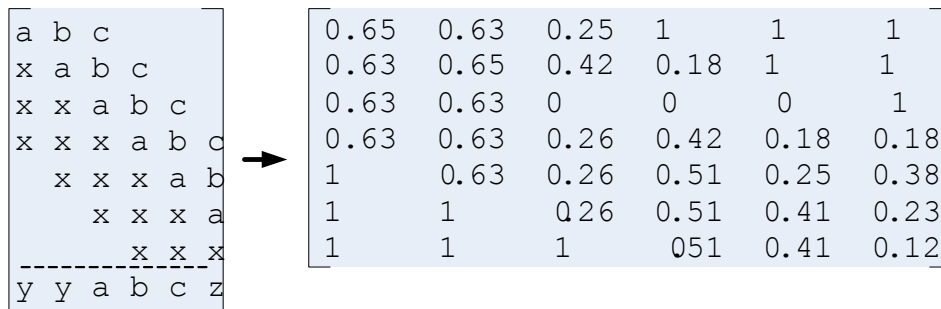


Figure 51: Translating Matrix of Characters into Matrix of Difference of Characters

Figure 51 demonstrates how to get from matrix of characters to matrix of difference values. To this new matrix can be applied series of masks. In the implementation of aligning algorithm following masks were used: [1 1 1], [1 0 1 1], [1 1 0 1], and [1 0 1 0 1]. These are the masks of size up to five characters that will have unique results. Mask [1 1 1 0] has the same effect as mask [1 1 1], similarly mask [1 1 0 1 0] will have the same effect as mask [1 1 0 1]. Value of one indicates that the character difference that falls on it will be accounted, and value zero indicates that the character difference that falls on it will be ignored.

Table 19
Examples of Masks and Their Applications

Mask	Array of Differences	Output
0.1 0.2 0.3	[1 1 1]	0.3
0.1 0.9 0.2 0.3	[1 1 0 1]	0.9
0.1 0.9 0.2 0.3	[1 0 1 1]	0.3

Table 19 shows how the masks are applied. At each point of the character difference matrix each mask is applied. Output of each applied mask is the maximum value of the character

difference values that fall on cell of the mask of value one. If at any point the mask partially falls outside of matrix, output in that case is set to one.

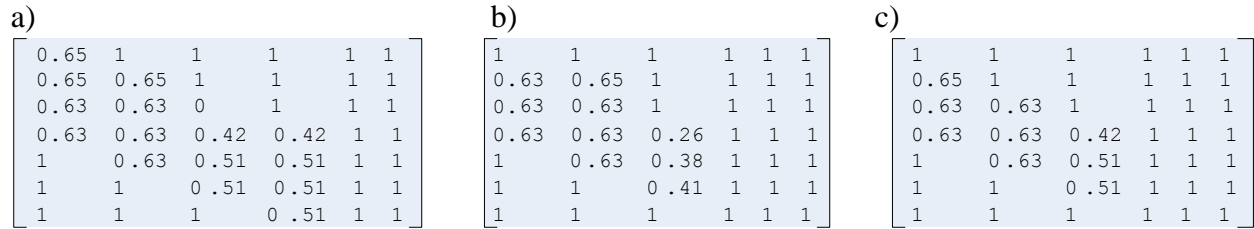


Figure 52: Masks a) [1 1 1], b) [1 0 1 1], and c) [1 1 0 1] Applied to Character Difference Matrix

Figure 52 shows result of the application of the three masks. It can be noted that the last N-1 columns, where N is the length of the mask, will always be filled with ones.

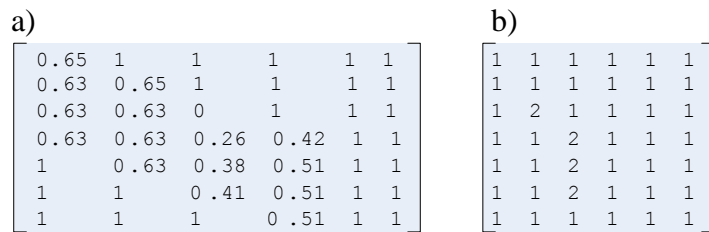


Figure 53: a) Minimum Values of the Three Matrices, b) Number of Matrix to Which the Minimum Value Belongs

Once the masks have been applied to the character difference matrix, the matrices need to be combined into one that will have the best (minimum) values. Also an auxiliary matrix is needed that will keep track of which mask generated the best output. This will allow the algorithm to know how many characters (size of the mask) have been marked as similar, which

indicates at which point it needs to start looking for the next similar character. Figure 53 shows the threshold results of these two matrices.

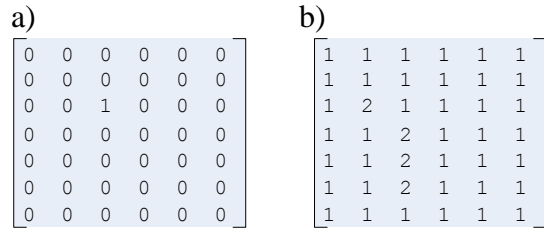


Figure 54: a) Minimum Values of the Three Matrices after Threshold Was Applied, b) Number of Matrix to Which the Minimum Value Belongs

After the minimum outputs of the applied masks have been found, the next step is to find which of these values satisfy the maximum difference. Cells that have value less than the threshold are equated to one and the ones that are larger are assigned a value of zero. It is not uncommon to get more than one cell with a value one. The best choice is the one where the shift is minimal.

$$\begin{aligned} Shift_1' &= N - M + k + l - 2 \\ Shift_2' &= l - 1 \end{aligned} \tag{3.9}$$

where:

$Shift_1'$ and $Shift_2'$ – beginning of similar parts for the two strings

N – length of compared strings

M – length of smallest mask

k, l – position in difference matrix

Equation 3.9 shows the relationship between position within the difference matrix and the beginning of similar parts within each compared strings. Theoretically, the shift values could be

negative; however, those values are only possible when the position in the difference matrix falls on an empty cell. This cell will always have a difference value one and will never have a value of one after the threshold has been applied. If there are several values of one, the best choice would be the one that has the smallest sum of squares of these two values.

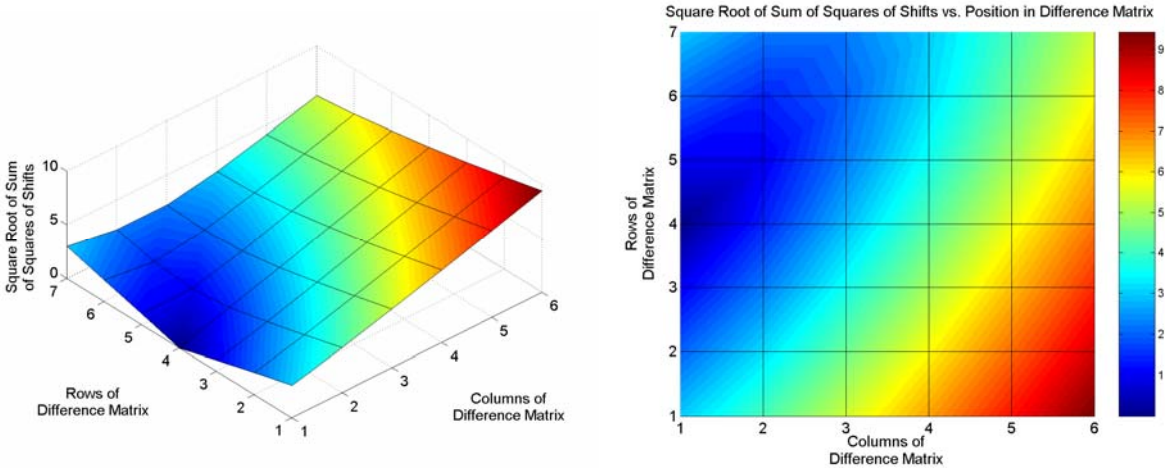


Figure 55: Sum of Squares of Shifts versus Position in Difference Matrix in 3D and 2D views

Figure 55 shows graphical representation of relationship of square root of sum of squares of shifts versus position within difference matrix. There is no shift required when there is value one in first column middle row.

This new improved method does not improve the amount of text found similar. It allows the user to easier implement, change, and most importantly, to debug the source code.

Alignment of individual words of two texts

Once similar parts were found, next step on the way to alignment of three text documents is to align individual words of two texts. Before proceeding, recall that the similar and different parts are stored in the memory following way:

```
listText( i ).stringDifferentPartText1
```

listText(i).stringDifferentPartText2

listText(i).stringSimilarPartText1

listText(i).stringSimilarPartText2

It is also important that the similar parts go after the different parts. For example, if the two texts would start out as similar, first different parts would be empty. The output of this process will be a list, each element of which will contain two string variables.

listTwoTexts(j).stringWord1

listTwoTexts(j).stringWord2

Since no space characters are allowed in the similar parts unless they match to another space characters, similar parts can be separated into words using a space character as a delimiter. Different parts, however, might contain different number of spaces, which makes it impossible to just use space as a delimiter. Fortunately, all possible combinations can be divided into finite number of cases shown in Table 20.

