EXTENDING THE PAGE SEGMENTATION ALGORITHMS OF THE OCROPUS DOCUMENTATION LAYOUT ANALYSIS SYSTEM

by

Amy Alison Winder

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

August 2010

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Amy Alison Winder

Thesis Title: Extending the Page Segmentation Algorithms of the OCRopus Document Layout Analysis System

Date of Final Oral Examination: 28 June 2010

The following individuals read and discussed the thesis submitted by student Amy Alison Winder, and they evaluated her presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Elisa Barney Smith, Ph.D.	Co-Chair, Supervisory Committee
Timothy Andersen, Ph.D.	Co-Chair, Supervisory Committee
Amit Jain, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Elisa Barney Smith, Ph.D., Co-Chair of the Supervisory Committee. The thesis was approved for the Graduate College by John R. Pelton, Ph.D., Dean of the Graduate College.

Dedicated to my parents, Mary and Robert, and to my children, Samantha and Thomas.

ACKNOWLEDGMENTS

The author wishes to express gratitude to Dr. Barney Smith for providing a structured and supportive environment in which to formulate, develop, and complete this thesis. Not only did she provide the initial concept for the work, but she was instrumental in selecting the specific topic and followed through by meeting with me weekly to guide and support this effort. The author is also grateful for the instruction Dr. Barney Smith provided in Image Processing, which, paired with computer science, is an exciting field.

Thanks also go to Dr. Andersen for introducing me to the intriguing world of Artificial Intelligence and for allowing me to fulfill the requirements of my thesis under his advisement. The author is grateful for his contributions to the understanding of page segmentation algorithms and performance metrics.

Additionally, the author appreciates the help of Dr. Jain who has been a reliable source of information for the author's entire graduate career at Boise State University. Not only did he streamline the author's academic schedule upon arrival, but he expressed interest and placed value upon the experience and education the author brought to the university, subsequently strengthening her resolve to obtain a second Master of Science degree.

Finally, the author would like to thank the undergraduate students from the university who created and scanned a majority of the document images: Josh Johnson, Kris Burch, and Will Grover.

Funding for this thesis, the classes the author has taken to fulfill the requirements

of this degree, and a stipend was provided by the United States Government under the Trade Adjustment Assistance Act as petitioned by Micron Technology, Inc., the author's former employer. The author is grateful for the guidance provided by Ruby Rangel, Senior Consultant at the Idaho Department of Commerce and Labor.

AUTOBIOGRAPHICAL SKETCH

The author was born in Princeton, NJ and attended Westtown School in Pennsylvania and the University of Rochester in New York where she earned Bachelor of Science and Master of Science degrees in Optics. Following graduation, she worked in the Electro-Optics Division of Honeywell in the Boston area of Massachusetts, supporting the Strategic Defense Initiative. In the Systems Engineering group, she supported simulation efforts of an infrared sensor and in the Optics group she analyzed telescope lens designs.

After a brief sojourn to raise her children, the author took an engineering position at Micron Technology, Inc. in Boise, Idaho. For five years, she worked in the Advanced Reticle group, supporting the development of new reticles used in the photo-lithography process of semiconductor manufacturing. Then, she transferred to the Design department within which she relocated to Japan as a CAD engineer to support the recently opened DRAM design center. Upon returning to the United States two years later, she designed layouts, wrote Design Rule Verification tool sets, and provided general CAD support until deciding to broaden her skill set by pursuing a Master of Science degree in Computer Science at Boise State University.

ABSTRACT

With the advent of more powerful personal computers, inexpensive memory, and digital cameras, curators around the world are working towards preserving historical documents on computers. Since many of the organizations for which they work have limited funds, there is world-wide interest in a low-cost solution to obtaining these digital records in a computer-readable form. An open source layout analysis system called OCRopus is being developed for such a purpose. In its original state, though, it could not process documents that contained information other than text. Segmenting the page into regions of text and non-text areas is the first step of analyzing a mixed-content document, but it did not exist in OCRopus. Therefore, the goal of this thesis was to add this capability so that OCRopus could process a full spectrum of documents.

By default, the RAST page segmentation algorithm processed text-only documents at a target resolution of 300 DPI. In a separate module, the Voronoi algorithm divided the page into regions, but did not classify them as text or non-text. Additionally, it tended to oversegment non-text regions and was tuned to a resolution of 300 DPI. Therefore, the RAST algorithm was improved to recognize non-text regions and the Voronoi algorithm was extended to classify text and non-text regions and merge non-text regions appropriately. Finally, both algorithms were modified to perform at a range of resolutions.

Testing on a set of documents consisting of different types showed an improvement of 15-40% for the RAST algorithm, giving it at an average segmentation accuracy

of about 80%. Partially due to the representation of the ground truth, the Voronoi algorithm did not perform as well as the improved RAST algorithm, averaging around 70% overall. Depending on the layout of the historical documents to be digitized, though, either algorithm could be sufficiently accurate to be utilized.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

DRAM – Dynamic Random Access Memory

CAD – Computer Aided Design

CCD – Charge-Coupled Device

PDF – Portable Document Format

OCR – Optical Character Recognition

MP – Mega Pixel

DPI – Dots Per Inch

IUPR – Image Understanding Pattern Recognition

ASCII – American Standard Code for Information Interchange

RXYC – Recursive X-Y Cut

RLSA – Run-Length Smearing Algorithm

DAFS – Document Attribute Format Specification

XML – eXtensible Markup Language

PSET – Page Segmentation Evaluation Toolkit

ICDAR – International Conference on Document Analysis and Recognition

HTML – Hyper Text Markup Language

CSS – Cascading Style Sheets

RAST – Recognition by Adaptive Subdivision of Transformation Space

SAX – Simple API for XML

DOM – Document Object Model

CHAPTER 1

INTRODUCTION

The ability to create, store, and modify documents on computers has only existed for two to three decades. The printing press, on the other hand, invented in Germany and adopted by the rest of the developed world over time, has been in use for nearly six centuries [1]. Consequently, a multitude of printed documents have been generated in book, magazine, and newspaper form. While many have been lost over the years, a significant portion has been preserved. As historical documents, they are not only fragile, but are inaccessible to most people. In the interest of sharing and preserving their contents for eternity, there is a movement to digitize and store them on computers.

At this time, the most common method for digitizing documents is to use an image scanner [3]. Image scanners, also known as flatbed or desktop scanners, contain a light source, an image sensor such as a CCD, and a glass top upon which the document is placed. Standard scanners that scan documents and produce images of them cost a few hundred dollars; however, it is also possible to purchase large format scanners capable of scanning large books and converting the images into searchable PDF files, but they cost on the order of five thousand dollars. In standard scanners, documents are digitized by OCR software installed onto the computer.

With the advent of inexpensive digital cameras, it is now possible to photograph

the pages of books, the bindings of which may be too brittle to withstand the pressure of being placed, and temporarily deformed, on a scanner bed. Once these images have been obtained, it is necessary to process and analyze them so that they can be converted into text documents that are easily readable and searchable. Since the institutions that house many of these documents have limited funds, a low-cost solution to digitization is the only feasible option.

The impetus for this thesis was a non-profit organization in Germany called the Bavarian Traditional Clothing Culture Center and Archive [2], which was formed to preserve traditional Bavarian costumes and dances. It has been acquiring the newspapers and magazines of various clubs in the area, which it plans to house in a new archive facility. The organization then hopes to digitize these documents so that researchers can examine them to gain a better understanding of how costumes and dance have evolved over the years. Many of these documents were written in the German Fraktur font and contain illustrations, but have standard Manhattan layouts.

At this time, there is an open source document analysis program - OCRopus [4], also developed in Germany - which is capable of converting images of multiple column text documents into text files; however, it cannot process documents that include non-text areas, such as the newspapers mentioned previously. Therefore, non-profit organizations such as the Bavarian Traditional Clothing Culture Center and Archive cannot digitize and share their materials with historians. In response to this need and that of thousands of other libraries and organizations, the goal of this master's thesis is to improve the OCRopus program by extending its page segmentation capability to include mixed-content documents of camera-acquired images.

1.1 Document Recognition and Analysis

1.1.1 Image Acquisition and Processing

The first step in the process of digitizing a document is to capture an image of it. This can be done by either scanning or photographing it. Mid-priced digital cameras are capable of taking pictures with resolutions of 3,872 x 2,592 pixels (10 MP) to 4,672 x 3,104 pixels (15 MP). When these images are printed out at a resolution of 300 DPI, they range in size from 12.9" x 8.6" to 15.6" x 10.3", approximately the same size as a page of a bound historical document. Typical desktop scanners can image documents with resolutions of 150 to 1200 DPI. So, today's common digital cameras can produce images comparable to those generated by a desktop scanner.

Employing digital cameras for image acquisition, on the other hand, introduces a host of other issues that need to be resolved before the documents can be analyzed. First, unless the camera is lined up perfectly with the page, it can capture some areas outside of it including the table top, the adjacent page, and the edges of the pages residing between it and the outer cover. The extraneous information contained within these areas generally needs to be removed prior to analyzing the document so that only the relevant sections of the document are analyzed. This process is typically referred to as border removal.

Once the border has been removed, the image's orientation needs to be checked for skew and corrected. Other factors that need to be taken into consideration are the perspective of the page and any distortions that may be present, such as warping due to stiff spines. Finally, if the lighting under which the photograph was taken was not optimal or if the pages of the document itself have yellowed with age, the image may need to be processed so that it is only represented by black-and-white pixels.

This is called binarization. Additionally, if there is speckle noise present on the page, it will need to be removed as well.

At this time, there is an open source program called PhotoDoc [5] that is capable of handling all of these issues except for noise removal and distortion caused by warping. PhotoDoc can be used in conjunction with an OCR engine such as (open source) Tesseract [6] or OCRopus for image-to-text conversion. In addition to PhotoDoc, researchers in the Image Understanding Pattern Recognition (IUPR) Research Group of Kaiserslautern, Germany, in partnership with the Adaptive Technology Resource Centre of Toronto, are developing a hardware/software solution for document analysis called Decapod [7]. Decapod is being designed to work in conjunction with OCRopus, which has skew correction, binarization, and noise-reduction functionality, but not border removal. The hardware component of Decapod will consist of a camera/tripod assembly for photographing the documents and it is assumed that border removal will be added to OCRopus to complete the software component.

1.1.2 Document Analysis

Once the image has been acquired and processed, it needs to be analyzed in terms of layout. That is, if the page contains information other than text like graphs, tables, and half-tone images, the program needs to determine which areas are text and which are not. This way only the text regions are sent to the OCR engine, preventing unnecessary errors. Dividing a document in this fashion is called page segmentation. Once the text regions have been identified, the individual lines are sorted into reading order.

At this point, the OCR engine is called upon to recognize the characters in the text regions and convert them into ASCII or Unicode characters. The first step

in this process is to segment the lines into words then the words, into characters. Depending on the algorithm used, certain features like geometrical moments, contour Fourier descriptors or number of pixels per row are extracted for each character. These features can then be matched to a character in a database using a K-Nearest Neighbor algorithm or can be input into a Decision Tree or Neural Network that returns the most likely character.

Since the motivation behind this thesis is to help provide a means for curators to digitize documents in a cost-efficient manner, open source document analysis systems were researched. Besides OCRopus, a program called Gamera [8] was found, but it is more of a toolkit than a comprehensive document analysis system. It has image processing and OCR capabilities, but no apparent page segmentation functionality. Therefore, OCRopus was deemed the system of choice. Like Gamera, page segmentation has not been developed in OCRopus; however, it has some algorithms in place that can be expanded upon.

1.1.3 Page Segmentation Algorithms

Over the years researchers have developed a number of page segmentation algorithms, which can be categorized as top-down, bottom-up, or hybrid methods [9]. Top-down methods involve operating on the document as a whole and subdividing it, whereas bottom-up methods start with pixel-level operations, which create low-level groups that are merged into segmented regions. Hybrid methods do not fall into either of these categories, but may include a little of both.

The Recursive X-Y Cut (RXYC) and Run-Length Smearing Algorithms (RLSA) fall into the top-down category. RXYC [10] starts by examining the image and constructing a block profile where white pixels are represented as zeros and black pixels

are represented as ones. The block profile then consists of vertical and horizontal projections of the black areas. Zeros extending across the entire document in the block profile, or valleys, are possible column candidates with the widest being the best candidate. Once the largest valley is discovered, the document is subdivided around it and one of the new blocks is examined for the existence of valleys. After it has been completely subdivided, the other block is addressed in the order of a depth-first traversal. The blocks are represented in a data structure called an X-Y tree, where the valleys are the nodes and the blocks the elements. The structure can also be visualized as a set of nested, rectangular blocks.

Like RXYC, RLSA [11] also operates from the top down; however, it classifies the regions as well. It examines each of the pixels in a row-by-row and column-by-column fashion and changes each white pixel to black if it is surrounded by enough black pixels. Black pixels are not changed. After the pixels have been updated, the generated row and column bit maps are ANDed together to form a single bit map. This bit map then undergoes a horizontal smoothing operation to ensure the connection of words in a text line. The final bit map typically consists of blocks corresponding to individual text lines and non-text areas. At this point, measurements are taken of the blocks (i.e., numbers of black and white pixels, dimensions, coordinates) from which histograms are built and block classifications derived.

In terms of bottom-up approaches, two documented methods include the Docstrum and Voronoi algorithms. Docstrum [12] is a contraction of Document Spectrum and only segments and classifies text. So, it is not a page segmentation algorithm in the strict sense; however, its methodology is of interest. It starts by extracting the connected components (groups of adjacent black pixels) of the image that typically correspond to characters. Next the K-Nearest Neighbors of each component are found

based on the coordinates of their centroids and the angle made by the line connecting them. So, components placed in close proximity and side-by-side (e.g., along a text line) are given priority. Once these have been grouped, they are classified as text, title, abstract, etc., based on histograms of their dimensions.

The Voronoi method [13] also starts by identifying connected components. Afterwards, it extracts sample points along the boundaries from which it constructs a Voronoi point diagram. Since the number of components is on the order of the number of characters, a large number of edges are created, most of which are superfluous. These unnecessary edges are deleted based on length (i.e., short ones) and whether or not they are connected to other lines. In this way, the diagram is converted to an area Voronoi diagram whose areas represent the page regions.

Comparing the two, the top-down approach requires a priori knowledge of the document because parameters need to be set for determining which white areas are valleys in RXYC as well as for setting the smearing threshold and smoothing filters of RLSA. Additionally, neither one of these algorithms lends itself to segmenting layouts that include regions with diagonal or curved boundaries (non-Manhattan layouts). The bottom-up approaches, on the other hand, do not require a priori knowledge of the layout, but will accumulate errors if any exist. Additionally, the Voronoi method is capable of segmenting more complex, non-Manhattan layouts.

1.1.4 Page Segmentation Accuracy

To assess the accuracy of various page segmentation algorithms, it is necessary to compare the output to the true region types or so called "ground truth" of the page. Three possible formats that this ground truth can take are: image files with labeled pixels, Document Attribute Format Specification (DAFS) [14] files, or eXtensible

Markup Language (XML) files. In the first case, pixels are labeled with their region number or type to which a corresponding unique color is assigned (i.e., green for text, red for images, etc.). The colors can be assigned using a common graphics program. An advantage to this format is that regions of any shape can be represented, although generating the ground truth for non-rectangular regions can be time consuming. The color coding can be extended to define reading order as well, which is done by OCRopus where the color gradually changes (e.g., gets "greener") as successive lines are encountered in a column.

In the second case, the image is converted into either an ASCII, Unicode, or binary file, which contains tags representing the following entities: doc (the document as a whole), page, column, paragraph, line, word, and glyph (a single character in the text); however, a more general file format than DAFS is XML in which the regions are defined by the user. For example, regions can be represented by "zone" tags that have a "classification" attribute specifying its type (i.e., text, graph, image, etc.), allowing for non-text types. The zones can also have "dimension" subtags that include attributes for the coordinates of the corners or vertices that constrain them to being rectangles or polygons. Realizing that there was need for a tool to generate ground truth of this type, researchers created TrueViz [15], an open source graphical application for producing XML ground truth files.

Once the ground truth and a file containing the detected regions have been generated, they need to be compared and an assessment made as to how well they match. The same researchers that supplied TrueViz also created a toolkit called PSET [16, 17], which stands for Page Segmentation Evaluation Toolkit. PSET contains several algorithms for segmenting document images as well as an algorithm for measuring the performance of the segmentation; however, PSET generates DAFS

formatted files and measures the segmentation performance in terms of text-line accuracy. So, it is not suitable for documents that include images.