Table 20
Possible Combinations of Numbers of Spaces in Different Parts

<u>Case Number</u>	<u>Case Description</u>
1.1	Number of spaces is the same for both different parts
1.2	Number of characters in one of the parts is zero, and the other one is not
1.3	<u>Numbers of spaces in both parts are not zero, but not equal to each other</u>

In addition the cases listed in Table 20 additional complexity is created by the fact that the previous similar parts of the two texts might end with space or other than space character and the current similar parts might begin with space or other than space character. Since numbers of spaces and their positions have to be the same for similar parts, it is true to say that if the similar

part of one text begins and/or ends with space then the similar part of the other text also begins and/or ends with space.

Table 21
Possible Combinations of Ending and Beginning of Similar Parts

Case Number	Case Description
2.1	Previous parts end and current parts begin with non-space character
2.2	Previous parts end and current parts begin with space
2.3	Previous parts end with space and current parts begin with non-space character
2.4	Previous parts end with non-space character and current parts begin with space

Since cases from Table 20 and Table 21 are independent, there are total of twelve possibilities. Each one of those possibilities requires an action that does not necessarily have to be unique to that particular combination of cases. Instead of splitting the words on the fly, it is more efficient to add extra spaces where there is no matching word in one text to another and replace a space with null if there should not be any separation at that particular space. The nulls can be changed back to space character after the words have been separated.

Table 22
Combinations of Cases of Table 20 and Table 21 and Corresponding to Them Actions

Combination	Action
1.1 and 2.1	No action
1.1 and 2.2	No action
1.1 and 2.3	No action
1.1 and 2.4	No action
1.2 and 2.1	Set spaces to nulls in the string that has spaces

1.2 and 2.2	Set spaces to nulls in the string that has spaces
1.2 and 2.3	Set spaces to nulls in the string that has spaces
1.2 and 2.4	Set spaces to nulls in the string that has spaces
1.3 and 2.1	Insert N_2-1 spaces at the first space of different part of first text and N_1-1 spaces at the last space of different part of second text
1.3 and 2.2	Insert N_2-1 spaces at the beginning of different part of first text and N_1-1 spaces at the end of different part of second text
1.3 and 2.3	Insert N_2-1 spaces at the beginning of different part of first text and N_1-1 spaces at the last space of different part of second text
1.3 and 2.4	Insert N_2-1 spaces at the first space of different part of first text and N_1-1 spaces at the end of different part of second text

Note: N_1 and N_2 are numbers of spaces in different parts of first and second text respectively.

The basic goal behind each action is to make number of spaces the same for both text documents. After these actions have been applied to the similar and different parts, the texts are merged together and separated using a space as a delimiter. The output of the separation will be a list of aligned words. Some of these words will have nulls that need to be changed back to spaces.

	Previous Similar Parts	Current Different Parts	Current Similar Parts
First Text	"abc def"	"xxx"	"ghi klm"
Second Text	"abc def"	"y z"	"ghi klm"

After actions have been applied

"abc def"	"xxx"	"ghi klm"
"abc def"	"y?z"	"ghi klm"

Merged text

"abc defxxxghi klm"	"abc defy?zghi klm"
---------------------	---------------------

Separated text using space as a delimiter

First Text	Second Text
"abc"	"abc"
"defxxxghi"	"defy zghi"
"klm"	"klm"

Figure 56: Example of Combination of cases 1.2 and 2.1

Figure 56, Figure 57, and Figure 58 demonstrate combinations of cases 1.2 and 2.1, 1.1 and 2.1, and 1.3 and 2.1 respectively. Other cases are handled in similar way with slightly differences according to the set of actions specified in Table 22.

	Previous Similar Parts	Current Different Parts	Current Similar Parts
First Text	"abc def"	"x x"	"ghi klm"
Second Text	"abc def"	"y z"	"ghi klm"

After actions have been applied

"abc def"	"x x"	"ghi klm"
"abc def"	"y z"	"ghi klm"

Merged text

"abc defx xghi klm"	"abc defy zghi klm"
---------------------	---------------------

Separated text using space as a delimiter

First Text	Second Text
"abc"	"abc"
"defx"	"defy"
"xghi"	"zghi"
"klm"	"klm"

Figure 57: Example of Combination of cases 1.1 and 2.1

	Previous Similar Parts	Current Different Parts	Current Similar Parts
First Text	"abc def"	"x x x"	"ghi klm"
Second Text	"abc def"	"y z"	"ghi klm"

After actions have been applied

"abc def"	"x x x"	"ghi klm"
"abc def"	"y z"	"ghi klm"

Merged text

"abc defx x xghi klm" "abc defy zghi klm"

Separated text using space as a delimiter

First Text	Second Text
"abc"	"abc"
"defx"	"defy"
"x"	"z"
"xghi"	"zghi"
"klm"	"klm"

Figure 58: Example of Combination of cases 1.3 and 2.1

Table 23
Sample of Two Aligned Texts

<u>Abby OCR</u>	<u>Omni OCR</u>
""	"a"
""	"a"
"SELLER"	"SELLER"
"AND"	"AND"
"PURCHASER;"	"PURCHASER;"
"/\\"	"(4"
"AGRBE"	"AGREE"
"Purchaser"	"Purchaser"
"will"	"will"

"deposit"	"deposit"
"\$2,320,000.00"	"\$2,320,000.00"
"as"	"as"
"earnest"	"eamest"
"money"	"money"
"pursuant"	"pursuant"
"to"	"to"
"tho"	"the"
<u>"Buy-S&I"</u>	<u>"Buy-fc~I"</u>

Table 23 shows the first 18 entries in the list of two aligned texts (results of Abby's and Omni's OCR engines).

The only one major drawback of this method is that in case 1.2, when one of the different parts does not have any spaces and the other has them, the algorithm merges the words.

Table 24
Examples of Merged Words after Aligned of Two Texts

<u>Abby OCR</u>	<u>Omni OCR</u>
"appropriat>tifnc^clircct"	"appropriatq tifiu; direct"
"transfcrof"	"lransf of"
<u>"xlialUlposit"</u>	<u>"shall tTcposit"</u>

This happens only in the different parts, and the best decision that can be made is to merge the words, since there is no way of telling where the word of the other text needs to be broken apart. However, from the Table 24 it can be concluded that this situation happens most of the time where original image file has been significantly corrupted, and even if there would be

a way to break apart joined words, the voting engine would fail since most of the times the rest of the characters would not be recognized correctly by all three OCR engines.

Alignment of individual words of three texts

Alignment of the three text documents is rather simple. Input to this algorithm would be three lists of aligned pairs of texts; Omni and Abbyy, Abbyy and Doculex, and Doculex and Omni. The out put of the algorithm will be a single list containing elements of the words:

`listThreeTexts(k).stringWord1`

`listThreeTexts(k).stringWord2`

`listThreeTexts(k).stringWord3`

If texts would have the same number of words and no noise, merged are broken apart words, it would simply be a matter of combining element by element. Unfortunately numbers of elements in each pair of aligned documents are different. It is safe, however, to assume that most of the spaces of text will be recognized correctly.

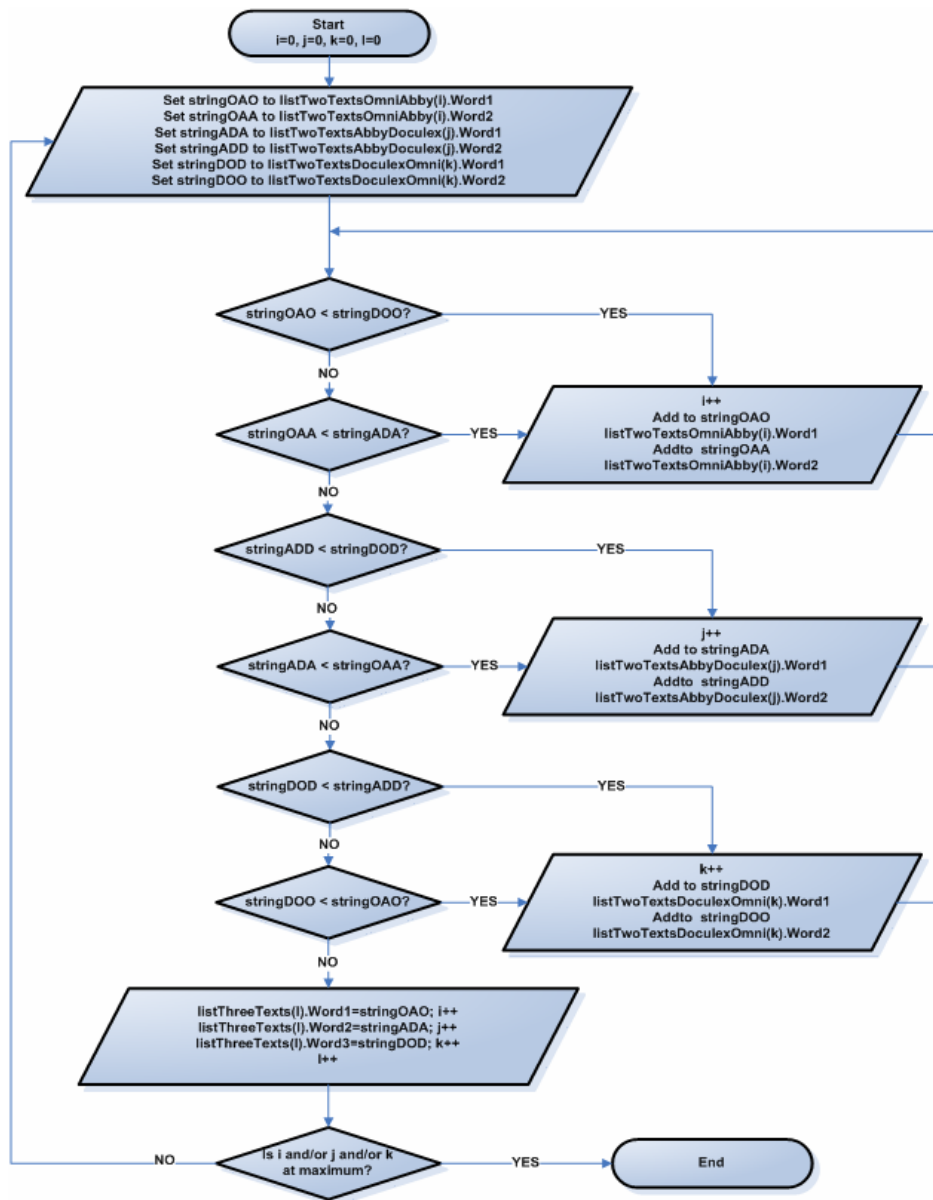


Figure 59: Algorithm for Aligning of Three Texts

Given that the input to the algorithm is a set of three lists listTwoTextsOmniAbby, listTwoTextsAbbyDoculex, listTwoTextsDoculexOmni, the algorithm will be as demonstrated in Figure 59. What the algorithm does is that it creates six temporary strings that hold word or several words from each of the element from each of the three lists. It adds to each word an additional word as needed to even out them in order to get to the point where each of the three

lists agree that these are the aligned parts now stored in these six strings. Out of six strings two correspond to Abby's text, two to Omni's and two to Doculex's. Within each pair strings have to be equal to each other. After the common point is reached, the three unique strings are stored in the output list. Variables i, j, and k are guaranteed to reach their maximum value at the same time.



Figure 60: Example of Aligning of Three Texts

Figure 60 gives an example of how the three texts are aligned. This particular example has four steps. Upper part of the figure shows position within each of the three pairs of aligned texts indicated by blue shading, and lower part shows current values of each of six variables at each of the four steps. In each step, strings that are not equal are shaded. The green shaded string is larger than the orange shaded string. In this case, in each of three pairs of two aligned

texts shift of one has occurred, it is not necessary that they will all have equal shifts. On the fourth step pairs corresponding to the same OCR engine strings are equal, at which point these values will be stored in output list and the algorithm will move on onto the next set of words.

This part of the aligning algorithm will introduce even more merged words. Just like with the algorithm for aligning two texts, this normally happens around parts of the text that OCR engines has hard time processing, which normally indicates that the words would not be recognized correctly anyway.

Table 25
Sample of Three Aligned Texts

Abbyy OCR	Omni OCR	Doculex
""	"a"	"" "
""	"a"	"i"
"SELLER"	"SELLER"	"SELLER"
"AND"	"AND"	"AND"
"PURCHASER;"	"PURCHASER;"	"PURCHASER:"
"/\"	"(4"	"L_ : " ? " : / _ "
"AGRBE"	"AGREE"	"AGREE"
"Purchaser"	"Purchaser"	"Purchaser"
"will"	"will"	"will"

Table 25
Sample of Three Aligned Texts

Abbyy OCR	Omni OCR	Doculex
"deposit"	"deposit"	"deposit"
"\$2,320,000.00"	"\$2,320,000.00"	"\$2,320,000.00"
"as"	"as"	"as"

"earnest"	"eamest"	"earnest"
"money"	"money"	"money"
"pursuant"	"pursuant"	"pursuant"
"to"	"to"	"to"
"tho"	"the"	"the"
"Buy-S&l"	"Buy-fc~l"	"Buy-S I"

Table 25 shows a sample of alignment of three texts, which are results of processing of the hard copy of the text document partially shown in Figure 1 by the three OCR engines.

Performance of Aligning Algorithm

Since the whole purpose of the aligning algorithm is to automate the voting system, the best way to determine whether the algorithm is successful or not is to use it in the complete system and compare the output to the results that were acquired from voting on manually aligned text. For comparison, a below-average quality document was chosen. Comparing the automatically combined text to the manually combined one, one can notice that the person who aligned them was able to separate words that were merged together. Also some noise characters were filtered out based on human perception whether they should or should not be. These two abilities cannot be programmed into an algorithm without some kind of artificial intelligence. This disadvantage caused an approximate 7% merged together or mixed with extra characters text. However, examining more closely, the out put of the voting engine was not correct for almost 7%. Merged text and extra characters would only be introduced in the areas where image quality was poor; which is almost always accompanied by a numerous errors within the single

word. Since the words produced by all three OCR engines had errors, the voting engine at the current state would never be able to generate a correct output. Out of seven hundred manually aligned words four more than the automatically aligned were voted correctly, which is less than 1%.

CHAPTER FOUR: CONCLUSIONS

At this point it is hard to conclude anything about application of morphological filters in text restoration. In a large portion of the cases when the OCR made gruesome errors it is hard even for a human to read the text. There is definitely more research needed in this area, with possibly introducing some kind of artificial intelligence.

The aligning algorithm did an outstanding job of matching words that were generated by the three OCR engines processing the same word in the text image. Even though that manually aligned text looks better 7% of the times for a below-average quality text, the output accuracy of the multi-engine OCR system went down by about 0.5%.

The difference can be reduced or even eliminated by optimizing many different variables, most of which were selected intuitively. Optimization and benchmarking can be done as a future part of future development. Unfortunately, variable and parameters such as thresholds for similarity when looking for next similar character or next similar section of the text, sizes and shapes of masks used for detecting the beginning of next similar part, functions used to determine visual character difference, and a few others do not have a way for analytical selection and have to be determined experimentally. In addition to benchmarking, the algorithm for text aligning can be made adaptive. It has been mentioned that setting various masks and thresholds can introduce conflicts between certain sections of algorithm. Introducing a way by which the algorithm can go back a step, so that the knowledge of the present iteration can be used to set the parameters prior to decision being made. Based on the average length of different parts, the algorithm can start out looking for a next similar part of the text in the larger section text, unlike in this thesis where the starting windows was of the same size as the smallest mask. Also

eliminating or at least reducing problem with word merging, can make it easy expand the alignment algorithm from a three-engine OCR to a N-engine OCR system, by simply running two-text alignment for $N-3$ additional pairs of texts and adding $2(N-3)$ “if” statements for the algorithm that merges two-text alignments into all-text alignment.

A significant improvement in overall voting system performance can be achieved by aligning characters once words have been aligned, which was also mentioned in Sprague’s thesis [3]. However, unlike Sprague recommended to split the image into characters, with tools developed in this thesis for character comparison using visual based techniques, this will not be necessary.