Using color-coded ground truth files, one could apply the method developed by Shafait and Breuel [18] for measuring segmentation accuracy whereby counts of the number of correct, over and under segmentations are taken in addition to several other measurements. In the case of comparing rectangular zones, though, one could apply the metric used in the page segmentation competition held by the International Conference on Document Analysis and Recognition (ICDAR) every odd year [19]. This method involves calculating and tabulating "match scores" for the regions, extracting parameters from this table, calculating detection and recognition accuracies based on these parameters, then using this information to calculate performance rates for each region as well as an overall performance measurement. Since the documents of interest for this thesis have Manhattan layouts and no program is publicly available to measure segmentation performance, one was written based on the ICDAR method.

1.2 Document Analysis Programs

As mentioned earlier, OCRopus is an open source layout analysis and OCR program. It is being developed for large-scale digital library applications and is distributed under the Apache 2 license. Its design supports multi-lingual and multi-script recognition by using Unicode as well as HTML and CSS standards to represent the typographic formats of the world's scripts. OCRopus itself is built in modules that can be switched to test different algorithms as well as incorporate new ones. The programming language is C++, along with a built-in scripting language called Lua. Its architecture consists of Layout Analysis, Text Line Recognition, and Statistical

Language Modeling.

The Layout Analysis module includes five page segmentation algorithms: a trivial morphological segmenter, a single-column projection-based segmenter, a RXYC segmenter, a Voronoi segmenter, and a Recognition by Adaptive Subdivision of Transformation Space (RAST) segmenter. The morphological segmenter simply applies a smearing algorithm to the image to obtain isolated blocks; whereas, the projection-based segmenter examines the horizontal projection profiles to segment text lines into characters.

The RXYC and Voronoi segmenters apply the algorithms discussed earlier, but do not classify or color code the regions by themselves so they cannot be used to convert images to text. Also, all four of these algorithms only output image files, not XML files. Of the four, the Voronoi algorithm showed the most promise because it was able to segment a small collection of complex layouts with the most accuracy (this topic will be covered in more detail in Chapter 3). Therefore, it was deemed a good candidate for further improvement.

RAST [4, 20], on the other hand, was the most developed algorithm of the five and operates by default; however, it is not a page segmentation algorithm, per se. It was designed for text-only documents [21] and consists of three steps: finding the columns, finding the text-lines, then determining the reading order. To find the columns it employs a whitespace rectangle algorithm [22] which was inspired by RXYC. This algorithm differs from RXYC in that it keeps track of the white spaces rather than the blocks, and combines them as opposed to subdividing the blocks.

RAST starts by extracting the connected components then determines the largest possible (maximal) whitespace rectangles (or covers) based on the component bounding boxes. These are then sorted based on how many connected components (e.g.,

text lines) touch each major side. In this way, column dividers rather than paragraph or section dividers take priority. The covers are then merged iteratively as long as the combined cover obeys a given rule of how many components must be incident upon it. Once the columns dividers (or gutters) have been found, the connected components are examined and classified as text lines, graphics, and vertical/horizontal rulings based on their shapes and the fact that they do not cross any gutters.

At this point, the reading order is determined by considering pairs of lines such that either the line below or the line to the right at the top of the page (e.g., in the next column) goes next. Once these have been ordered, the pairs are sorted to give the final reading order. Preliminary tests of the RAST algorithm indicated that it was capable of processing multiple column documents as long as they did not contain images; however, when images were included errors were output and the reading order was negatively impacted (more on this in Chapter 3). For these reasons, the RAST module was judged as needing improvement.

While the goal of this thesis is to improve the performance of the OCRopus system, the performance of a commercial program, ABBYY FineReader [23], was also measured for comparison. As written earlier, the motivation behind this thesis is to aid curators in their effort to digitize historical documents, specifically Bavarian documents that were written in the Fraktur font. ABBYY has recently added the Fraktur font to its OCR engine so it should be able to recognize the characters in these documents; however, its page segmentation capabilities were unknown. Since the topic of this thesis is page segmentation, this product was evaluated in this area only.

1.3 Thesis Statement

The goals of this thesis are to:

1. Develop an algorithm based on the OCRopus RAST algorithm that can segment text-only documents, mixed-text, and non-text documents. Ensure that it can process layouts similar to to that of the Bavarian documents and can recognize the regions with an accuracy of least 90% over a range of resolutions.

2. Develop an algorithm based on the Voronoi method that not only segments a document into text and non-text regions, but ensures that like regions are merged and all regions are classified. As for performance, impose the same constraints as in the previous objective.

In order to be able to measure these goals, the following tasks were completed:

1. A program was written that compares detected segments to ground truth and returns a performance measurement.

2. XML output of segmented regions was implemented in OCRopus.

As a measure of performance before and after the improvement, as well as with respect to industry standards, eight classes of documents stored at five different resolutions were segmented by the following programs, then analyzed:

1. OCRopus' current and improved RAST algorithms

2. OCRopus' current and improved Voronoi algorithms

3. ABBYY FineReader

CHAPTER 2

METHODS

As covered in the first chapter, the OCRopus document analysis system is the most suitable open source program for digitizing large numbers of historical documents. In its current state, though, it is incapable of processing complex layouts because its page segmentation algorithms are not fully developed. In order to assess the performance of these methods, OCRopus needed to be modified to output the detected page regions. The format chosen for this representation was XML. Similarly, documents called ground truth, that represent the true regions of the page, needed to be generated for comparison. Then, a program needed to be written to compare the detected regions to the ground truth.

Since overall performance metrics fail to convey how a particular method might be failing, images of the output were also examined. For example, when creating text blocks, the RAST algorithm labels them by assigning slightly different colors to them, which are subsequently used to define the reading order. By modifying these colors, the author was able to observe the different text blocks as well as the segmentation of the non-text areas.

As for the Voronoi method, it was less sophisticated than RAST because it did not classify the regions, so the graphical output could only be examined for segmentation. In this case, it was not necessary to color the regions differently; lines were simply

drawn around them in the original implementation. The accuracy of these regions could then be examined by analyzing the amount of fracturing and merging.

2.1 Comparison Program Algorithm

A search of open source XML zone comparison programs based on the ICDAR Page Competetion method [19] did not yield any software, so a program was written to compare detected regions to ground truth. The algorithm starts by calculating "match scores" for each of the regions. That is, each of the regions of the ground truth are compared to each of the detected regions and given a score indicating how well they match. If the regions match perfectly, they are given a score of one; otherwise, if they are completely separate, they are given a score of zero. If they overlap partially, the score is given by

$$MatchScore(i,j) = a\frac{T(G_i \cap R_i \cap I)}{T((G_i \cup R_i) \cap I))} \tag{2.1}$$

where

$$a = \begin{cases} 1 & \text{if } g_j = r_i \\ 0 & \text{otherwise} \end{cases}$$

and

T(s) is a function that counts the elements of set s,

G_j is the set of all points inside the j^{th} ground truth region,

g_j is the j^{th} ground truth region,

R_i is the set of all points inside the i^{th} detected (or result) region,

r_i is the i^{th} detected region,

I is the set of all ON image points.

In the case of rectangles being compared according to Phillips and Chhabra [24], the equation for the match score is reduced to

$$MatchScore(i,j) = a\frac{area(g_i \cap r_i)}{max(area(g_i), area(r_i))}. \tag{2.2}$$

Once the match scores have been calculated, properties of the table are extracted, including the number of one-to-one matches, the number of one-to-many matches, and the number of many-to-one matches. The latter two quantities are computed from both perspectives: the ground truth and detected. For example, if the ground truth contained a text region of four paragraphs, but the segmenter detected these as four separate regions, it would count as a ground truth one-to-many match and four detected many-to-one matches. These values are determined for each region then used to determine the detection rates and recognition accuracies as given by

$$DetectRate_i = w_1\frac{one-to-one_i}{N_i} + w_2\frac{g\_one-to-many_i}{N_i} + w_3\frac{g\_many-to-one_i}{N_i} \tag{2.3}$$

$$RecognitionAccuracy_i = \left\{ w_4\frac{one-to-one_i}{M_i} + w_5\frac{d\_one-to-many_i}{M_i}, + w_6\frac{d\_many-to-one_i}{M_i} \right\} \tag{2.4}$$

where w_1, w_2, w_3, w_4, w_5 and w_6 are pre-determined weights,
N_i is the number of ground truth elements belonging to the i^{th} entity,

M_i is the number of detected elements belonging to the i^{th} entity.

Using the detection rates and recognition accuracies, the Entity Detection Metric (EDM) for each region can be calculated as

$$EDM_i = \frac{2\ DetectRate_i\ RecognitionAccuracy_i}{DetectRate_i + RecognitionAccuracy_i} \tag{2.5}$$

and an overall performance metric or Segmentation Metric (SM) can be given by

$$SM_i = \frac{\sum N_i\ EDM_i}{\sum N_i}. \tag{2.6}$$

2.2 Method to Output in XML Format

Since the layout of interest is Manhattan and the ICDAR comparison algorithm was applied, the output of the segmenters needed to be in XML format. The release of OCRopus at the onset of this thesis (Alpha) has a module called "buildhtml", but it is not complete. It outputs the preamble, or metadata of the document, but none of the text. A contributor to the project built a patch for it that can output the text of a simple document; however, this output does not contain any page segmentation information. There are no tags for regions. So, it cannot be used for comparison to the ground truth. Therefore, XML page segmentation output needed to be implemented by the author in OCRopus.

2.3 Original RAST Algorithm

The RAST module of OCRopus was run on the test documents mentioned earlier. When run in regular, text-recognition mode, the presence of half-tone images and

graphs resulted in unusable output. That is, since it was unable to segment the page into text and non-text regions, it treated the entire page as text. Therefore, when it encountered non-text areas, it attempted to recognize characters within them, which translated into nonsensical text intermingled with a series of error messages.

As for evaluating its page segmentation capability, since XML format was not originally an option, color-coded images were output and examined instead. In terms of classification, it has three types: text, graphics (i.e., non-text), and column dividers or gutters. The column dividers are colored yellow, graphics light green, and text all other colors.

The most prevalent error found was sections of non-text being classified as both text and non-text. Figure 2.1 shows a page with two figures. The figure at the top of the page is a book colored bright green, red, orange, and blue. Similarly, the figure at the bottom is a rabbit colored bright green and blue. When the program was adjusted so that only non-text pixels were output, both figures were completely green, meaning all of the pixels were classified as non-text; however, when both types of pixels were output, multiple colors emerged in the figures, indicating that some pixels were considered both text and non-text.

Graphs also tended to contain both text and non-text pixels; however, they did not overlap as in the case described in the previous paragraph. Figure 2.2 shows the output of a page taken from a scientific journal. The legends and axis labeling were classified as text, but the border, data, and data lines were classified as non-text.

Tables, on the other hand, not only contained text and non-text pixels, but column divider pixels as well. Figure 2.3 shows the output of a page containing a table for illustration. Note the presence of gutters between each column of the table. This resulted in oversegmentation of the table so that the correct reading order could not

Figure 2.1: Example of RAST output of OCRopus. Note the multiple colors in both figures.

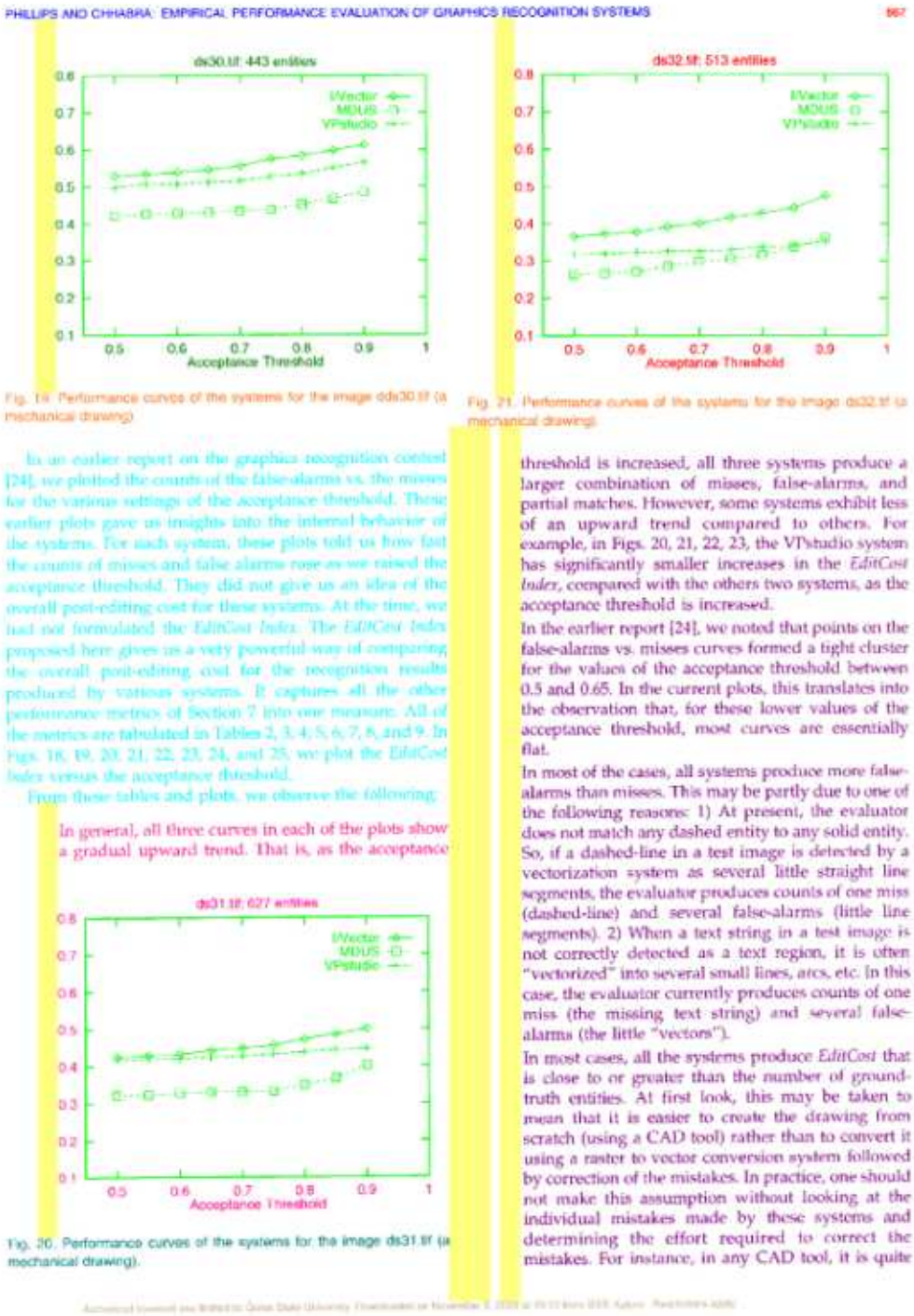


Figure 2.2: Example of RAST output of OCRopus. Note the text coloring of the x-axis labels and the absence of the y-axis labels.

be found for the OCR engine.

Based on this data, it was determined that the RAST algorithm could be improved by correcting the segmentation of non-text (e.g., half-tone images) so that text is not included, as well as properly identifying and segmenting graphs and tables. Since the goal of this thesis is to enable OCR of text areas, these regions need to be grouped properly and identified as non-text along with any encountered images. Once this is done, only text should be fed to the OCR engine. Also, since the images will be acquired using cameras with different resolutions, RAST needs to be robust enough to segment low resolution images as well. Therefore, the first goal of this thesis is to implement these improvements, ensuring that they perform at a range of resolutions as discussed in Section 3.3.

2.4 Voronoi Basis

The Voronoi module of OCRopus, conceived and implemented by Kise et al. [13], was also run on the test documents discussed in Section 2.1. It is less sophisticated than RAST in that it does not classify the regions, so consequently it cannot place the text in reading order, which means there is no text output. As a segmentation algorithm, though, it works fairly well. While it does not identify columns, it groups blocks of text in different columns correctly and usually creates separate segments for picture captions.

Figure 2.4 shows the Voronoi output from the same page as Figure 2.1. While the text blocks are segmented properly, the non-text areas (e.g., half-tone images) are oversegmented. The left side of the figure of the book at the top of the page contains over fifteen regions alone. Similarly, the figure of the rabbit at the bottom of the page

illustrated by plotting block segments in the R-H plane (see Fig. 3). Each table entry is equal to the number of block segments in the corresponding range of R and H. Thus, such a plot can be considered as a two-dimensional histogram. The text lines of the document shown in Fig. 2(a) form a clustered population within the range $20 < H < 35$ and $2 < R < 8$. The two solid black lines in the lower right part of the original document have high R and low H values in the R-H plane, whereas the graphic and halftone images have high values of H. Note that the scale used in Fig. 3 is highly nonlinear.