APPENDIX A: SOURCE CODE FOR MORPHOLOGICAL FILTERS

addSaltPepper.m	95
fltMorph.m	96
myAddSalt.m	96
myGenCirMask.m	97
myMorph02.m	97
myMorph.m	99

addSaltPepper.m

```

% This program reads in an image and generates images with salt and peper
% noise noise for each image salt and pepper probabilities are specified by
% pNoise and pSalt

imgIn = double( imread( '..\text\test.jpg', 'jpg' ) );
imgIn = round(imgIn ./ max( max( imgIn ) ));

imgX = size( imgIn, 1 );
imgY = size( imgIn, 2 );

imgNoise = rand( imgX, imgY );

pNoise = .05;
pSalt = .05;
imgOut = ( imgNoise < ( pNoise - pSalt ) ) .* -1 + ( imgNoise > 1 - pSalt );
imgOut = imgOut + imgIn;
imgOut = min( imgOut, 1 );
imgOut = max( imgOut, 0 );
imwrite( imgOut .* 255, '..\text\test_sp005005.jpg' );

if( 0 )
pNoise = .4
pSalt = .2;
imgOut = ( imgNoise < ( pNoise - pSalt ) ) .* -1 + ( imgNoise > 1 - pSalt );
imgOut = imgOut + imgIn;
imgOut = min( imgOut, 1 );
imgOut = max( imgOut, 0 );
imwrite( imgOut .* 255, '..\text\test_sp0402.jpg' );

pNoise = .6
pSalt = .3;
imgOut = ( imgNoise < ( pNoise - pSalt ) ) .* -1 + ( imgNoise > 1 - pSalt );
imgOut = imgOut + imgIn;
imgOut = min( imgOut, 1 );
imgOut = max( imgOut, 0 );
imwrite( imgOut .* 255, '..\text\test_sp0603.jpg' );

pNoise = .2
pSalt = .2;
imgOut = ( imgNoise < ( pNoise - pSalt ) ) .* -1 + ( imgNoise > 1 - pSalt );
imgOut = imgOut + imgIn;
imgOut = min( imgOut, 1 );
imgOut = max( imgOut, 0 );
imwrite( imgOut .* 255, '..\text\test_sp0202.jpg' );

```



```

pNoise = .4
pSalt = .4;
imgOut = ( imgNoise < ( pNoise - pSalt ) ) .* -1 + ( imgNoise > 1 - pSalt );
imgOut = imgOut + imgIn;
imgOut = min( imgOut, 1 );
imgOut = max( imgOut, 0 );
imwrite( imgOut .* 255, '..\text\test_sp0404.jpg' );

pNoise = .6
pSalt = .6;
imgOut = ( imgNoise < ( pNoise - pSalt ) ) .* -1 + ( imgNoise > 1 - pSalt );
imgOut = imgOut + imgIn;
imgOut = min( imgOut, 1 );
imgOut = max( imgOut, 0 );
imwrite( imgOut .* 255, '..\text\test_sp0606.jpg' );

end

```

fltMorph.m

```

% This function realizes dilation/erosion.
% Inputs are image array, mask, and value which will be used as a threshold
% (if lowVal is set to 1 it becomes dilation wrt black color, if lowVal is
% set to sum( sum( myMask ) ) the filter becomes erosion)

function [ imgOut ] = fltMorph( imgIn, myMask, lowVal )

% Image inversion
imgIn = 1 - imgIn;

% Initialization
mskX = size( myMask, 1 );
mskY = size( myMask, 2 );
mskX2 = floor( size( myMask, 1 ) / 2 );
mskY2 = floor( size( myMask, 2 ) / 2 );
imgX = size( imgIn, 1 );
imgY = size( imgIn, 2 );
imgOut( 1:imgX - mskX + 1, 1:imgY - mskY + 1 ) = 0;

% Processing
for k = 1:mskX;
    for l = 1:mskY;
        if( myMask( k, l ) == 1 )
            imgOut = imgOut + imgIn( k:imgX - mskX + k, l:imgY - mskY + l );
        end
    end
end

imgOut = ( imgOut >= lowVal );

imgOut( imgX - mskX + 2:imgX, 1:imgY - mskY + 1 ) = 0;
imgOut( 1:imgX, imgY - mskY + 2:imgY ) = 0;

% Image inversion
imgOut = 1 - imgOut;
% Shift image to center (after filter has been applied the image shifts
% down to the left by half of mask size)
imgOut = circshift( imgOut, [ mskX2 mskY2 ] );

return;

```

myAddSalt.m

```

% This function simply adds salt with probability pSalt to an image array
% and returns new image array

```

```

function [ imgOut ] = myAddSalt( imgIn, pSalt )
imgX = size( imgIn, 1 );
imgY = size( imgIn, 2 );

imgNoise = rand( imgX, imgY );
imgSalt = double( imgNoise > ( 1 - pSalt ) );
imgOut = max( imgIn, imgSalt );

return;

```

myGenCirMask.m

```

% This function generates circle-shaped mask for both even and odd sizes
% Input is length of mask (assumed to be square mask)
% Output is a matrix containing circle-shaped mask

function [ mskOut ] = myGenCirMask( mskSize )

if( mod( mskSize, 2 ) == 1 )
    mskCenter = ceil( ( mskSize ) / 2 );
    mskOut = zeros( mskSize, mskSize );
    for k = 1:mskSize
        for l = 1:mskSize
            if( ( ( k - mskCenter ) ^ 2 + ( l - mskCenter ) ^ 2 ) <= ( mskCenter - 1 ) ^ 2 )
                mskOut( k, l ) = 1;
            end
        end
    end
else
    mskCenter = ceil( ( mskSize ) / 2 );
    mskOut = zeros( mskSize, mskSize );
    for k = 1:mskSize
        for l = 1:mskSize
            if( ( ( k - mskCenter - .5 ) ^ 2 + ( l - mskCenter - .5 ) ^ 2 ) <= ( mskCenter ) ^ 2 )
                mskOut( k, l ) = 1;
            end
        end
    end
end

return;

```

myMorph02.m

```

% This filter is a combination of dilation and erosion (opening/closing filters)
% Inputs are original image array, mask matrix, and processing sequence
% (last one not used at this time)
% Output is an filtered image array
function [ imgOut ] = myMorph02( imgIn, myMask, procSequence )

hiMask = sum( sum( myMask ) );
loMask = 1;

imgOut = fltMorph( imgIn, myMask, loMask );
imgOut = fltMorph( imgOut, myMask, hiMask );

```

myMorph.m

```
% This filter is a combination of dilation and erosion (opening/ closing filters)
% It opens ainput image and saves output to a file
function [ imgInName ] = myMorph( imgInName, strLoadPath, strSavePath, strImageType )

myMask = [ 0 0 0; 1 1 1; 0 0 0 ];

hiMask = sum( sum( myMask ) );
loMask = 1;

imgIn = double( imread( [ strLoadPath imgInName '.' strImageType ], strImageType ) );
imgIn = round( imgIn ./ max( max( imgIn ) ) );
imgOut = fltMorph( imgIn, myMask, loMask );
imgOut = fltMorph( imgOut, myMask, hiMask );
imwrite( imgOut * 255, [ strSavePath imgInName '_out_cls.' strImageType ] );
```

APPENDIX B: SOURCE CODE FOR TEXT ALIGNMENT ALGORITHM

AlignText.m	100
AlignTexts01a.m	101
AlignWords02.m	103
compareASCIIv4.m	105
findNextChar02a.m	107
findNextPart02a.m	108
fixlines.m	110
myTrimSpaces.m	111

AlignText.m

```

% This is the main function that loads 3 text files, alignes them, and
% stores the output in lstText as described in thesis.
% Text files must meet following requirements
%   - no white space characters except space character
%   - no mora than one space character in the row, except
%   - at the end same number of characters must be inserted as the
%     size of the starting find-next-part window plus two
%   - no characters that are not in similarity look-up table are allowed

% Output to screen major steps completion (should also be passed as an
% argument in the future)
blnShowProgress = 1;

% Starting timer
tic;

if( blnShowProgress ) disp( 'Staring text alignment ...' ); end;

% Opening files (In the final version names of the files would be passed as
% arguments)
inID1 = fopen( 'input\abby_test.txt' );
inID2 = fopen( 'input\omni_test.txt' );
inID3 = fopen( 'input\doculex_test.txt' );

% Loading predefined visula character similarity lookup table
errASCII = importdata( 'data\errASCIIv3ed3x3.mat' );
% Loading automatically pregenerated stick-like representation of
% characters
lstChars = importdata( 'data\lstChars.mat' );
% This function is a programmed manual adjustment of stick-like
% representation of characters
fixlines;

% Reading input files
inText1 = fread( inID1 );
inText2 = fread( inID2 );
inText3 = fread( inID3 );

```

```

if( blnShowProgress ) disp( 'Reading of data complete.' ); end;

% Setting characters out of range to visually common character 124 (any other
% than space character can be used). Ideally, source code for checking and
% fixing files according to requirements listed at the top of the document
% would be here.
inText1( find( ( inText1 > 126 ) + ( inText1 < 32 ) ) ) = 124;
inText2( find( ( inText2 > 126 ) + ( inText2 < 32 ) ) ) = 124;
inText3( find( ( inText3 > 126 ) + ( inText3 < 32 ) ) ) = 124;

if( blnShowProgress ) disp( 'Text clean-up is complete.' ); end;

% Closing input files
fclose( inID1 );
fclose( inID2 );
fclose( inID3 );

% Aligning sections of texts of three pairs of the texts (ORDER IS IMPORTANT)
[intWordsFound12, lstWords12] = compareASCIIV4( inText1, inText2, 3, 6, errASCII, lstChars );
if( blnShowProgress ) disp( 'Alignment of sections of text of first pair is complete.' ); end;
[intWordsFound23, lstWords23] = compareASCIIV4( inText2, inText3, 3, 6, errASCII, lstChars );
if( blnShowProgress ) disp( 'Alignment of sections of text of second pair is complete.' ); end;
[intWordsFound31, lstWords31] = compareASCIIV4( inText3, inText1, 3, 6, errASCII, lstChars );
if( blnShowProgress ) disp( 'Alignment of sections of text of third pair is complete.' ); end;

% Splitting up similar and not similar texts into words for each part of
% aligned text
lstTable12 = AlignWords02( lstWords12 );
if( blnShowProgress ) disp( 'First pair of aligned text is divided into words.' ); end;
lstTable23 = AlignWords02( lstWords23 );
if( blnShowProgress ) disp( 'Second pair of aligned text is divided into words.' ); end;
lstTable31 = AlignWords02( lstWords31 );
if( blnShowProgress ) disp( 'Third pair of aligned text is divided into words.' ); end;

% Final Alignment
lstText = AlignTexts01a( lstTable12, lstTable23, lstTable31 );
if( blnShowProgress ) disp( 'Text has been aligned.' ); end;

% Generating output file (in final version name of the output file will be
% passed to this function as an argument)
outID = fopen( 'output\alignedtext01.txt', 'w' );
intWords = size( lstText, 2 );
for k = 1:intWords
    fwrite( outID, sprintf( '%s\t%s\t%s\n', lstText( k ).Word1, lstText( k ).Word2, lstText( k
).Word3 ) );
end
fclose( outID );
if( blnShowProgress ) disp( 'Output file has been created.' ); end;

% Stopping timer.
time = toc;
if( blnShowProgress ) disp( sprintf( 'Text aligned is complete (elapsed time: %d:%d).', floor(
time / 60 ), round( time - floor( time / 60 ) * 60 ) ) ); end;

```

AlignTexts01a.m

```

% This function aligns the 3 texts together
% Input is a set of three lists that contain 3 pairs of texts with aligned
% words
% Output is a single list that contains the 3 aligned texts

function [ AlignedTexts ] = AlignTexts01a( lstTable12, lstTable23, lstTable31 )

% Setting up initial variables

intTable12 = size( lstTable12, 2 );

```

```

intTable23 = size( lstTable23, 2 );
intTable31 = size( lstTable31, 2 );

intCurrWord = 1;
intCurrWord12 = 1;
intCurrWord23 = 1;
intCurrWord31 = 1;
lstText = 0; clear lstText;
lstText( 1 ).Word1 = '';
lstText( 1 ).Word2 = '';
lstText( 1 ).Word3 = '';

intCount = 1;

Word12_1 = lstTable12( intCurrWord12 ).Word1;
Word12_2 = lstTable12( intCurrWord12 ).Word2;
Word23_1 = lstTable23( intCurrWord23 ).Word1;
Word23_2 = lstTable23( intCurrWord23 ).Word2;
Word31_1 = lstTable31( intCurrWord31 ).Word1;
Word31_2 = lstTable31( intCurrWord31 ).Word2;

while( intTable12 > intCurrWord12 && intTable23 > intCurrWord23 && intTable31 > intCurrWord31 )
% Since words could have been merged by one or more OCR engines, the only
% way to comeup with a single word that is present in all three rexrs is to
% combine several words together until lenght of each combined word in each
% text is the same
    if( strcmp( Word12_1, Word31_2 ) && strcmp( Word12_2, Word23_1 ) && strcmp( Word23_2,
Word31_1 ) )
        lstText( intCurrWord ).Word1 = Word12_1;
        lstText( intCurrWord ).Word2 = Word12_2;
        lstText( intCurrWord ).Word3 = Word23_2;
        intCurrWord12 = intCurrWord12 + 1;
        intCurrWord23 = intCurrWord23 + 1;
        intCurrWord31 = intCurrWord31 + 1;
        intCurrWord = intCurrWord + 1;
        Word12_1 = lstTable12( intCurrWord12 ).Word1;
        Word12_2 = lstTable12( intCurrWord12 ).Word2;
        Word23_1 = lstTable23( intCurrWord23 ).Word1;
        Word23_2 = lstTable23( intCurrWord23 ).Word2;
        Word31_1 = lstTable31( intCurrWord31 ).Word1;
        Word31_2 = lstTable31( intCurrWord31 ).Word2;
    end
% IF statements below add next word to shortes pair of combined words
    if( size( Word12_1, 2 ) > size( Word31_2, 2 ) )
        intCurrWord31 = intCurrWord31 + 1;
        Word31_1 = [ Word31_1 ' ' lstTable31( intCurrWord31 ).Word1 ];
        Word31_2 = [ Word31_2 ' ' lstTable31( intCurrWord31 ).Word2 ];
        Word31_1 = myTrimSpaces( Word31_1 );
        Word31_2 = myTrimSpaces( Word31_2 );
    end

    if( size( Word12_1, 2 ) < size( Word31_2, 2 ) )
        intCurrWord12 = intCurrWord12 + 1;
        Word12_1 = [ Word12_1 ' ' lstTable12( intCurrWord12 ).Word1 ];
        Word12_2 = [ Word12_2 ' ' lstTable12( intCurrWord12 ).Word2 ];
        Word12_1 = myTrimSpaces( Word12_1 );
        Word12_2 = myTrimSpaces( Word12_2 );
    end

    if( size( Word12_2, 2 ) > size( Word23_1, 2 ) )
        intCurrWord23 = intCurrWord23 + 1;
        Word23_1 = [ Word23_1 ' ' lstTable23( intCurrWord23 ).Word1 ];
        Word23_2 = [ Word23_2 ' ' lstTable23( intCurrWord23 ).Word2 ];
        Word23_1 = myTrimSpaces( Word23_1 );
        Word23_2 = myTrimSpaces( Word23_2 );
    end

    if( size( Word12_2, 2 ) < size( Word23_1, 2 ) )
        intCurrWord12 = intCurrWord12 + 1;

```

```

        Word12_1 = [ Word12_1 ' ' lstTable12( intCurrWord12 ).Word1 ];
        Word12_2 = [ Word12_2 ' ' lstTable12( intCurrWord12 ).Word2 ];
        Word12_1 = myTrimSpaces( Word12_1 );
        Word12_2 = myTrimSpaces( Word12_2 );
    end

    if( size( Word23_2, 2 ) > size( Word31_1, 2 ) )
        intCurrWord31 = intCurrWord31 + 1;
        Word31_1 = [ Word31_1 ' ' lstTable31( intCurrWord31 ).Word1 ];
        Word31_2 = [ Word31_2 ' ' lstTable31( intCurrWord31 ).Word2 ];
        Word31_1 = myTrimSpaces( Word31_1 );
        Word31_2 = myTrimSpaces( Word31_2 );
    end

    if( size( Word23_2, 2 ) < size( Word31_1, 2 ) )
        intCurrWord23 = intCurrWord23 + 1;
        Word23_1 = [ Word23_1 ' ' lstTable23( intCurrWord23 ).Word1 ];
        Word23_2 = [ Word23_2 ' ' lstTable23( intCurrWord23 ).Word2 ];
        Word23_1 = myTrimSpaces( Word23_1 );
        Word23_2 = myTrimSpaces( Word23_2 );
    end

    intCount = intCount + 1;
end

% Return list
AlignedTexts = lstText;