The mean value of block height H_m and the block mean black pixel run length R_m for the text cluster may vary for different types of documents, depending on character size and font. Furthermore, the text cluster's standard deviations $\sigma(H_m)$ and $\sigma(R_m)$ may also vary depending on whether a document is in a single font or multiple fonts and character sizes. To permit self-adjustment of the decision boundaries for text discrimination, estimates are calculated for the mean values H_m and R_m of blocks from a tightly defined text region of the R-H plane. Additional heuristic rules are applied to confirm that each such block is likely to be text before it is included in the cluster. The members of the cluster are then used to estimate new bounds on the features to detect additional text blocks [5]. Finally, a variable, linear, separable classification scheme assigns the following four classes to the blocks:

Class 1 Text:
$R < C_{21} R_m$ and
$H < C_{22} H_m$.

Class 2 Horizontal solid black lines:
$R > C_{21} R_m$ and
$H < C_{22} H_m$.

Class 3 Graphic and halftone images:
$R > 1/C_{23}$ and
$H > C_{22} H_m$.

Class 4 Vertical solid black lines:
$R < 1/C_{23}$ and
$H > C_{22} H_m$.

Values have been assigned to the parameters based on several training documents. With $C_{21} = 3$, $C_{22} = 3$, and $C_{23} = 5$, the outlined method has been tested on a number of test documents with satisfactory performance. Figure 2(e) shows the result of the blocks which are considered text data (class 1) of the original document in Fig. 2(a).

There are certain limitations to the block segmentation and text discrimination method described so far. On some documents, text lines are linked together by the block segmentation algorithm due to small line-to-line spacing, and thus are assigned to class 3. A line of characters exceeding 3 times H_m (with $C_{22} = 3$), such as a title or heading in a

Table 1 Processing result of document in Fig. 2. Columns 1 to 7 contain the results of the measurements performed simultaneously with labeling. The last column is the text classification result.

BC	s_{min}		s_{max}		DC	TC	Class
702	995		2341		302		
6089	1090	771	2266		5608	170	
6387	307	265	2243		1142	396	
15396	307	657	2208		3118	1005	
9341	1090	366	2184		2469	580	
16706	183	779	2171		3459	1070	
19447	1090	771	2147		5181	1110	
9244	185	385	2098		1780	528	
5244	1401	152	2077		1947	303	
18502	185	779	2066		4112	1183	
17592	185	779	2024		3613	1018	
5667	1091	770	2008		5394	72	
12500	183	474	1947		4751	735	
19318	1144	717	1923		4436	1155	
19252	1101	760	1883		3981	1059	
11101	183	483	1830		2213	756	
1258	1144	171	1821		414	123	
20200	1082	779	1783		4349	1229	
276147	188	380	1037	706	47742	8738	
418268	1084	780	1209	546	195328	59906	
10935	1089	503	1117	30	2139	648	
18370	1088	773	1080		3406	996	
19120	1088	773	1043		3612	1046	
3434	1203	126	969		770	226	
13694	193	489	954		2251	758	
17336	1088	773	934		3417	1012	
21586	193	783	915		3574	1164	
17601	1088	773	895		3580	1040	
19174	193	779	880		3437	1165	
23506	193	778	840		4167	1256	
15500	1088	773	824		3544	1038	
20263	193	778	804		3800	1183	
16619	1088	773	788		3310	989	
10112	193	393	731		2911	666	
16727	1088	773	703		3749	1083	
585911	1087	778	175	504	175114	51527	
19512	193	777	643	29	4046	1205	
406563	193	777		538	63103	12598	
10467	1089	477	113	28	1835	542	
18717	1089	776	78	30	3427	968	

document, is assigned to class 3. An isolated line of text printed vertically, e.g., a text description of a diagram along the vertical axis, may be classified as a number of small text blocks, or else as a class 3 block.

Consequently, in order to further classify different data types within class 3, a second discrimination based on shape factors [10] is used. The method uses measurements of border-to-border distance within an arbitrary pattern. These measurements can be used to calculate meaningful features like the "line-sharpness" or "compactness" of objects within binary images. While the calculations are complex and time-consuming, they are done only for class 3, and thus add only a small increment to the overall processing time. This hierarchical decision procedure, in which "easy" class

Figure 2.3: Example of RAST output of OCRopus. Note the column dividers in the table and the absence of some entries.

contains at least four additional regions.

The output of the scientific paper containing graphs is shown in Figure 2.5. Each of the graphs contains four to twelve regions within the boxed area, as well as individual regions for each of the axis numbers and labels when only one region should be created for each graph.

The last example, shown in Figure 2.6, illustrates the output of the document containing a table. The table is oversegmented along the columns as in the RAST case; however, the titles are not included in the column regions.

In terms of zone classification, a number of papers have been written on the subject. The paper documenting the Voronoi method itself [13] states that the zones were classified as either text or non-text in their study; however, it is not clear how this was done. From what the author can discern, it may have been when the lines between the characters were deleted, thus assigning the area containing those lines the class text.

Two other groups of researchers report classifying segmented regions using neural networks [26, 27]. First, they extract the connected components, then they segment the image into regions using either a RXYC or RLSA method. Then, based on the bounding boxes of the connected components, they use features including the amount of overlap between boxes, the amount of touching between boxes, the fill ratio of the boxes (number of black pixels to box area), the dimensions (height, aspect ratio, and size) of the boxes, the ratio of black to white pixels, the number of horizontal transitions from black to white pixels, the length of the horizontal run of black pixels, and the angle subtended from the lower-left corner to the upper-right corner to classify the regions using a neural network.

A third method [28] uses a simple nearest-neighbor approach with various his-

Figure 2.4: Example of Voronoi output of OCRopus. Note the oversegmentation of the figures.

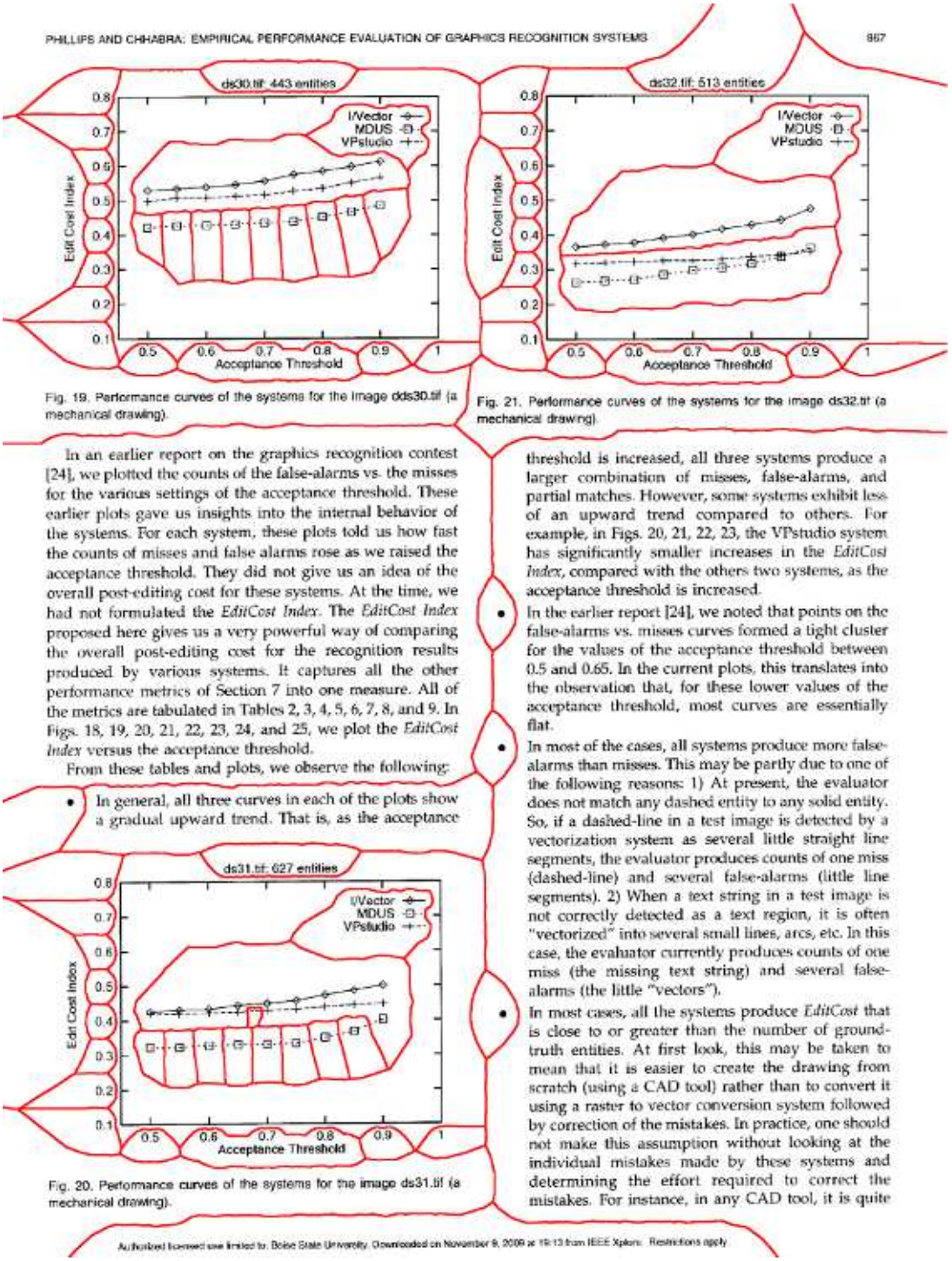
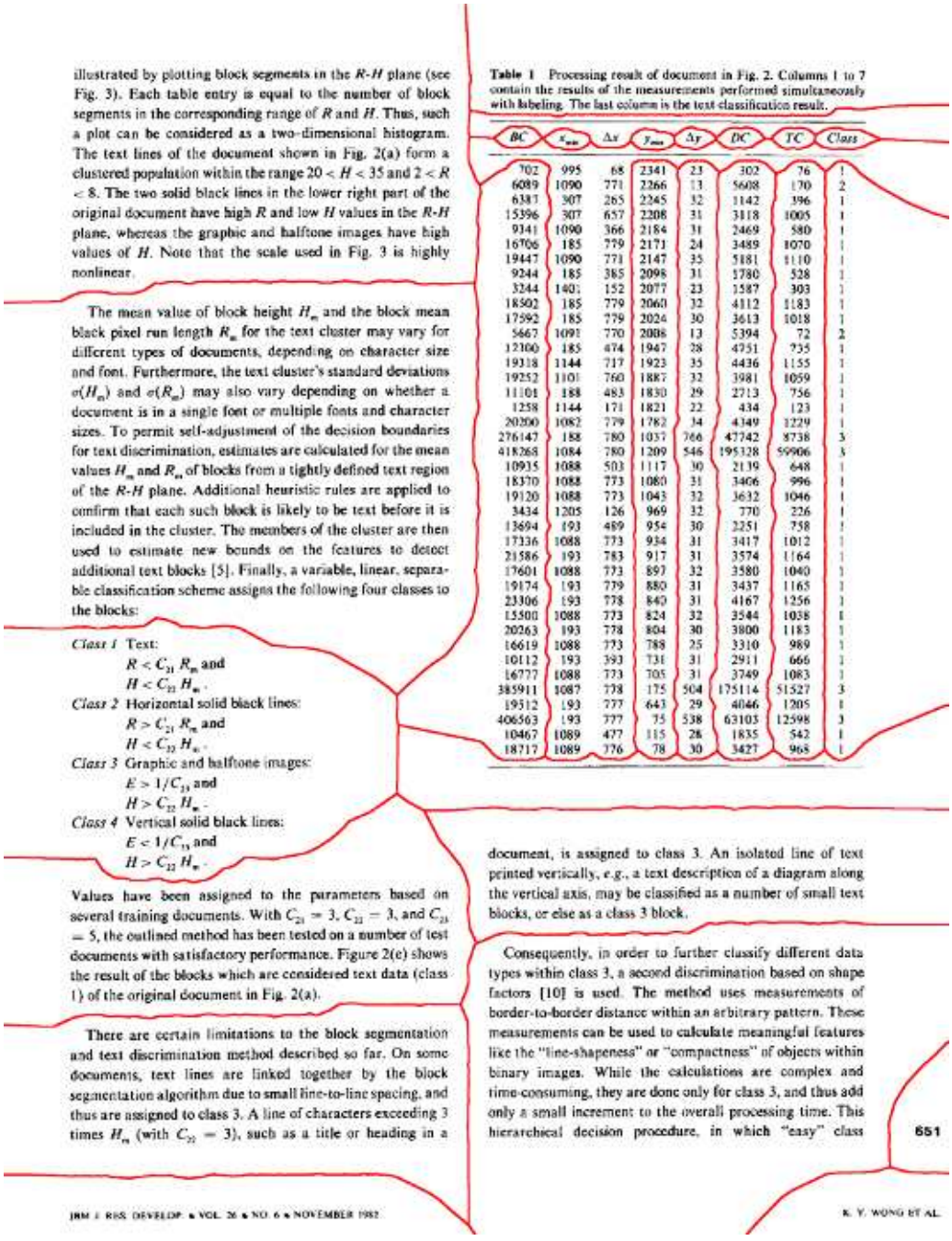


Figure 2.5: Example of Voronoi output of OCRopus. Note the oversegmentation of the graphs.



illustrated by plotting block segments in the R-H plane (see Fig. 3). Each table entry is equal to the number of block segments in the corresponding range of R and H. Thus, such a plot can be considered as a two-dimensional histogram. The text lines of the document shown in Fig. 2(a) form a clustered population within the range $20 < H < 35$ and $2 < R < 8$. The two solid black lines in the lower right part of the original document have high R and low H values in the R-H plane, whereas the graphic and halftone images have high values of H. Note that the scale used in Fig. 3 is highly nonlinear.

The mean value of block height H_m and the block mean black pixel run length R_m for the text cluster may vary for different types of documents, depending on character size and font. Furthermore, the text cluster's standard deviations $\sigma(H_m)$ and $\sigma(R_m)$ may also vary depending on whether a document is in a single font or multiple fonts and character sizes. To permit self-adjustment of the decision boundaries for text discrimination, estimates are calculated for the mean values H_m and R_m of blocks from a tightly defined text region of the R-H plane. Additional heuristic rules are applied to confirm that each such block is likely to be text before it is included in the cluster. The members of the cluster are then used to estimate new bounds on the features to detect additional text blocks [5]. Finally, a variable, linear, separable classification scheme assigns the following four classes to the blocks:

Class 1 Text:
$R < C_{21} R_m$ and
$H < C_{22} H_m$.

Class 2 Horizontal solid black lines:
$R > C_{21} R_m$ and
$H < C_{22} H_m$.

Class 3 Graphic and halftone images:
$E > 1/C_{23}$ and
$H > C_{22} H_m$.

Class 4 Vertical solid black lines:
$E < 1/C_{23}$ and
$H > C_{22} H_m$.

Values have been assigned to the parameters based on several training documents. With $C_{21} = 3$, $C_{22} = 3$, and $C_{23} = 5$, the outlined method has been tested on a number of test documents with satisfactory performance. Figure 2(e) shows the result of the blocks which are considered text data (class 1) of the original document in Fig. 2(a).

There are certain limitations to the block segmentation and text discrimination method described so far. On some documents, text lines are linked together by the block segmentation algorithm due to small line-to-line spacing, and thus are assigned to class 3. A line of characters exceeding 3 times H_m (with $C_{22} = 3$), such as a title or heading in a

IBM J. RES. DEVELOP. • VOL. 26 • NO. 6 • NOVEMBER 1982

Table 1 Processing result of document in Fig. 2. Columns 1 to 7 contain the results of the measurements performed simultaneously with labeling. The last column is the text classification result.