```

AlignWords02.m

```

% This function splits similar parts into words
% Input is a list of similar parts
% Outputs is a list of aligned words
function [ lstTable ] = AlignWords02( lstWords )

intWords = size( lstWords, 2 );
strText1 = '';
strText2 = '';

% Code below removes extra space characters at the end of each last word of
% each list
while( ~isempty( lstWords( intWords ).simWord1 ) && lstWords( intWords ).simWord1( size(
lstWords( intWords ).simWord1 ,2 ) ) == 32 )
    lstWords( intWords ).simWord1 = lstWords( intWords ).simWord1( 1:size( lstWords( intWords
).simWord1 ,2 ) - 1 );
end
while( ~isempty( lstWords( intWords ).simWord2 ) && lstWords( intWords ).simWord2( size(
lstWords( intWords ).simWord2 ,2 ) ) == 32 )
    lstWords( intWords ).simWord2 = lstWords( intWords ).simWord2( 1:size( lstWords( intWords
).simWord2 ,2 ) - 1 );
end

% Code below ads several space characters to the end of last word so that
% the last words of each list would have the same lenght. This is done to
% simplify (mainly eliminate having to deal with last word separately)
% the code.
intSpaces1 = sum( double( lstWords( intWords ).simWord1 == 32 ) );
intSpaces2 = sum( double( lstWords( intWords ).simWord2 == 32 ) );
if( intSpaces1 ~= intSpaces2 )
    lstWords( intWords ).simWord1 = [ lstWords( intWords ).simWord1 ones( 1, max( 0, intSpaces2 -
intSpaces1 ) ) .* 32 ];
    lstWords( intWords ).simWord2 = [ lstWords( intWords ).simWord2 ones( 1, max( 0, intSpaces1 -
intSpaces2 ) ) .* 32 ];
end

```



```

for k = 1:intWords %!!!!DEAL WITH LAST WORD!!!!
% First word is treaky, the simplest way to deal with it is to add a dummy character
% followed by space (i.e. "a ") at the beginning of each one of the input texts)

% "sum" function is used to calculate number of spaces in the text. Since
% sum of an empty set will generate an error this case needs to be treated
% as an exception
    blnDif1Empty = isempty( lstWords( k ).difWord1 );
    if( ~blnDif1Empty )
        intDif1Spaces = sum( double( lstWords( k ).difWord1 == 32 ) );
    else
        intDif1Spaces = 0;
    end
    blnDif2Empty = isempty( lstWords( k ).difWord2 );
    if( ~blnDif2Empty )
        intDif2Spaces = sum( double( lstWords( k ).difWord2 == 32 ) );
    else
        intDif2Spaces = 0;
    end
% Below is an implimitation of rools listed in the thesis. Space character
% that did not match between the two texts are going to be set to 1 since
% this value is not used by any other character
    if( intDif2Spaces == intDif1Spaces )
        strText1 = [ strText1 lstWords( k ).difWord1 ];
        strText2 = [ strText2 lstWords( k ).difWord2 ];
    elseif( ( intDif2Spaces == 0 && intDif1Spaces >= 1 ) || ( intDif1Spaces == 0 && intDif2Spaces
>= 1 ) )
        if( k ~= 1 )
            lstWords( k ).difWord1( find( lstWords( k ).difWord1 == 32 ) ) = 1;
            lstWords( k ).difWord2( find( lstWords( k ).difWord2 == 32 ) ) = 1;
        else
            lstWords( k ).difWord1 = [ ones( 1, intDif2Spaces ) .* 32 lstWords( k ).difWord1 ];
            lstWords( k ).difWord2 = [ ones( 1, intDif1Spaces ) .* 32 lstWords( k ).difWord2 ];
        end
        strText1 = [ strText1 lstWords( k ).difWord1 ];
        strText2 = [ strText2 lstWords( k ).difWord2 ];
    else
        intSpaces1 = sum( double( lstWords( k ).difWord1 == 32 ) ) - 1;
        intSpaces2 = sum( double( lstWords( k ).difWord2 == 32 ) ) - 1;
        intSpace1 = min( find( lstWords( k ).difWord1 == 32 ) );
        intSpace2 = max( find( lstWords( k ).difWord2 == 32 ) );
        intSize1 = size( lstWords( k ).difWord1, 2 );
        intSize2 = size( lstWords( k ).difWord2, 2 );
        lstWords( k ).difWord1 = [ lstWords( k ).difWord1( 1:intSpace1 ) ones( 1, intSpaces2 ) .*
32 lstWords( k ).difWord1( intSpace1 + 1:intSize1 ) ];
        lstWords( k ).difWord2 = [ lstWords( k ).difWord2( 1:intSpace2 ) ones( 1, intSpaces1 ) .*
32 lstWords( k ).difWord2( intSpace2 + 1:intSize2 ) ];
        strText1 = [ strText1 lstWords( k ).difWord1 ];
        strText2 = [ strText2 lstWords( k ).difWord2 ];
    end

    strText1 = [ strText1 lstWords( k ).simWord1 ];
    strText2 = [ strText2 lstWords( k ).simWord2 ];

end

% After the nonmatching spaces have been removed number of spaces that will
% be used as a delimiter must be the same
if( sum( double( strText1 == 32 ) ) ~= sum( double( strText2 == 32 ) ) )
    disp( 'ERROR: Number of spaces do not match.' );
    return;
end

lstTable( 1 ).Word1 = '';
lstTable( 1 ).Word2 = '';

% Below are two loops that split entire text into a list using space

```

```

% character as a delimiter
intCurrWord = 1;
numChars = size( strText1, 2 );
for k = 1:numChars
    if( strText1( k ) == 32 )
        intCurrWord = intCurrWord + 1;
        lstTable( intCurrWord ).Word1 = '';
    else
        lstTable( intCurrWord ).Word1 = [ lstTable( intCurrWord ).Word1 strText1( k ) ];
    end
end

intCurrWord = 1;
numChars = size( strText2, 2 );
for k = 1:numChars
    if( strText2( k ) == 32 )
        intCurrWord = intCurrWord + 1;
        lstTable( intCurrWord ).Word2 = '';
    else
        lstTable( intCurrWord ).Word2 = [ lstTable( intCurrWord ).Word2 strText2( k ) ];
    end
end

% Now that there is no need for delimiting spaces, unmatched spaces that
% have been set to 1 can be changed back to 32
intRows = size( lstTable, 2 );
for k = 1 : intRows
    lstTable( k ).Word1( find( lstTable( k ).Word1 == 1 ) ) = 32;
    lstTable( k ).Word2( find( lstTable( k ).Word2 == 1 ) ) = 32;
end

```

compareASCIIv4.m

```

% This function finds similar parts in two text documents
% inputs: two text documents, number of characters that will be used for
% the first attempt to find next similar part, number of characters that
% will be passed to findNextChar function, array of character comparison
% table, and list that contains stick-like representation of the characters
% Output is a list of series of smatched and unmatched text.

function [ intWordsFound, lstWords ] = compareASCIIv4( inText1, inText2, intNumComp,
intNumCompNext, errASCII, lstChars )

% Setting up initial values

intWordsFound = 1;
intState = 1;
intPosition1 = 1;
intPosition2 = 1;
intGoBack1 = 1;
intGoBack2 = 1;
intTextLen1 = size( inText1, 1 );
intTextLen2 = size( inText2, 1 );
intFound = 0;
intShift1 = 0;
intShift2 = 0;
intMoveNext = 1;
intProcessing = 1;

lstWords( 1 ).difWord1 = '';
lstWords( 1 ).difWord2 = '';
lstWords( 1 ).simWord1 = '';
lstWords( 1 ).simWord2 = '';

% Code below is an implementation of alignment algorithm described in the
% thesis. The same numbering for states is used as in flow chart.

```

```

while( intProcessing == 1 )
  if( intState == 1 )
    [ intFound, intShift1, intShift2, intSimChunk ] = findNextPart02a(
inText1(intPosition1+1:intPosition1 - 1 + intMoveNext + intNumComp ),
inText2(intPosition2+1:intPosition2 - 1 + intMoveNext + intNumComp ), errASCII, lstChars);
    if( intFound == 1 )
      intState = 2;
    else
      intState = 6;
    end
    continue;
  end
  if( intState == 6 )
    intMoveNext = intMoveNext + 1;
    intState = 8;
    continue;
  end
  if( intState == 2 )
    intPosition1 = intPosition1 + intShift1;
    intPosition2 = intPosition2 + intShift2;
    lstWords( intWordsFound ).difWord1 = char( reshape( inText1( intGoBack1:intPosition1 - 1
+ 1 ), 1, [ ] ) );
    lstWords( intWordsFound ).difWord2 = char( reshape( inText2( intGoBack2:intPosition2 - 1
+ 1 ), 1, [ ] ) );
    intGoBack1 = intPosition1+1;
    intGoBack2 = intPosition2+1;
    intPosition1 = intPosition1 + intSimChunk - 1;
    intPosition2 = intPosition2 + intSimChunk - 1;
    intMoveNext = 0;
    intState = 3;
    continue;
  end
  if( intState == 3 )
    [ intFound, intShift1, intShift2 ] = findNextChar02a( inText1(intPosition1:intPosition1 +
intNumCompNext ), inText2(intPosition2:intPosition2 + intNumCompNext ), errASCII, lstChars );
    if( intFound == 1 )
      intState = 4;
    else
      intState = 5;
    end
    continue;
  end
  if( intState == 4 )
    intPosition1 = intPosition1 + intShift1;
    intPosition2 = intPosition2 + intShift2;
    intState = 7;
    continue;
  end
  if( intState == 5 )
    lstWords( intWordsFound ).simWord1 = char( reshape( inText1( intGoBack1:intPosition1 ),
1, [ ] ) );
    lstWords( intWordsFound ).simWord2 = char( reshape( inText2( intGoBack2:intPosition2 ),
1, [ ] ) );
    intGoBack1 = intPosition1 + 1;
    intGoBack2 = intPosition2 + 1;
    intMoveNext = 1;
    intState = 1;
    intWordsFound = intWordsFound + 1;
    continue;
  end
  if( intState == 7 )
    if( ( intPosition1 + intNumCompNext <= intTextLen1 ) && ( intPosition2 + intNumCompNext
<= intTextLen2 ) )
      intState = 3;
    else
      intState = 9;
    end
    continue;
  end
end