BC	x_{min}	Δx	y_{min}	Δy	DC	TC	Class
702	995	68	2341	23	302	76	1
6089	1090	771	2266	13	5608	170	2
6381	307	265	2245	32	1142	396	1
15396	307	657	2208	31	3818	1005	1
9141	1090	366	2184	31	2469	580	1
16706	185	779	2171	24	3489	1070	1
19447	1090	771	2147	35	5181	1110	1
9244	185	385	2098	31	1780	528	1
3244	1401	152	2077	23	1587	303	1
18502	185	779	2060	32	4112	1183	1
17592	185	779	2024	30	3613	1018	1
5667	1091	770	2008	13	5394	72	2
12100	185	474	1947	28	4751	735	1
19318	1144	717	1923	35	4436	1155	1
19252	1101	760	1887	32	3981	1059	1
11101	188	483	1830	29	2713	756	1
1258	1144	171	1821	22	434	123	1
20200	1082	779	1782	34	4349	1229	1
276147	188	780	1037	766	47742	8738	3
418268	1084	780	1209	546	195328	59906	3
10935	1088	503	1117	30	2139	648	1
18370	1088	773	1080	31	3406	996	1
19120	1088	773	1043	32	3632	1046	1
3434	1205	126	969	32	770	226	1
13694	193	489	954	30	2251	758	1
17336	1088	773	934	31	3417	1012	1
21586	193	783	917	31	3574	1164	1
17601	1088	773	897	32	3580	1040	1
19174	193	779	880	31	3437	1165	1
23306	193	778	849	31	4167	1256	1
15500	1088	773	824	32	3544	1038	1
20263	193	778	804	30	3800	1183	1
16619	1088	773	788	25	3310	989	1
10112	193	393	731	31	2911	666	1
16777	1088	773	705	31	3749	1083	1
385911	1087	778	175	504	175114	51527	3
19512	193	777	643	29	4046	1205	1
406563	193	777	75	538	63103	12598	3
10467	1089	477	115	28	1835	542	1
18717	1089	776	78	30	3427	968	1

document, is assigned to class 3. An isolated line of text printed vertically, *e.g.*, a text description of a diagram along the vertical axis, may be classified as a number of small text blocks, or else as a class 3 block.

Consequently, in order to further classify different data types within class 3, a second discrimination based on shape factors [10] is used. The method uses measurements of border-to-border distance within an arbitrary pattern. These measurements can be used to calculate meaningful features like the "line-shapeness" or "compactness" of objects within binary images. While the calculations are complex and time-consuming, they are done only for class 3, and thus add only a small increment to the overall processing time. This hierarchical decision procedure, in which "easy" class

651

K. Y. WONG ET AL.

Figure 2.6: Example of Voronoi output of OCRopus. Note the oversegmentation of the table.

tograms (Tamara texture, relational invariant feature, run-length of black and white pixels in eight different directions, heights, widths and separations of the bounding boxes) and other aspects (fill ratio and the total number, mean and variance of black and white pixel runs) as the features. This method is also accurate, but as in the previous method, determining the values of all of the features is time consuming, and thus, since this the focus of this thesis is on page segmentation rather than region classification, a simpler approach was sought.

Based on the segmentation results shown earlier and the need for region classification, the second goal of this thesis is to improve the Voronoi algorithm in OCRopus so that it does not oversegment half-tone images, graphs, and tables. Once this was done, it needed to classify these regions as text or non-text for which a robust, yet non-complex solution was found. Since placing the text regions in reading order is beyond the scope of this thesis, it was not implemented for this effort. Like RAST, Voronoi needed to operate successfully at low resolutions as well. The design and implementation of this algorithm is covered in Section 3.4.

CHAPTER 3

DESIGN AND IMPLEMENTATION

This chapter covers the algorithm development and implementation details of the comparison program, XML output in OCRopus, RAST page segmentation, and Voronoi page segmentation. Based on the method described in Section 2.1, a comparison program was implemented and tested iteratively to ensure the correct analysis of various types of errors. Since it was to be used as the metric for both algorithms, it was imperative that it be correct. On the other hand, introducing XML output to the OCRopus program was straightforward and is explained in Section 3.2.

Once OCRopus could output XML page regions and they could be compared to the ground truth, the algorithms were developed. Since the RAST algorithm was more sophisticated than the Voronoi algorithm, it was addressed first. A collection of different types of documents were processed by it and their segmentations evaluated. The most frequently occurring errors were addressed first by introducing additional steps in the algorithm, running more tests, then analyzing the results. This process was repeated until satisfactory performance levels were achieved at 300 DPI.

At this point, the program was examined for resolution dependent parameters. Upon their discovery, they were replaced by parameters that were extracted from the document itself (i.e., certain connected components within it) so that the performance would not change as a function of resolution.

As for the Voronoi algorithm, since it did not classify the regions, this functionality needed to be added first. After this was done, it was possible to address the quality of the segments themselves in terms of oversegmentation using a selection of different documents. Since non-text areas suffered from this problem the majority of the time, the algorithm only needed to treat the non-text regions. Once the regions were segmented properly, resolution issues were resolved as in the RAST algorithm.

3.1 Comparison Program Implementation

The first step in comparing detected regions to ground truth is parsing the XML files. There are two ways this can be done in C++: SAX (Simple API for XML) and DOM (Document Object Model) [25]. The SAX method involves event-based parsing where either callback functions or an object that implements various methods are created and, as certain tags are encountered, actions are taken. The DOM method, on the other hand, creates a tree data structure while parsing the file so that the elements and their descendants can be accessed repeatedly. Since this method essentially has built-in parsing functionality, it was chosen for this program.

As written earlier, each XML file contains a list of zones corresponding to the segmented regions of the page. Each of the zones has the following tags: "ZoneCorners," "Vertex," "Classification," and "CategoryValue." Figure 3.1 illustrates how a file with two zones would be structured. The information for the zones is kept in the leaves of the tree. So, in this case, the "Vertex" leaves contain the coordinates of the corners of the rectangles and the "CategoryValue" leaf contains the class of the zone (i.e., "Text" or "Non-text").

Once the file has been parsed into the XML data structure, each of the zones is

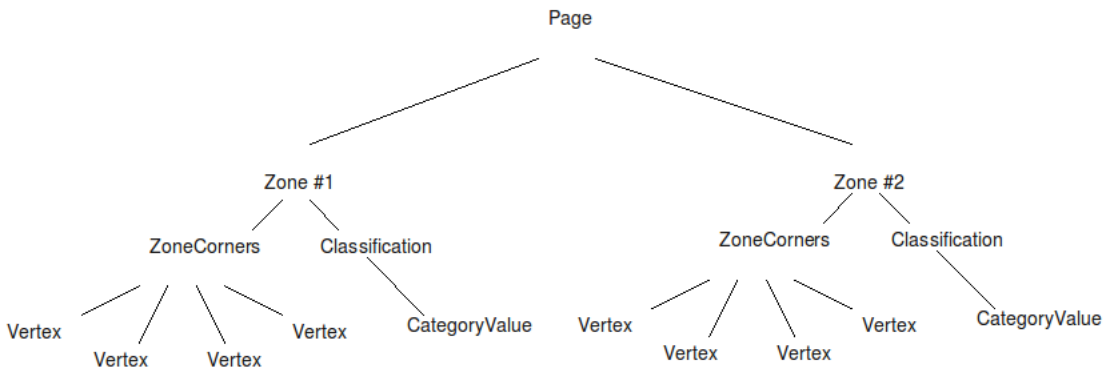


Figure 3.1: XML file data structure.

examined and placed into a custom **Rect** object that has attributes for the vertices and classification. It is shown in detail in Appendix A. A two-dimensional array, or vector, of **Rects** is then created to house the objects where one dimension corresponds to the class of the zone and the other to the number of the zone. In this way, the statistics for each class can be tabulated easily.

Following **Rect** vector construction, the work of comparing the data files begins. The first step is to calculate the match scores of each of the regions and place them into a two-dimensional array where one dimension represents the ground truth regions and the other the detected regions. Each of the regions is considered in turn and the amount of overlap between it and each of the other regions is calculated. The overlap is determined by comparing the vertices of each rectangle then summing the pixels in the area of overlap, if any. The match score is the amount of overlap divided by the area of the larger rectangle.

A table of thresholded match scores is also created where regions with match scores exceeding a user given threshold are assigned a value of one and those that do not are assigned zero. The tables of match scores can be visualized as listing the ground truth

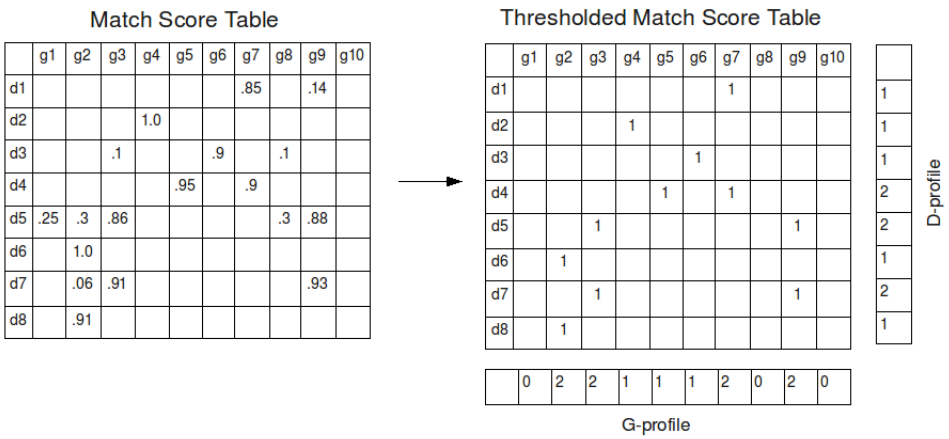


Figure 3.2: Example of Match Score tables - actual value on the left, thresholded on the right - and G and D-profiles. Taken from page 851 of [24].

regions along one direction (i.e., horizontal) and the detected regions along the other (i.e., vertical). If one were to sum the thresholded match scores for each region in each direction, Ground Truth and Detected profiles could be constructed for each of the regions. An illustration of the tables and the G/D-profiles is shown in Figure 3.2.

Now that the groundwork has been laid, counts of the one-to-one, many-to-one, and one-to-many matches can be calculated. First, the easy one-to-one matches are counted by adding up the thresholded match scores equal to one that have corresponding G and D-profiles of one, meaning they are perfect matches. For each case meeting this criteria, the corresponding G and D-profiles are set to -1 so that they are not reconsidered.

The next step is to calculate the one-to-one matches where there are multiple detected regions corresponding to given ground truth regions. Initially the regions with thresholded match scores and D-profiles of one, but G-profiles greater than one (indicating multiple matches) are placed into a candidate pool. The candidates for each ground truth region are then compared and the one with the highest actual

match score is selected as the matching one. After this, regions matching the above criteria, with the exception that the D-profile must be greater than one, are considered and selected in the same fashion. In both cases, the G and D-profiles are set to -1 upon selection of the match and the profiles of the runners up are decremented by one.

Following the resolution of many-to-one detected regions, the one-to-many detected regions are resolved in a similar fashion. In this case, the best candidates with D-profiles equal or exceeding two and G-profiles greater than zero are selected. Then, the opposite cases are considered, where D-profiles are greater than zero and G-profiles are equal to or exceed two.

After all of the one-to-one matches are tallied, the program counts the detected one-to-many and many-to-one, as well as ground truth one-to-many and many-to-one matches. This is done by pooling all of the ground truth regions with match scores above the user-given rejection threshold for each of the detected regions. If the sum of the match scores exceeds the acceptance threshold for the detected region under consideration, it is deemed a one-to-many detected match. The number of corresponding ground truth regions is then added to the ground truth many-to-one match count. The same algorithm applies to calculating ground truth one-to-many and detected many-to-one matches.

After all of this information has been extracted from the match score tables, the performance of the segmenter can be determined. The detection rate and recognition accuracies for each class are calculated by the formulas given in Section 2.1 and the overall segmentation metric is calculated using Equation 2.6.

3.2 Implementation of XML Output

Since the classification of OCRopus' segments is rendered by coloring the pixels and outputting them to a PNG image file, but the comparison program requires XML files, a module was added to OCRopus to create and output the regions in XML format.

Starting with the default RAST module of OCRopus, the columns of text and graphics boxes correspond to the "Text" and "Non-text" regions of the page. Therefore, the easiest way to output the segmentation data to an XML file is to export these rectangles. After the column separators, or gutters, are found, the horizontal and vertical rulings, along with the graphics, are extracted from the connected components of the image. At this point, the text lines are found using this data and parameters gleaned from the statistics of the connected components (i.e., the estimated height and width of a text line). Then, the text lines are sorted into reading order and the columns are found.

After fixing a couple bugs in the original implementation and making some minors edits to the "get-text-columns" function in `ocropus/ocr-layout/ocr-detect-columns.cc`, the text blocks could be defined properly (i.e., where all are included, but non-text areas are excluded). Then, the non-text regions are passed to the `hps_dump_regions` function of the new `ocropus/ocr-layout/ocr-hps-output.cc` file. This function prints a page tag to the given output file then enters a loop where the text regions are printed to the file. This is accomplished by reading each rectangle in the text array and printing its coordinates and class with the appropriate tags. A similar exercise involving the non-text array finishes the file. The code details can be found in Appendix B.

3.3 Mixed-Content RAST Algorithm

Once OCRopus was capable of producing output in the correct format, the page segmentation algorithm itself was addressed. While RAST was designed for text-only documents, it does partially support text/non-text segmentation. It divides pixels into groups of text, non-text, gutters, and rulings; however, some of the pixels can be classified as both text and non-text. It starts by binarizing the page, extracting the connected components, then determining the bounding boxes of each of them. At this point, it calculates some statistics for the boxes, including height and width, and uses them to determine whether or not each of the boxes contains a character. Those that do contain characters are called character boxes and are saved into an array.

Next, the original algorithm computes the whitespace covers (i.e., white rectangles, a.k.a., gutters) of the page using statistics dervived from the character boxes. Then, the non-text pixels, which are classified as either graphics or horizontal/vertical rulings, are extracted from the large components. All of these items, with the exception of the horizontal rulings, are placed into an array representing text-line obstacles.

Now, the basic RAST algorithm determines the text lines of the page, which for each line is the collection of contiguous character boxes on that particular line. First, the character boxes that lie within gutters are excluded, then the remaining character boxes are sorted by x-value. Each of these are then considered in terms of "matchability." Character boxes are deemed matches if they obey certain constraints, including text-line length, gap distance, and number of characters. Once the text lines have been found, they are sorted into reading order and then grouped into text blocks as described in Section 3.2.

Then, the author added new functionality to the RAST algorithm. Since one of the observed deficiencies of the algorithm was the dual labeling of pixels as shown in Figure 2.1, the first improvement was made to the text-line extraction function. Where it filters out character boxes that lie within gutters, it now also filters out character boxes that are additionally labeled non-text. So, the "character boxes" that actually contain connected components that are not characters can no longer be used to build text lines.

It was also discovered that character boxes not overlapped by the bounding boxes of any text lines, as shown in Figure 3.3, were dropped from consideration completely. So, the new algorithm now captures, closes (i.e., dilates, merges, then erodes [29]), then adds them to the non-text array of boxes. The amount of dilation is one fourth of the height of an average text-line box so that only "character boxes" in close proximity to each other are merged. Another problem was that gray areas of images were not being classified as non-text. So, isolated pixels and very small bounding boxes, such as those shown in Figure 3.4, are now saved, closed (using the same amount of dilation as the non-character boxes), and added to the non-text array as well.

Figures that contained writing, such as book covers, were being partially classified as text and partially as non-text; however, when considered as a whole, they should have been classified as one non-text region. So, routines were added to manipulate the text and non-text boxes to merge the non-text regions. Also, some sections of non-text areas were classified as text even though they did not contain text as shown previously in Figure 2.1. By examining both types of bounding boxes for several different figures, the author found that these text boxes tended to overlap non-text boxes and/or other text boxes. By identifying these overlaps, erroneous text boxes can be converted and merged into non-text regions.