```

```

    if( intState == 8 )
        if( ( intPosition1 - 1 + intMoveNext + intNumComp <= intTextLen1 ) && ( intPosition2 - 1
+ intMoveNext + intNumComp <= intTextLen2 ) )
            intState = 1;
        else
            intState = 10;
        end
        continue;
    end
    if( intState == 9 )
        lstWords( intWordsFound ).simWord1 = char( reshape( inText1( intGoBack1:intTextLen1 ), 1,
[] ) );
        lstWords( intWordsFound ).simWord2 = char( reshape( inText2( intGoBack2:intTextLen2 ), 1,
[] ) );
        intState = 11;
        continue;
    end
    if( intState == 10 )
        lstWords( intWordsFound ).difWord1 = char( reshape( inText1( intGoBack1:intTextLen1 ), 1,
[] ) );
        lstWords( intWordsFound ).difWord2 = char( reshape( inText2( intGoBack2:intTextLen2 ), 1,
[] ) );
        intState = 11;
        continue;
    end
    if( intState == 11 )
        intProcessing = 0;
        intState = 12;
        continue;
    end
end
end
return;

```

findNextChar02a.m

```

% This function checks if next character or next set of characters is
% similar
% Inputs: two strings, character similarity table, and list of stick-like
% representations of characters
% Output: Success flag, shift (length) of similar characters part

function [ intFound, intShift1, intShift2 ] = findNextChar02a( inText1, inText2, errASCII,
lstChars )

% Initialization
intFound = 0;
intShift1 = -1;
intShift2 = -1;
% Best value could be determined by benchmarking
dblMaxDiff1 = .2;
intNumSticks = 4;

% Increasing number of masks could speed up the alignment process. Masks
% from findNextPart could also be used here
lstCompare( 1 ).mtxMask = [ 1 ];
lstCompare( 2 ).mtxMask = [ 0 1 1 ];

intCompares = size( lstCompare, 2 );
intStrLen = size( inText1, 1 );

% Comparing using masks
for k = 1 : intCompares
    intChars = size( lstCompare( k ).mtxMask, 2 );
    if( intStrLen > intChars + 1 )
        dblMaxCurrDiff = 0;
    end
end

```

```

        for m = 1 : intChars
            dblMaxCurrDiff = max( dblMaxCurrDiff, errASCII( inText1( 1 + m ) - 31, inText2( 1 + m
) - 31 ) * lstCompare( k ).mtxMask( m ) );
            if( ( xor( inText1( 1 + m ) == 32, inText2( 1 + m ) == 32 ) ) )
                dblMaxCurrDiff = 1;
            end
        end
        if( dblMaxCurrDiff <= dblMaxDiff1 )
            intFound = 1;
            intShift1 = intChars;
            intShift2 = intChars;
            return;
        end
    end
end

% Comparing using stick like representation of characters. No masks needed
% here

%return; %!!!!!!!!!!!!!!!!!!!!!! If a error occurs uncomment "return;" and see if that helps
nextChars1 = double( inText1( 2:intStrLen ) ) - 31;
nextChars2 = double( inText2( 2:intStrLen ) ) - 31;
for k = 1 : intNumSticks
    for m = 1 : intNumSticks
        if( sum( nextChars1( 1:k ) == 1 ) == 0 && sum( nextChars2( 1:m ) == 1 ) == 0 )
            nextLines1 = [ lstChars( nextChars1( 1:k ) ).lines ];
            nextLines2 = [ lstChars( nextChars2( 1:m ) ).lines ];
            if( size( nextLines1, 2 ) == size( nextLines2, 2 ) )
                if( sum( nextLines1 == nextLines2 ) == size( nextLines2, 2 ) )
                    intFound = 1;
                    intShift1 = k;
                    intShift2 = m;
                    return;
                end
            end
        end
    end
end
end
end
end

```

findNextPart02a.m

```

% This function looks for next similar part
% Inputs: two strings, character similarity table, and list of stick-like
% representations of characters
% Output: Success flag, shift (length) to next similar characters part for
% each text

function [ intFound, intShift1, intShift2, intSimChunk ] = findNextPart02a( inText1, inText2,
errASCII, lstChars )

% Initialization

    dblMaxDiff = .0000001;
    dblMaxDiff = 0;
    intShift1 = -1;
    intShift2 = -1;
    intFound = 0;
    intSimChunk = 0;
    intMaxChunk = 60;
    intMinOverlap = 3;
    intStrLen = size( inText1, 1 );
% If maximum size of a window has been reached this function will terminate
    if( intStrLen == intMaxChunk )
        disp( char( inText1' ) );
        disp( char( inText2' ) );
        disp( 'No further aligning possible' );
    end

```

```

        return;
    elseif( intStrLen > intMaxChunk )
        return;
    end
% Code below realizes matrix-like approach for finding next similar parts

% Building initial matrix
inInterText1 = [ zeros( intStrLen - intMinOverlap, 1 ); inText1; zeros( intStrLen -
intMinOverlap, 1 ) ];
intMatrixY = 2 * intStrLen - 2 * intMinOverlap + 1;
mtxCompareMatrix = zeros( intStrLen, intMatrixY );
mtxCompareMatrixMaskSize = zeros( intStrLen, intMatrixY );

for k = 1 : intMatrixY
    mtxCompareMatrix( :, k ) = inInterText1( k : k + intStrLen - 1 );
end
% Replacing characters in the matrix with similarity value
for k = 1 : intMatrixY
    for l = 1 : intStrLen
        if( mtxCompareMatrix( l, k ) == 0 )
            mtxCompareMatrix( l, k ) = 1;
        elseif( xor( mtxCompareMatrix( l, k ) == 32, inText2( l ) == 32 ) )
            mtxCompareMatrix( l, k ) = 2;
        else
            mtxCompareMatrix( l, k ) = errASCII( mtxCompareMatrix( l, k ) - 31, inText2( l )
- 31);
        end
    end
end

% Setting up masks. Performance of the alignment function can be improved
% by using more INDEPENDANT masks
lstMatchThis( 1 ).mtxMask = [ 1 1 1 ];
lstMatchThis( 2 ).mtxMask = [ 1 0 1 1 ];
lstMatchThis( 3 ).mtxMask = [ 1 1 0 1 ];
intMatchMasts = size( lstMatchThis, 2 );
% Matrices with best matches and corresponding to them mask are created
% below
for k = 1 : intMatrixY
    for l = 1 : intStrLen - intMinOverlap + 1;
        dblSmallestMax = 1;
        for m = 1 : intMatchMasts
            intCurrMaskSize = size( lstMatchThis( m ).mtxMask, 2 );
            if( intStrLen - l + 1 >= intCurrMaskSize && sum( mtxCompareMatrix( 1 : l +
intCurrMaskSize - 1, k ) == 2 ) == 0 )
                dblCurrMax = max( lstMatchThis( m ).mtxMask' .* mtxCompareMatrix( 1 : l +
intCurrMaskSize - 1, k ) );
            elseif( intStrLen - l + 1 >= intCurrMaskSize && sum( mtxCompareMatrix( 1 : l +
intCurrMaskSize - 1, k ) == 2 ) ~= 0 )
                dblCurrMax = 1;
            end
            if( dblCurrMax < dblSmallestMax)
                dblSmallestMax = dblCurrMax;
                mtxCompareMatrixMaskSize( l, k ) = intCurrMaskSize;
            end
        end
        mtxCompareMatrix( l, k ) = dblSmallestMax;
    end
end

mtxCompareMatrix( intStrLen - intMinOverlap + 2 : intStrLen, : ) = 1;

mtxCompareMatrix = double( mtxCompareMatrix <= dblMaxDiff );
% Nearest match is selected. This will be more usefull if/when adaptive window
% size will be implemented
intMaxShift = intMatrixY ^ 2 + ( intStrLen - intMinOverlap + 1 ) ^ 2;
for k = 1 : intMatrixY
    for l = 1 : intStrLen - intMinOverlap + 1;
        if( mtxCompareMatrix( l, k ) == 1 )