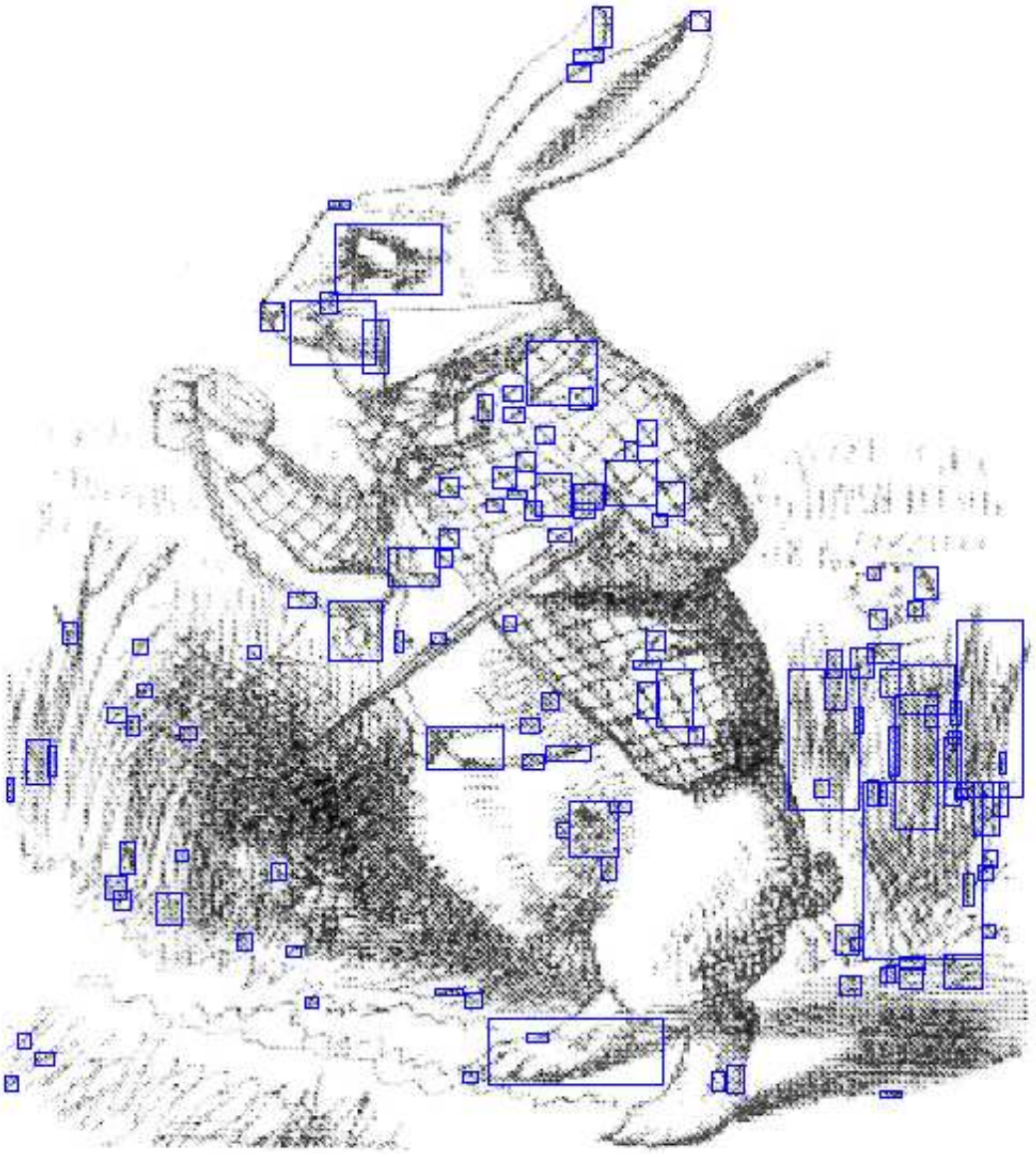


Figure 3.3: This figure illustrates character boxes that were not overlapped by any text line boxes and had been previously omitted from consideration as either text or graphics.

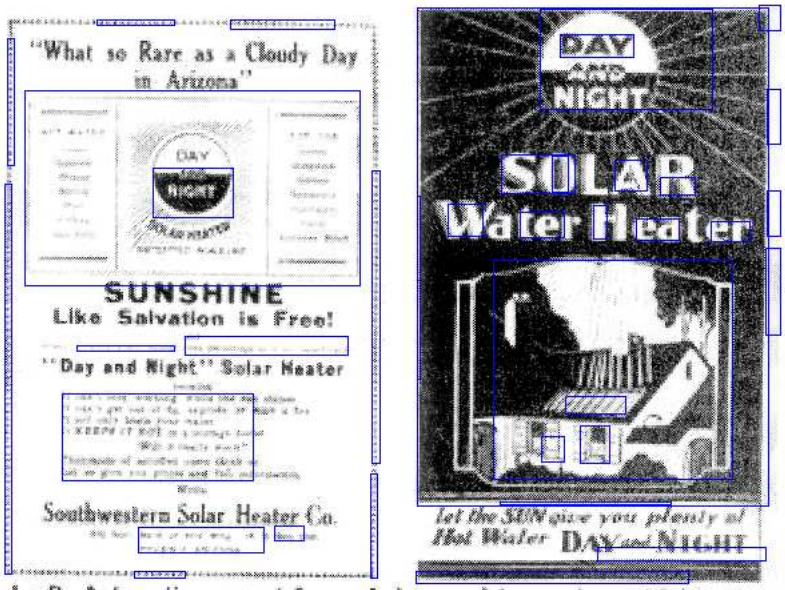
Figure 3.4: This figure illustrates small isolated character boxes that had been previously omitted from consideration as either text or non-text.

The new process outlined above starts by merging text boxes that overlap other text boxes then relabeling their union as non-text. Then, small non-text boxes (i.e., below a threshold of 10% of the square of the height of an average text line) are filtered out, since they most likely correspond to noise in the document image.

At this point, the improved algorithm iterates through a series of three steps until the array of non-text boxes is stable. First, text lines that overlap non-text boxes are reclassified as non-text. Second, non-text boxes that overlap other non-text boxes are merged, and third, non-text boxes are closed so that isolated boxes are merged. Since the second and third steps can cause non-text boxes to overlap text boxes, the first step is run again. Similarly, since the first step can cause newly created non-text boxes, to overlap other non-text boxes the second and third steps need to be repeated. Therefore, the algorithm iterates through all three steps until no more boxes are reclassified or merged.

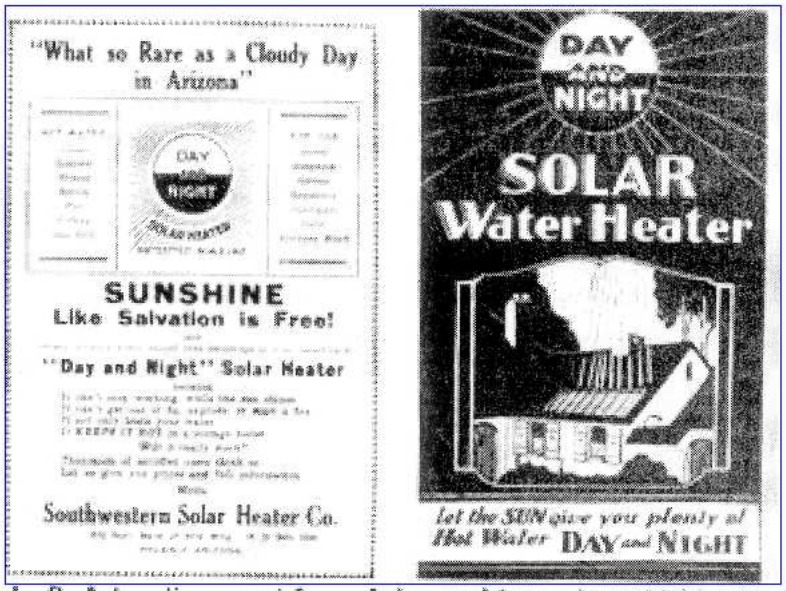
Figure 3.5 illustrates the picture of the book previously shown in Figure 2.1. The boxes outlined in blue indicate the non-text boxes prior to manipulation. Note the large number of boxes including a nested set in the upper-left corner. There are also many overlapping boxes on the right side of the figure, although they are difficult to see against the black area of the figure. Figure 3.6 shows the same figure after the text and non-text boxes have been manipulated as discussed earlier. Now there is only one non-text box, which covers the entire figure.

With the algorithm performing better on images captured at 300 DPI, the next step in the process was to evaluate it at higher and lower resolutions. Examining the program for hard-coded parameters, the author found that the minimum length of a text-line, fed to the text line extraction function, was set at thirty pixels. Since the dimensions of the character bounding boxes were calculated previously, the parameter



Left: Advertisement from Arizona Magazine, 1913.
Right: Day & Night Sales Borchure, circa 1923

Figure 3.5: Non-text boxes before converting text boxes, merging and closing.



Left: Advertisement from Arizona Magazine, 1913.
Right: Day & Night Sales Borchure, circa 1923

Figure 3.6: Non-text box after converting text boxes, merging and closing. Note that it fully encloses the figure.

was reset to a multiple of the width of this box.

Further testing using this new definition, however, revealed that the box width itself was not reliable. It was calculated by examining the histogram of the widths of the boxes and assigning the value of the first peak. Visual examination of the histograms of several images, though, indicated that the value of the first peak was much smaller than the width of a typical character. This even occurred in images not containing pictures, since the bounding boxes of periods, commas, apostrophes, and noise elements make up a significant portion of the histogram. Therefore, it is necessary to take the value of the next peak instead, which in the case of width corresponds to the right-most peak. In the case of height, it also corresponds to the right-most peak, but it is the third, not the second peak, because the second corresponds to the height of x-height characters (i.e., a, e, o, u, etc.), unless all of the text is capitalized.

Finding the correct peaks is not a simple matter. The histogram contains many local maxima that the program can mistakenly interpret as the peak of choice. Therefore, it needs to be smoothed until spurious local maxima disappear; however, it cannot be smoothed too much or the peaks themselves merge into one. So, the next step is to iteratively smooth the histogram until the expected number of peaks results. Then, the value of the right-most peak is obtained and assigned the box's height or width depending on the type of histogram. Figure 3.7 illustrates iterative smoothing until only three peaks remain.

The steps of the improved RAST algorithm are shown in Figure 3.8. There are seven original steps shown in standard font, four modified functions, which are italicized, and six new functions, which are bold. Also, the modified and new code is shown in Appendix C.

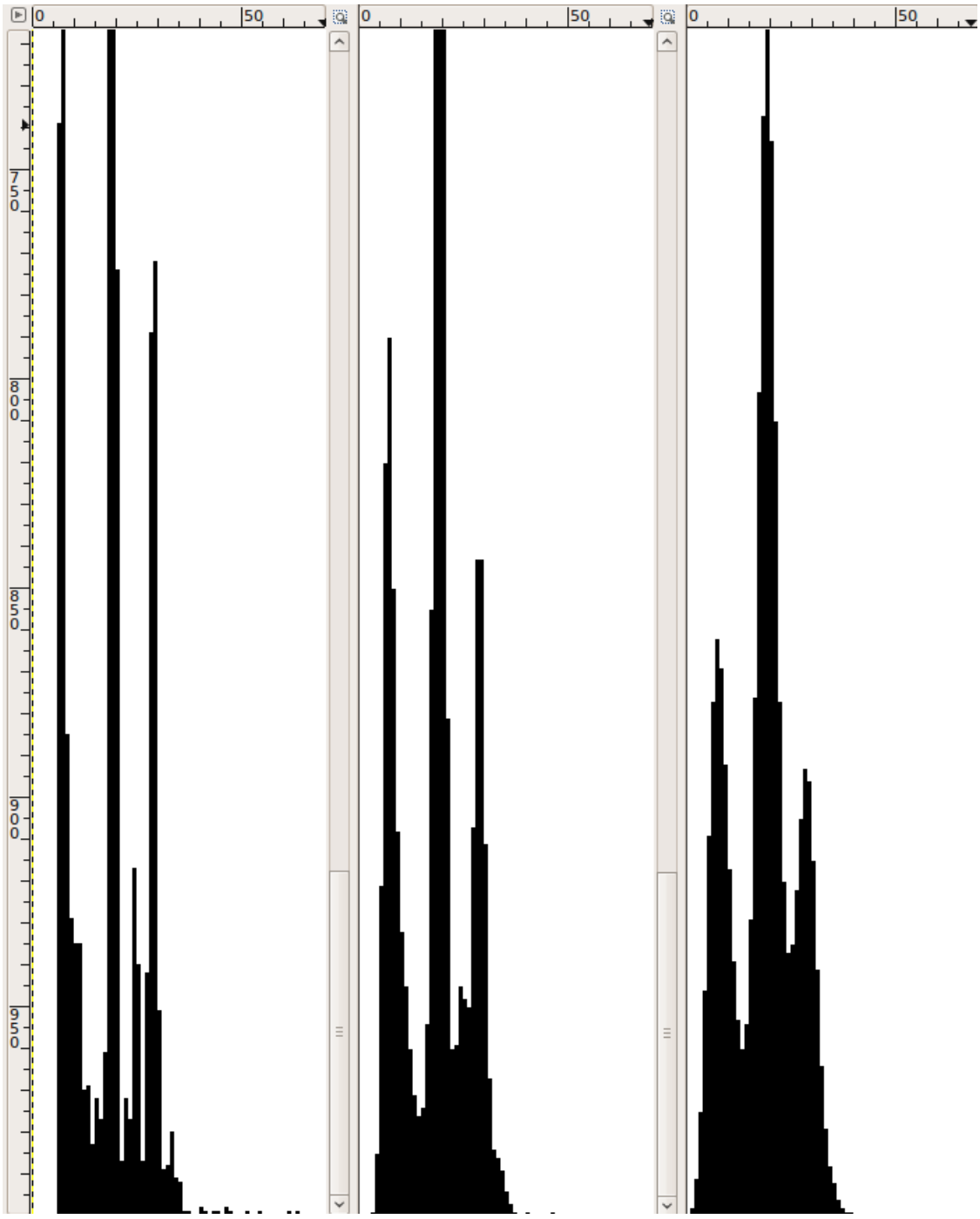


Figure 3.7: Histogram of the heights of the bounding boxes of the connected components with no smoothing (left), one iteration of smoothing (middle) and two iterations of smoothing (right). The rightmost peak corresponds to the height of ascenders (i.e. tall letters), the middle peak to the height of x-height characters (i.e. short letters) and the leftmost to the height of periods, commas, etc.

1. Binarize image.
2. Extract Connected Components (CC).
3. Calculate bounding boxes of CC's.
4. *Get character boxes and calculate statistics using iterative smoothing.*
5. Compute whitespace covers.
6. Find gutters.
7. Classify large CC's as either rulings or graphics.
8. *Extract text lines ignoring graphics pixels.*
9. **Capture, merge and reclassify rejected character boxes as graphics.**
10. **Capture, merge and reclassify very small CC's as graphics.**
11. **Merge overlapping text lines then reclassify as graphics.**
12. **Filter out very small graphics.**
13. **Merge text and graphics.**
    (a) **Merge and reclassify text lines that overlap graphics.**
    (b) **Merge overlapping graphics.**
    (c) **Close graphics rectangles.**
14. Sort text lines into reading order.
15. *Add gutters that do not overlap graphics and vertical rulings to vertical separators.*
16. *Group text lines into text regions (columns).*
17. **Group text and graphics regions in XML format.**

Figure 3.8: Steps of the improved RAST algorithm. The original steps have a standard font, the modified functions are italicized and new functions are bold.

3.4 Voronoi Page Segmentation with Classification

As written earlier, Voronoi page segmentation was not fully implemented in OCRopus. That is, users could segment document images into Voronoi zones, but they were not classified as text or non-text so could not be appropriately routed to the OCR engine. Frequent oversegmentation of zones has also been demonstrated. Additionally, since XML output is required to measure the accuracy of the page segmentation, the segmentation needed to be converted to this format as well.

Addressing all three concerns, the algorithm was extended in three steps: classification of the zones, merging of non-text zones, and clean up of any overlapping non-text regions (note that the term "zones" corresponds to geometries created by the basic Voronoi algorithm and "regions" corresponds to page segments). The original algorithm starts by binarizing the image, finding the Voronoi zones, numbering them, and creating an image of the numbered zones as depicted in Figure 3.9. At this point, the original Voronoi algorithm ends and the new algorithm developed by the author begins. The first step of the new algorithm is to save the interior zone boundary lines into another image as shown in Figure 3.10.

The new algorithm continues by extracting the connected components of the original image and identifying the character boxes as in RAST. The non-overlapping character boxes are saved into an array to be used for text classification; whereas, the overlapping character boxes are considered later as non-text entities. For each zone, the character boxes located in the extreme upper, lower, left-most and right-most portions of the zone are found and used to create the smallest rectangular region as depicted in Figure 3.11, called the "text block." Then, the zones that contain "text blocks" are passed to a function that determines whether or not the blocks really

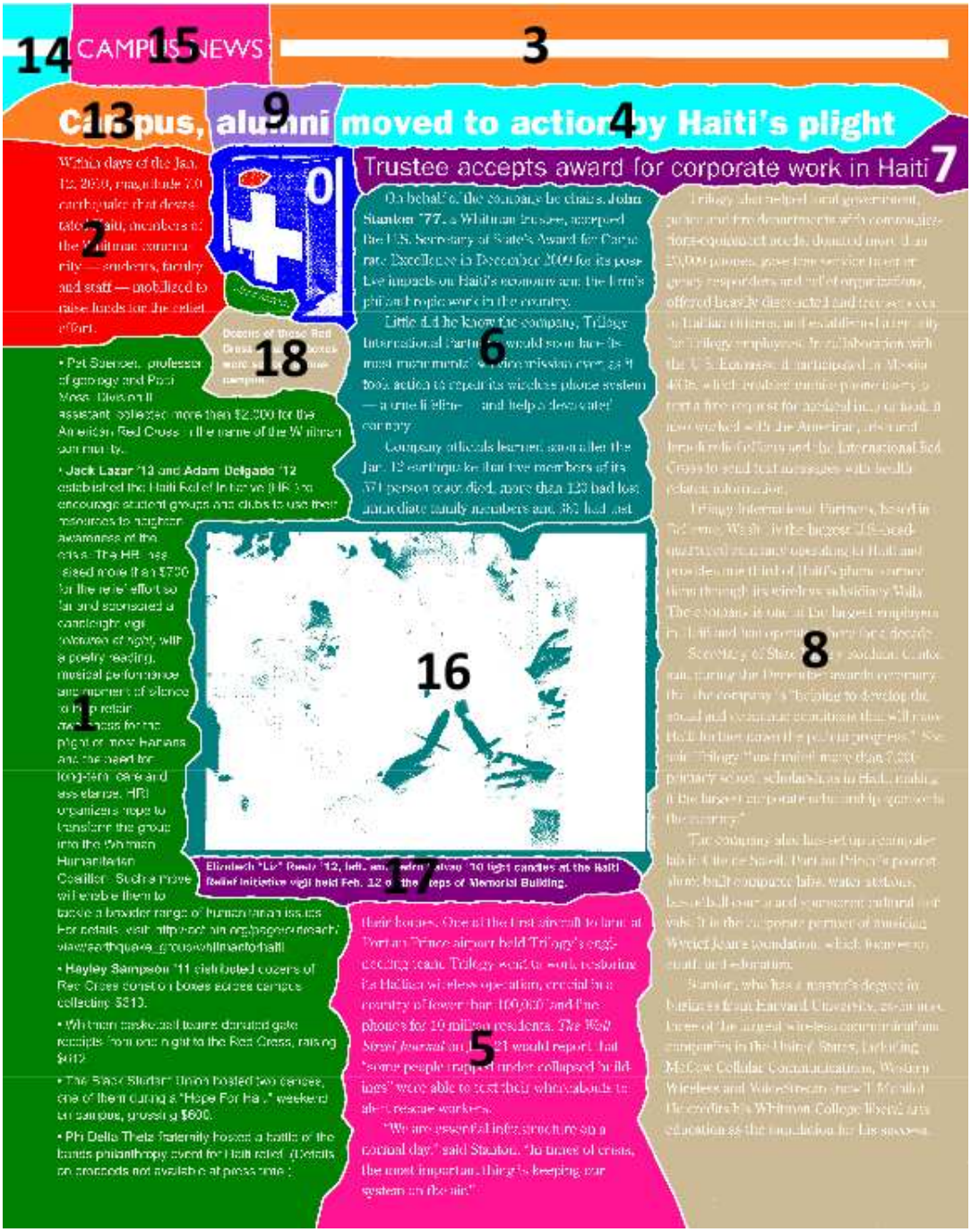


Figure 3.9: The numbered Vornonoi zones. The histograms in Figures 3.12-3.14 correspond to tan text zone number #8.

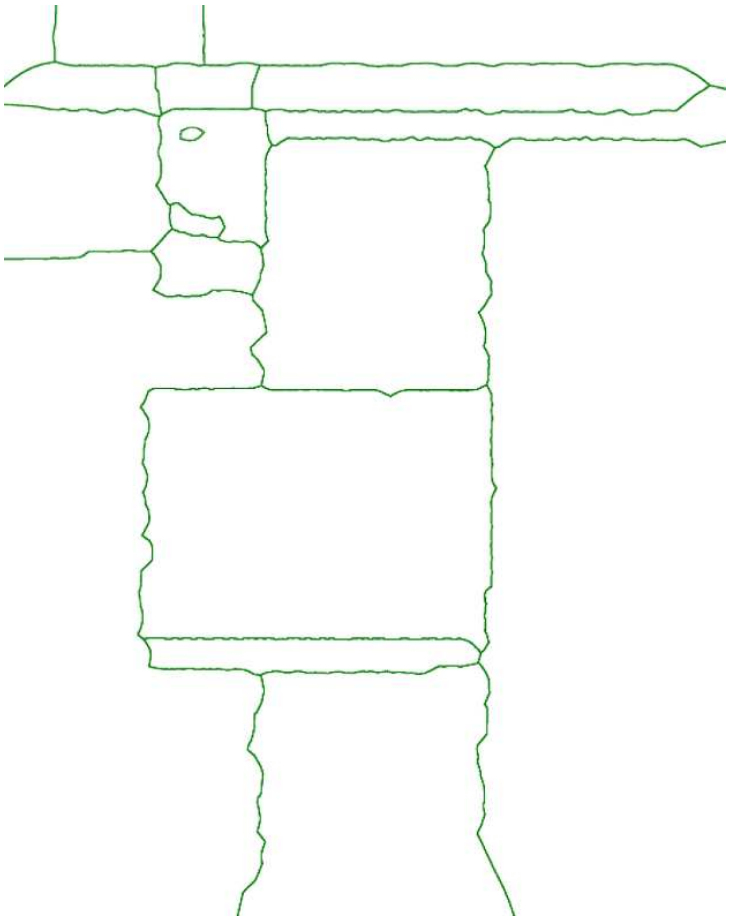


Figure 3.10: The Vornonoi lines.

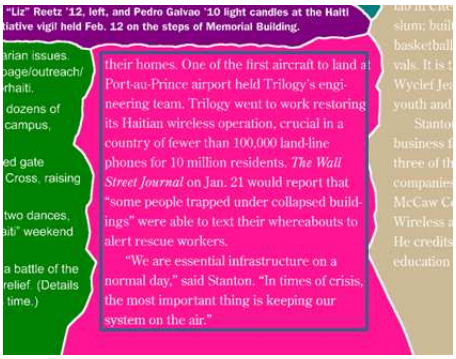


Figure 3.11: The "text rectangle" of an unclassified zone.

contain text, and based on this information, classify the zone as text or non-text.

The classification algorithm begins by creating a histogram of the locations of the lower-left corners (y0-values) of the character boxes so that it can determine the average location (or y-value) of each text line. A section of the histogram obtained from zone #8 of Figure 3.9 is shown in Figure 3.12. Notice that there exist shorter peaks to the left of each major peak. These correspond to the y0-values of descenders (i.e., letters that extend below the line like g, j, y). Since these values do not represent the location of the line, they need to be discarded, but in order to do this, the threshold under which they exist needs to be determined. This is done in a four-step process developed by the author.

First, the histogram is smoothed once, as shown in Figure 3.13, and the values of the peaks are found. Note that these values correspond to the number of occurrences of each y0-value, not the y0-values themselves. The histogram of these numbers (Figure 3.14) contains two prominent peaks: the one on the right represents the number of occurrences of the y0-values of letters sitting on the line and the one on the left represents the number of occurrences of the y0-values of letters extending below the line. Since the former is the desired parameter, the value of the right-most

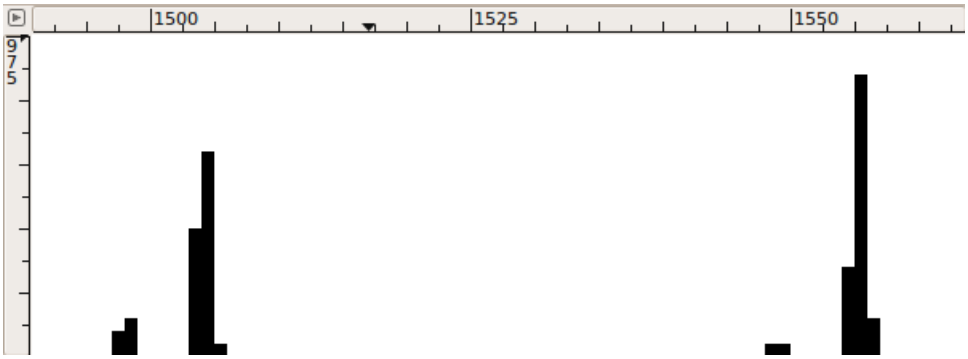


Figure 3.12: Section of the histogram of the y0-values of the character boxes of zone #8 in Figure 3.9. The peaks to the left correspond to letters extending below the line and the peaks to the right correspond to letters sitting on the line.

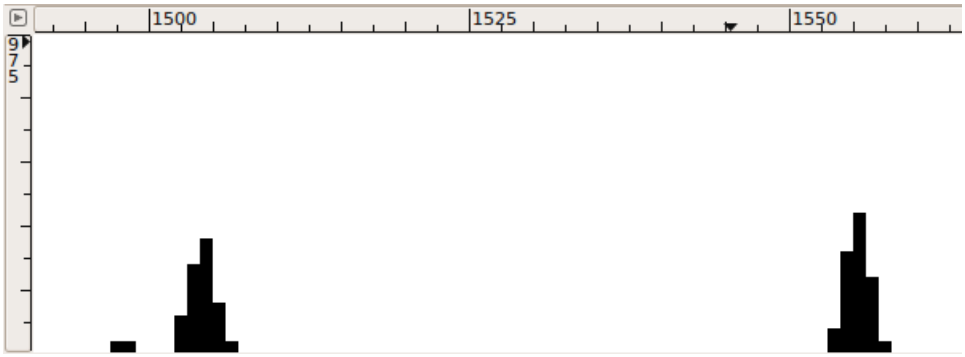


Figure 3.13: Section of the smoothed histogram of Figure 3.12.

peak is selected, which is twelve in this case. Half of this value is then used as the threshold for finding the peaks of the original histogram.

Once the y-values of the text lines have been found, the character boxes lying within a certain distance of each line (i.e., the width of an average character box) are found. For each line, the widths of the associated character boxes are summed and the x-values over which they extend is calculated. Densities for each line are then determined as the sum of the widths of the character boxes divided by their x-extent. If 80% of the lines have densities exceeding 50%, the zone is classified as text.

After the zones have been classified, the non-text ones are merged. The pixels of

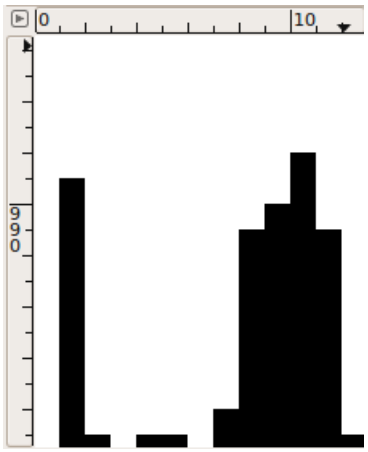
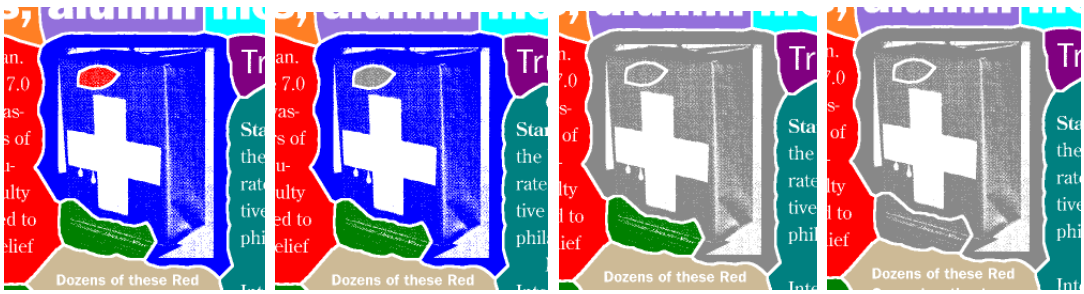


Figure 3.14: Histogram of the peaks of the y0-values of the histogram of Figure 3.13. In this example, for each of the lines, the median number of occurrences of the main y0-value is twelve.

each zone are placed into an array and their perimeters and found by dilating the Voronoi lines and ANDing them with the zone pixels. These pixels are then placed into another array. At this point, one of the non-text zones is selected and its non-text neighbors are merged with it recursively.

Part (a) of Figure 3.15 shows an oversegmented non-text region where the selected zone is colored red. To find its neighbors, the extreme upper, lower, left-most, and right-most perimeter pixels are identified and the pixels in the directions of the border are explored. For example, when the top pixel is under consideration, the pixels directly above it are explored. Since the width of the Voronoi lines are five pixels, the first five or so will correspond to the line; however, at some point after this, the exploration will encounter a pixel in a different zone. Based on this information, the identity of the neighbor is found, after which its label is updated to match the first zone's.

The remainder of Figure 3.15 depicts the relabeling of zone neighbors. This transformation occurs recursively until all of the non-text neighbors have been evaluated.



(a) (b) (c) (d)

Figure 3.15: Zone coloring of non-text relabeling process. a) Initial zone coloring, b) after the smallest has been relabeled, c) after its neighbor has been relabeled and d) after all of the neighboring non-text zones have been relabeled.

At this point, the next non-text zone that has not been evaluated is considered and its neighbors converted to its zone number, and so on.

Following the merging of non-text zones, the algorithm enters the clean-up phase. This is most easily done in rectangle space rather than pixel space since it involves merging overlapping rectangles. So, the upper, lower, left-most, and right-most pixels of each zone are found and used to define the inner rectangles.

The first step of the clean-up addresses all of the text rectangles that are completely covered by non-text rectangles. This is done by iterating through the rectangles and checking for complete overlaps. Completely covered text rectangles are simply deleted. The next step is to check for the opposite: resolve all non-text rectangles that are completely covered by text rectangles. In this case, the encompassing text rectangles are relabeled as non-text and the covered non-text rectangles are deleted.

The remaining steps address figures that have been merged across column boundaries as well as text that wraps around figures. The first case, illustrated in Figures 3.16 and 3.17, consists of breaking the non-text rectangle into two and removing

the text overlaps (i.e., in the upper-right and lower-left quadrants of the original non-text rectangle). Figures 3.18 and 3.19 show the second case where the oversized text rectangle is broken up into smaller rectangles to avoid overlapping the figure.

Once the algorithm was completed, it was tested at resolutions other than 300 DPI. At 200 DPI, the performance was slightly lower, but not appreciably and could be attributed to the loss of detail in the file; however, at 600 DPI, the performance dropped dramatically and was traced to the hard-coded parameter used to define noise pixels in the document. That parameter was changed to a fraction (1/326,774, which was determined based on the hard-coded value for 300 DPI) of the number of pixels on the page after which the segmentation performance improved.

The steps of the extended Voronoi algorithm are shown in Figure 3.20. There are two original steps displayed in standard font and six new functions displayed as bold. Appendix D contains the code.

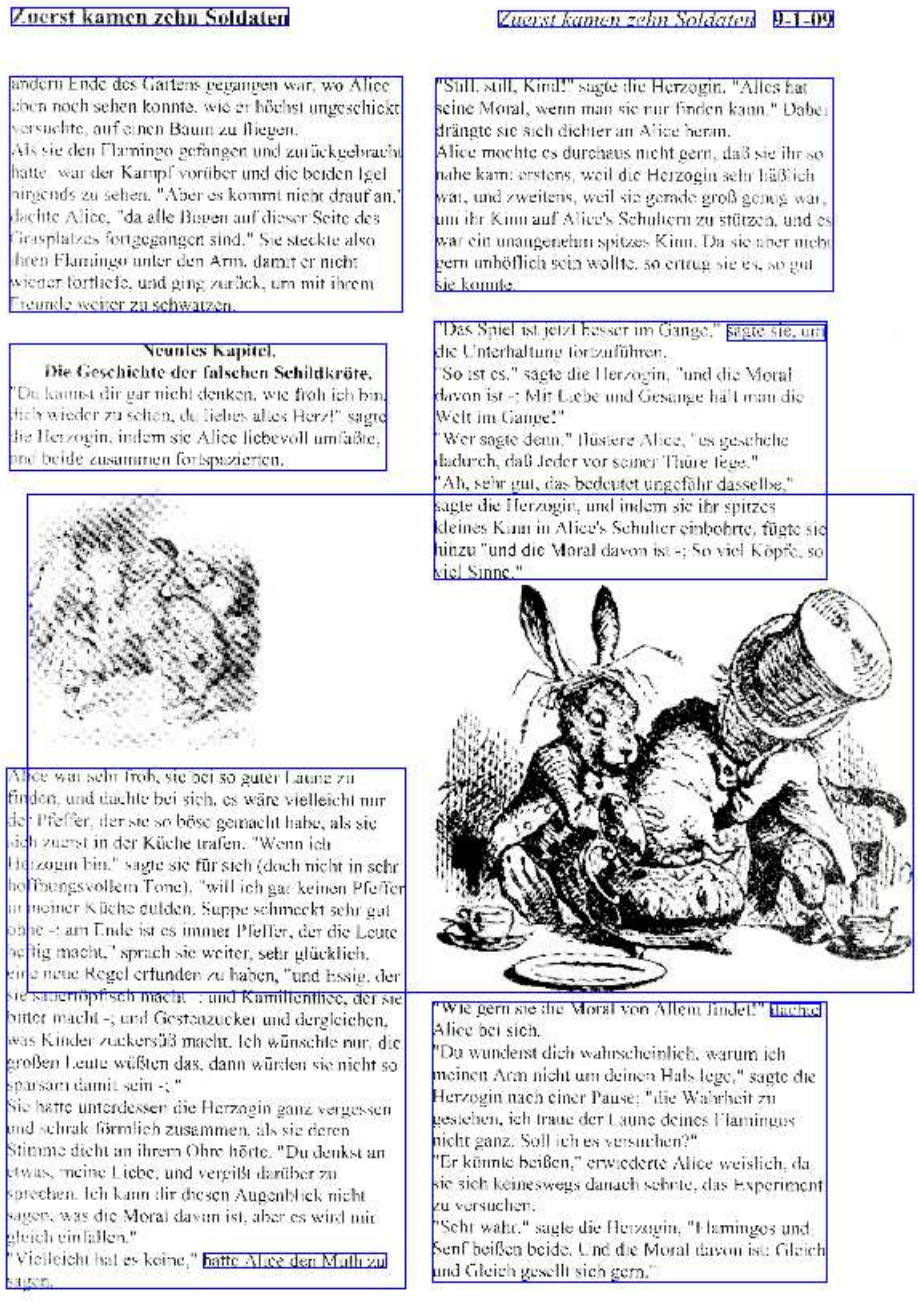


Figure 3.16: Pictures in two different columns are merged.