```

```

        intTempShift1 = - intStrLen + intMinOverlap + k + 1 - 1 - 1;
        intTempShift2 = 1 - 1;
        if( intMaxShift > intTempShift1 ^ 2 + intTempShift2 ^ 2 )
            intFound = 1;
            intShift1 = intTempShift1;
            intShift2 = intTempShift2;
            intSimChunk = mtxCompareMatrixMaskSize( l, k );
            intMaxShift = intTempShift1 ^ 2 + intTempShift2 ^ 2;
        end
    end
end
end
return;

```

fixlines.m

```

% This code simply redefines some (all in this case) line-like
% representations of characters

```

```

lstChars( 1 ).lines = [ 0 0 ];
lstChars( 2 ).lines = [ 0 2 ];
lstChars( 3 ).lines = [ 0 0 ];
lstChars( 4 ).lines = [ 0 1 0 1 ];
lstChars( 5 ).lines = [ 0 2 1 2 ];
lstChars( 6 ).lines = [ 0 2 1 2 ];
lstChars( 7 ).lines = [ 0 2 1 2 ];
lstChars( 8 ).lines = [ 0 0 ];
lstChars( 9 ).lines = [ 0 2 ];
lstChars( 10 ).lines = [ 0 2 ];
lstChars( 11 ).lines = [ 0 0 ];
lstChars( 12 ).lines = [ 0 0 1 0 ];
lstChars( 13 ).lines = [ 0 0 ];
lstChars( 14 ).lines = [ 0 0 ];
lstChars( 15 ).lines = [ 0 0 ];
lstChars( 16 ).lines = [ 0 1 ];
lstChars( 17 ).lines = [ 0 2 0 2 ];
lstChars( 18 ).lines = [ 0 2 ];
lstChars( 19 ).lines = [ 0 2 1 2 ];
lstChars( 20 ).lines = [ 0 1 0 2 ];
lstChars( 21 ).lines = [ 0 1 0 2 ];
lstChars( 22 ).lines = [ 0 2 1 2 ];
lstChars( 23 ).lines = [ 0 2 0 2 ];
lstChars( 24 ).lines = [ 0 1 2 ];
lstChars( 25 ).lines = [ 0 2 1 2 ];
lstChars( 26 ).lines = [ 0 2 1 2 ];
lstChars( 27 ).lines = [ 0 1 ];
lstChars( 28 ).lines = [ 0 1 ];
lstChars( 29 ).lines = [ 0 0 ];
lstChars( 30 ).lines = [ 0 0 ];
lstChars( 31 ).lines = [ 0 0 ];
lstChars( 32 ).lines = [ 0 1 2 ];
lstChars( 33 ).lines = [ 0 2 1 2 ];
lstChars( 34 ).lines = [ 0 2 1 2 ];
lstChars( 35 ).lines = [ 0 2 0 2 ];
lstChars( 36 ).lines = [ 0 2 0 1 ];
lstChars( 37 ).lines = [ 0 2 0 2 ];
lstChars( 38 ).lines = [ 0 2 0 1 ];
lstChars( 39 ).lines = [ 0 2 0 1 ];
lstChars( 40 ).lines = [ 0 2 0 2 ];
lstChars( 41 ).lines = [ 0 2 0 2 ];
lstChars( 42 ).lines = [ 0 2 ];
lstChars( 43 ).lines = [ 0 1 0 2 ];
lstChars( 44 ).lines = [ 0 2 0 2 ];
lstChars( 45 ).lines = [ 0 2 ];
lstChars( 46 ).lines = [ 0 2 0 2 0 2 ];
lstChars( 47 ).lines = [ 0 2 0 2 ];

```

```

lstChars( 48 ).lines = [ 0 2 0 2 ];
lstChars( 49 ).lines = [ 0 2 0 1 ];
lstChars( 50 ).lines = [ 0 2 0 2 ];
lstChars( 51 ).lines = [ 0 2 0 2 ];
lstChars( 52 ).lines = [ 0 2 1 2 ];
lstChars( 53 ).lines = [ 0 2 ];
lstChars( 54 ).lines = [ 0 2 0 2 ];
lstChars( 55 ).lines = [ 0 2 0 2 ];
lstChars( 56 ).lines = [ 0 2 0 2 0 2 ];
lstChars( 57 ).lines = [ 0 2 1 2 ];
lstChars( 58 ).lines = [ 0 1 2 1 ];
lstChars( 59 ).lines = [ 0 1 2 1 ];
lstChars( 60 ).lines = [ 0 2 ];
lstChars( 61 ).lines = [ 0 2 ];
lstChars( 62 ).lines = [ 0 2 ];
lstChars( 63 ).lines = [ 0 0 0 ];
lstChars( 64 ).lines = [ 0 0 0 ];
lstChars( 65 ).lines = [ 0 0 ];
lstChars( 66 ).lines = [ 0 1 0 1 ];
lstChars( 67 ).lines = [ 0 2 0 1 ];
lstChars( 68 ).lines = [ 0 1 0 1 ];
lstChars( 69 ).lines = [ 0 1 0 2 ];
lstChars( 70 ).lines = [ 0 1 0 1 ];
lstChars( 71 ).lines = [ 0 2 ];
lstChars( 72 ).lines = [ 0 1 0 2 ];
lstChars( 73 ).lines = [ 0 2 0 1 ];
lstChars( 74 ).lines = [ 0 1 ];
lstChars( 75 ).lines = [ 0 2 ];
lstChars( 76 ).lines = [ 0 2 0 1 ];
lstChars( 77 ).lines = [ 0 2 ];
lstChars( 78 ).lines = [ 0 1 0 1 0 1 ];
lstChars( 79 ).lines = [ 0 1 0 1 ];
lstChars( 80 ).lines = [ 0 1 0 1 ];
lstChars( 81 ).lines = [ 0 2 0 1 ];
lstChars( 82 ).lines = [ 0 1 0 2 ];
lstChars( 83 ).lines = [ 0 1 ];
lstChars( 84 ).lines = [ 0 1 0 1 ];
lstChars( 85 ).lines = [ 0 2 ];
lstChars( 86 ).lines = [ 0 1 0 1 ];
lstChars( 87 ).lines = [ 0 1 0 1 ];
lstChars( 88 ).lines = [ 0 1 0 1 0 1 ];
lstChars( 89 ).lines = [ 0 1 0 1 ];
lstChars( 90 ).lines = [ 0 1 0 1 ];
lstChars( 91 ).lines = [ 0 1 1 ];
lstChars( 92 ).lines = [ 0 2 ];
lstChars( 93 ).lines = [ 0 2 ];
lstChars( 94 ).lines = [ 0 2 ];
lstChars( 95 ).lines = [ 0 0 ];

```

myTrimSpaces.m

```

% This function trims space characters of a string. It can handle anything
% passed to it.
% Input and output are strings

```

```

function [ strOut ] = myTrimSpaces( strIn )

    strOut = strIn;
    intSize = size( strIn, 2 );
    if( sum( double( strIn == ' ' ) ) == intSize )
        strOut = '';
    end
    intSize = size( strOut, 2 );
    if( intSize == 1 && strOut( 1 ) == ' ' )
        strOut = '';
    elseif( intSize > 1 )
        if( strOut( intSize ) == ' ' && strOut( intSize - 1 ) == ' ' )

```



```
        strOut = strOut( 1:intSize - 1 );
    end
    intSize = size( strOut, 2 );
    if( intSize > 1 && strOut( 1 ) == ' ' )
        strOut = strOut( 2:intSize );
    end
end
intSize = size( strOut, 2 );
if( intSize > 0 && strOut( intSize ) == ' ' )
    strOut = strOut( 1:intSize - 1 );
end
return;
```

LIST OF REFERENCES

- [1] M.T. Rogers, *A Statistical Model of Multi-Engine OCR Systems*
Masters Thesis, University of Central Florida, Orlando, FL, 2000.
- [2] S. M. Richie, Associate Professor, Ph.D., Electrical Engineering, University of Central Florida, 1989, private communication.
- [3] C. Sprague, *Autonomous Repair of Optical Character Recognition Data through Simple Voting and Multi-Dimensional Indexing Techniques*, Masters Thesis, University of Central Florida, Orlando, FL, 2003.
- [4] A. Weeks, Ph.D. Electrical Engineering, University of Central Florida, 1987, private communication.
- [5] R. Owens, "Mathematical Morphology",
http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/OWENS/LECT3/node3.html.
- [6] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 2nd Edition,
Prentice Hall, 2002, p. 607.