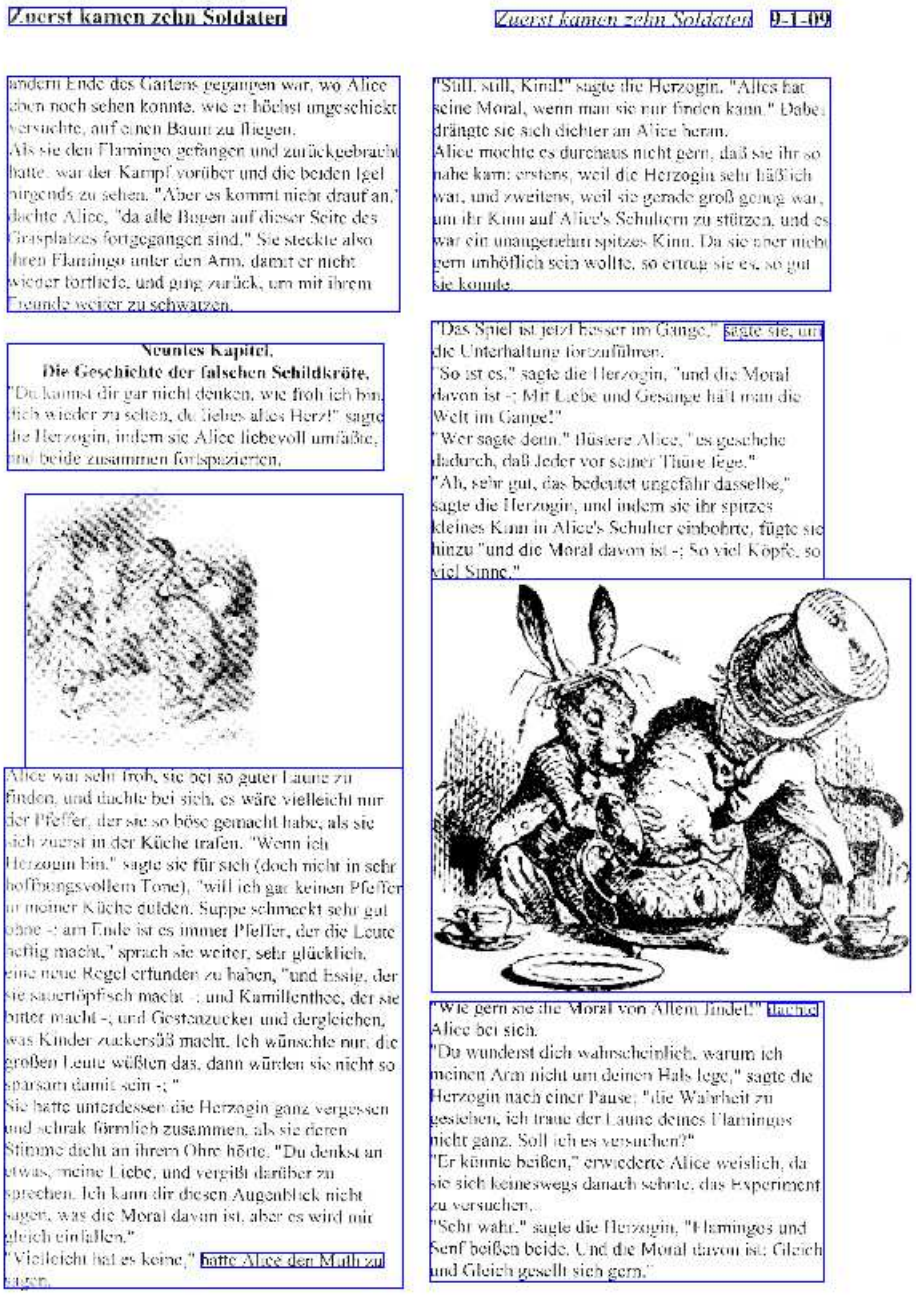


Figure 3.17: Merged graphics zone is broken in two and text overlaps removed.

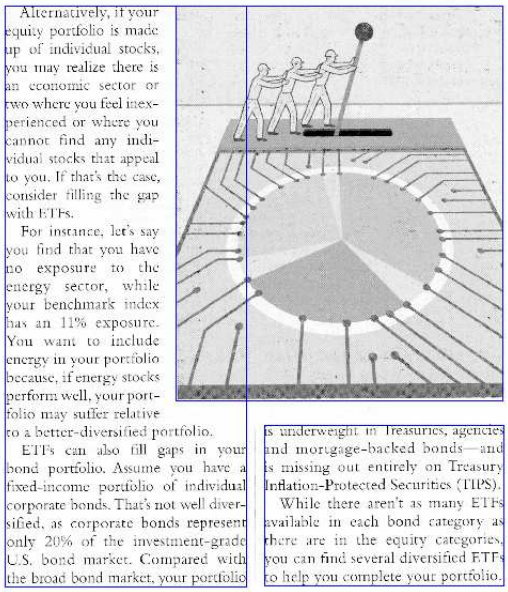


Figure 3.18: Wrap around text zone covers picture.

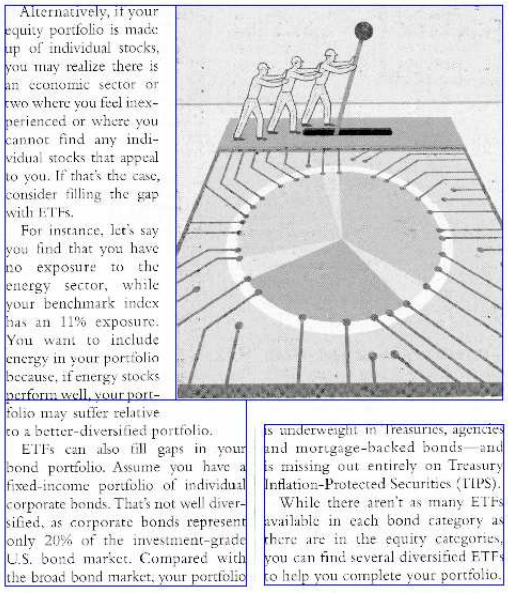


Figure 3.19: Wrap around text zone is broken into two zones.

1. Binarize image.
2. Create Voronoi area diagram then number each zone.
3. **Extract Connected Components (CC).**
4. **Calculate bounding boxes of CC's.**
5. **Get character boxes and calculate statistics using iterative smoothing.**
6. **Place non-overlapping character boxes into an array.**
7. **Zones are labeled to be text or non-text and rectangular zones are created.**
   (a) **Find the most frequently occurring y-values (text line locations).**
   (b) **Sort the boxes into text lines.**
   (c) **Calculate the density of the boxes for each text line.**
   (d) **If the density of 80% of the lines is at least 50% label as text.**
8. **Dilate the line pixels then AND them with the zone pixels to find the perimeter pixels. Place these and the zone pixels into two separate arrays.**
9. **Iterate through the non-text zones merging neighboring zones. For each non-text zone, use its perimeter pixels to explore outward and find its neighbors. Then relabel them with the original zone's label. The labeling method is recursive whereby after relabeling the given zone it finds its neighbors and relabels all of them and so on.**
10. **Clean up the segmentation.**
    (a) **Text zones which are completely overlapped by non-text zones are deleted.**
    (b) **Non-text zones which are completely overlapped by text zones are deleted and the text zones are reclassified as non-text.**
    (c) **Non-text zones which have merged across column dividers are broken so that they do not overlap neighboring text.**
    (d) **Text zones which partially overlap figures (wrap around text) are segmented.**

Figure 3.20: Steps of the extended Voronoi algorithm. The original steps have a standard font and the new functions are bold.

CHAPTER 4

TESTING AND ANALYSIS

This chapter covers the testing and analysis of the implementations of the improved RAST and Voronoi algorithms. 450 text documents were created comprising eight different types (i.e., single column, double column, etc.) and a range of resolutions. Then, their associated ground truth XML files were generated. These documents were used to test and analyze the algorithms such that the comparison program gave an overall metric and TrueViz provided a means to visualize the results. Using these tools, the algorithms were analyzed in terms of types of errors, both across and specific to particular classes, as well as a function of resolution.

4.1 Test Documents

The performance of the algorithms and commercial software was evaluated on a collection of 450 document images. Since the Bavarian documents of interest are located in Germany and have not yet been imaged, the document images evaluated for this thesis were acquired locally. The collection contains 300 hand-made documents written in the Times New Roman 12 point font saved at five different resolutions (50, 100, 200, 300, and 600 DPI) and three file formats (Tagged Image File Format (TIFF), Portable Network Graphics (PNG), and Joint Photographic Experts Group (JPEG)). The documents contain the following layouts: single column text only (10x5), double

column text only (10x5), single column text with half-tone images (10x5), double column text with half-tone images (10x5), and a mixture of single and double columns with half-tone images (10x5). The rest of the data set includes 50 pages taken from magazines (10x5) and 100 pages of technical journals that contain graphs, figures, tables and a title/abstract combination (20x5).

While the RAST and Voronoi algorithms were being developed, they were tested on a subset of the collection. Ground truth XML files were generated for each of the documents from the TIFF files so they could be compared using the comparison tool. Testing started from the first class and progressed to the most complex at a resolution of 300 DPI, using the PNG file format. Once the algorithms demonstrated acceptable performance levels at 300 DPI, they were analyzed at the remaining resolutions. If the performance dropped off, the algorithm was examined for resolution-dependent parameters and modified to be resolution independent as discussed in Section 3.3. Following the testing of the improved RAST and Voronoi algorithms, ABBYY's FineReader OCR package was evaluated to see how well a commercial program could analyze these types of layouts.

4.2 RAST Analysis

In order to assess the amount of improvement in the performance of the new RAST algorithm, the test images were first run through the original algorithm with the updated get-text-columns function (see Section 3.2). This output was then compared to the ground truth using the comparison program and two different sets of weights. The average accuracy for each class is plotted as a function of resolution in Figure 4.1 where 100% signifies perfect segmentation.

The graphs on the left illustrate the performance levels using the same weights as those used in the ICDAR 2007 Page Segmentation Competition [19] (1.0, 0.75, 0.75, 1.0, 0.75, and 0.75 for w_1 through w_6, respectively) for Equations 2.3 and 2.4; whereas, the graphs on the right depict the performance levels using the following weights: 1.0, 1.12, 1.0, 1.0, 1.0 and 1.12 for w_1 through w_6, respectively. For this algorithm, the results using the two different sets of weights are fairly similar.

Examining these plots, the single, double, and mixed column text-only pages were segmented fairly accurately, from 80-100%, by the original RAST algorithm; however, the performance level of the documents containing half-tone images peaked between 30-60% at 100 DPI, then dropped at higher resolutions. There are two issues to address here: 1) is 100 DPI a feasible resolution with which to image a document, and 2) why does the performance drop after 100 DPI? Addressing the first issue, 100 DPI is a low resolution at which most detail in a document is lost, in which case it may not even be possible to recognize the characters.

In order to assess the lowest resolution at which the OCRopus OCR engine could produce reliable output, the author scanned a single column, text-only document at eight resolutions and ran them through the OCR engine. Table 4.1 shows that at 100 DPI, the OCR engine could not recognize any of the characters. Therefore, the segmentation algorithms were not expected to perform at or below 100 DPI.

Regarding the second issue, while improving the RAST algorithm, the author found that the parameter used to specify the minimum length of the text lines was hard coded. As mentioned in Section 3.2, it was replaced by a multiple of the average character box height gleaned from the box width histograms.

The performance of the improved RAST algorithm is also shown in Figure 4.1, which displays not only better performance at 100 DPI, but better performance at

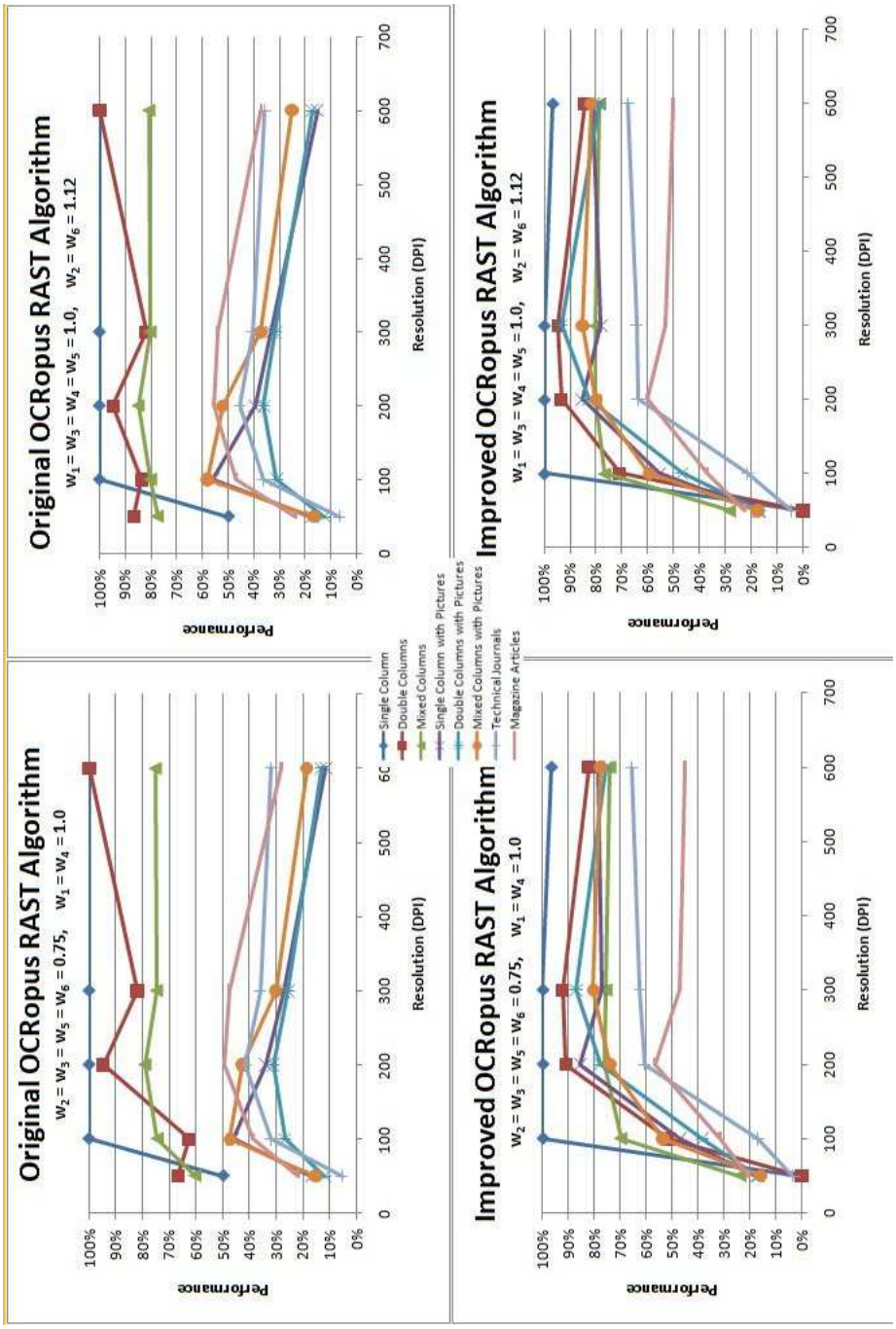


Figure 4.1: Performance of original (top) and improved (bottom) RAST algorithms with the ICDAR Page Segmentation Competition weights (left) and the weights compensated for segmentation of paragraphs (right). Higher numbers indicate higher performance.

Resolution (DPI)	OCR results
300	missed 1 line
266	missed 2 lines
240	missed 3 lines
200	missed 2 lines
150	missed 18 lines
96	no output
72	no output
50	no output

Table 4.1: The performance of the OCR engine of OCRopus on a single column, text-only document for a series of image resolutions.

higher resolutions as well. The single, double, and mixed column documents with half-tone images show the most improvement from 30-60% to 80-90%. The segmentation of the technical documents improved on the order of 25% from approximately 40% to 60-70%. They did not improve as much because they contain graphs and tables that are discontinuous and difficult to capture completely as non-text.

The axes labels of the graphs tend to be misclassified or completely dropped, and the text in the tables tends to be classified as text. Since they actually are text, one might argue that they should be classified as such anyway; however, mechanisms would be needed to be added to handle their reading order for the OCR engine. So, they were treated as non-text in this thesis. The magazine class improved the least amount from 50% to 65% due to text/non-text merging, which will be explained shortly.

Taking a closer look at the single and double column documents with half-tone images, which are similar in format to the magazine documents, three types of errors emerge. The first one is the oversegmentation of text regions. This typically happened in areas where one text line was either much shorter or slightly longer than its neighboring text lines. Figure 4.2 shows an example. Note the line in the middle

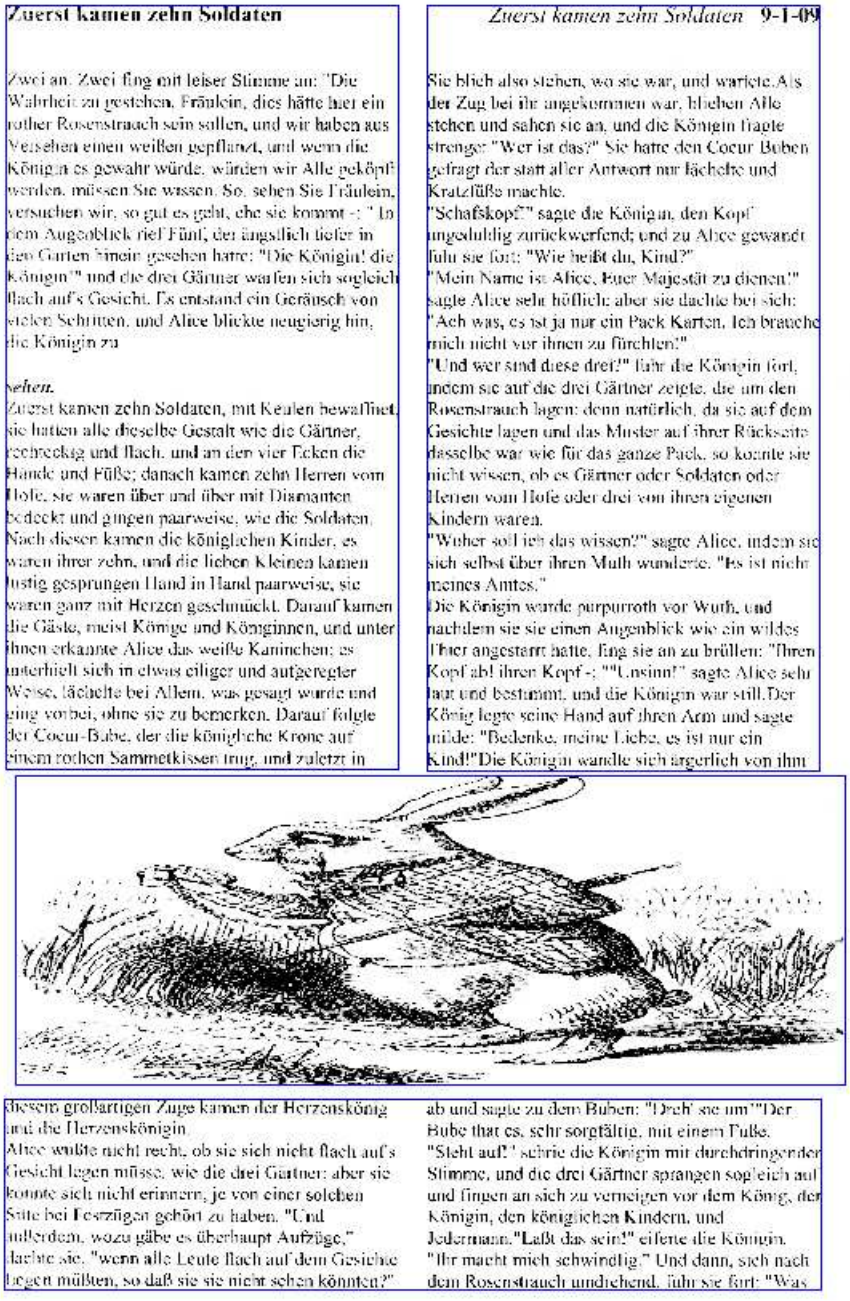
of the left column that has been defined as one region. It is slightly longer than the line above and below it, so it was not assigned to the same text column in the "get-text-columns" function.

The second type of error was the merging of text regions as depicted in Figure 4.3. In this case, as in all of the cases, they were short columns. The reason why short columns were merged is because the function to find white spaces, some of which are later turned into column separators, examines their aspect ratios and rejects those below a certain threshold. So, short columns are not separated by gutters. This could be fixed by reducing the expected aspect ratio.

The last type of error involved merging text and non-text regions. This occurred in three different cases: when text wrapped around the figure in a non-linear fashion, when the column was very narrow, and when non-text was incorrectly detected in text regions. In the first case, RAST was not designed to handle non-Manhattan geometries and XML output does not support it either, so this type of layout is beyond the scope of this thesis. Therefore, that type of error was not addressed. In the second case, RAST did not recognize the text as columns because they were too narrow to be defined as text lines. This is a limitation of the algorithm because the dimensions of text lines must pass certain threshold tests.

The last case occurred somewhat randomly in that the algorithm classified some pixels within text regions as non-text rather than text. In one of these instances, the pixels were associated with the first letter of the paragraph that was much larger than the other letters and gray rather than black. The other instance is shown in Figure 4.4 where one of the words of a text line was not included because too many small characters (i.e., -:"") separated it from the rest of the line. The word is "Ich" and is located to the left of the upper figure. It was classified as non-text and merged

Figure 4.2: Example of text oversegmentation in the improved RAST algorithm. Note the line in the middle of the left column that has been defined as one region. It is slightly longer than the line above and below it.



Zuerst kamen zehn Soldaten

Zwei an. Zwei fing mit leiser Stimme an: "Die Wahrheit zu gestehen, Fräulein, dies hätte hier ein rother Rosenstrauch sein sollen, und wir haben aus Versehen einen weißen gepflanzt, und wenn die Königin es gewahr würde, würden wir Alle geköpft werden, müssen Sie wissen. So, sehen Sie Fräulein, versuchen wir, so gut es geht, ehe sie kommt -; " In dem Augenblick rief Fünf, der ängstlich tiefer in den Garten hinein gesehen hatte: "Die Königin! die Königin!" und die drei Gärtner warfen sich sogleich flach auf's Gesicht. Es entstand ein Geräusch von vielen Schritten, und Alice blickte neugierig hin, die Königin zu

sehen.

Zuerst kamen zehn Soldaten, mit Keulen bewaffnet, sie hatten alle dieselbe Gestalt wie die Gärtner, rechteckig und flach, und an den vier Ecken die Hände und Füße; danach kamen zehn Herren vom Hofe, sie waren über und über mit Diamanten bedeckt und gingen paarweise, wie die Soldaten. Nach diesen kamen die königlichen Kinder, es waren ihrer zehn, und die lieben Kleinen kamen lustig gesprungen Hand in Hand paarweise, sie waren ganz mit Herzen geschmückt. Darauf kamen die Gäste, meist Könige und Königinnen, und unter ihnen erkannte Alice das weiße Kaninchen: es unterhielt sich in etwas eiliger und aufgeregter Weise, lächelte bei Allem, was gesagt wurde und ging vorbei, ohne sie zu bemerken. Darauf folgte der Coeur-Bube, der die königliche Krone auf einem rothen Sammetkissen trug, und zuletzt in

Zuerst kamen zehn Soldaten **9-1-09**

Sie blieb also stehen, wo sie war, und wartete. Als der Zug bei ihr angekommen war, hielten Alle stehen und sahen sie an, und die Königin fragte strenger: "Wer ist das?" Sie hatte den Coeur Buben gefragt der statt aller Antwort nur lächelte und Kratzfüße machte.

"Schafskopf!" sagte die Königin, den Kopf ungeduldig zurückwerfend; und zu Alice gewandt fuhr sie fort: "Wie heißt du, Kind?"

"Mein Name ist Alice, Euer Majestät zu dienen!" sagte Alice sehr höflich; aber sie dachte bei sich: "Ach was, es ist ja nur ein Pack Karten, Ich brauche mich nicht vor ihnen zu fürchten!"

"Und wer sind diese drei?" fuhr die Königin fort, indem sie auf die drei Gärtner zeigte, die um den Rosenstrauch lagen: denn natürlich, da sie auf dem Gesichte lagen und das Muster auf ihrer Rückseite dasselbe war wie für das ganze Pack, so konnte sie nicht wissen, ob es Gärtner oder Soldaten oder Herren vom Hofe oder drei von ihren eigenen Kindern waren.

"Woher soll ich das wissen?" sagte Alice, indem sie sich selbst über ihren Muth wunderte. "Es ist nicht meines Amtes."

Die Königin wurde purpurroth vor Wuth, und nachdem sie sie einen Augenblick wie ein wildes Thier angestarrt hatte, fing sie an zu brüllen: "Ihren Kopf ab! ihren Kopf -; ""Unsinn!" sagte Alice sehr laut und bestimmt, und die Königin war still. Der König legte seine Hand auf ihren Arm und sagte milde: "Bedenke, meine Liebe, es ist nur ein Kind!"Die Königin wandte sich ärgerlich von ihm

diesem großartigen Zuge kamen der Herzenskönig und die Herzenskönigin.

Alice wußte nicht recht, ob sie sich nicht flach auf's Gesicht legen müsse, wie die drei Gärtner: aber sie konnte sich nicht erinnern, je von einer solchen Sitte bei Festzügen gehört zu haben. "Und außerdem, wozu gäbe es überhaupt Aufzüge," dachte sie, "wenn alle Leute flach auf dem Gesichte liegen müßten, so daß sie sie nicht sehen könnten?"

ab und sagte zu dem Buben: "Dreh' sie um!"Der Bube that es, sehr sorgfältig, mit einem Fuße.

"Steht auf!" schrie die Königin mit durchdringender Stimme, und die drei Gärtner sprangen sogleich auf und fingen an sich zu verneigen vor dem König, der Königin, den königlichen Kindern, und Jedermann."Laßt das sein!" eifrte die Königin. "Ihr macht mich schwindlig." Und dann, sich nach dem Rosenstrauch umdrehend, fuhr sie fort: "Was

Figure 4.3: Example of column merging in the improved RAST algorithm. Note the diminutive height of the merged columns at the bottom of the page.

with the neighboring text-lines, which were subsequently merged with the figure. Since this example does not represent realistic punctuation this type of error was ignored.

In conclusion, while the expansion of the RAST algorithm improved its performance significantly, it still has some limitations. The root of the problem stems from the fact that parameters are needed to set length requirements of text lines and gutters. If the layout of a document does not conform to these criteria, it is not segmented correctly.

4.3 Voronoi Analysis

Since the basic Voronoi algorithm did not include zone classification, no measurements could be taken to assess the accuracy of the original segmentation; however, the images shown in Section 2.4 indicate that the figures were oversegmented. The set of documents described in Section 4.1 was run through the extended algorithm and compared to the ground truth for a range of resolutions. Figure 4.5 illustrates the performance of the algorithm alongside that of RAST using the two sets of weights mentioned in Section 4.2.

In this case, the results are markedly different for the two weight sets. For the balanced weights used in the ICDAR Page Segmentation Competition, the overall performance is lower than that of the other set. It is also much tighter in terms of variation between classes. This is because the second set of weights was tuned to avoid penalizing oversegmented text; however, it was not as effective in documents that contained half-tone images.

Figures 4.6 and 4.7 illustrate how the segments were defined in the ground truth

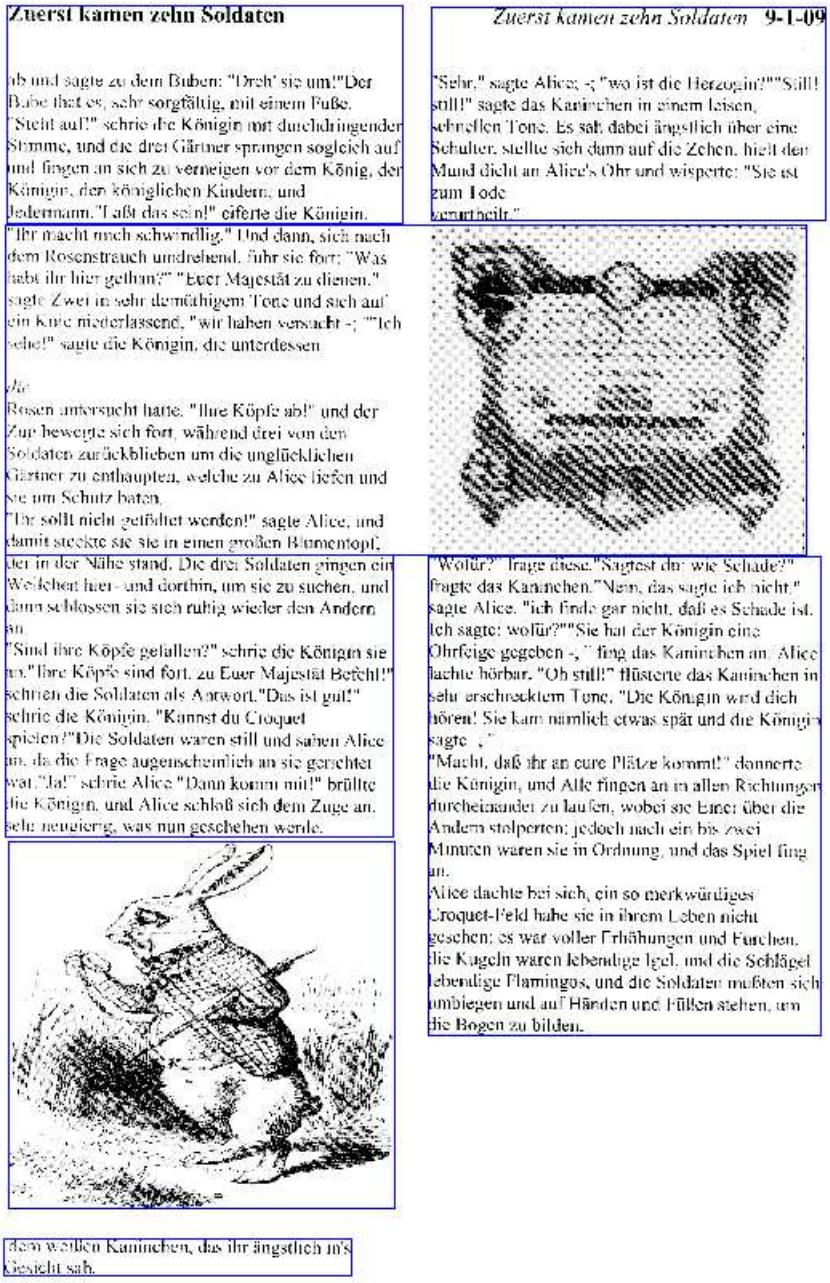


Figure 4.4: Example of text-image merging in the improved RAST algorithm. Note the "Ich" word to the left of the upper figure separated from the rest of the text on the line with -:"".

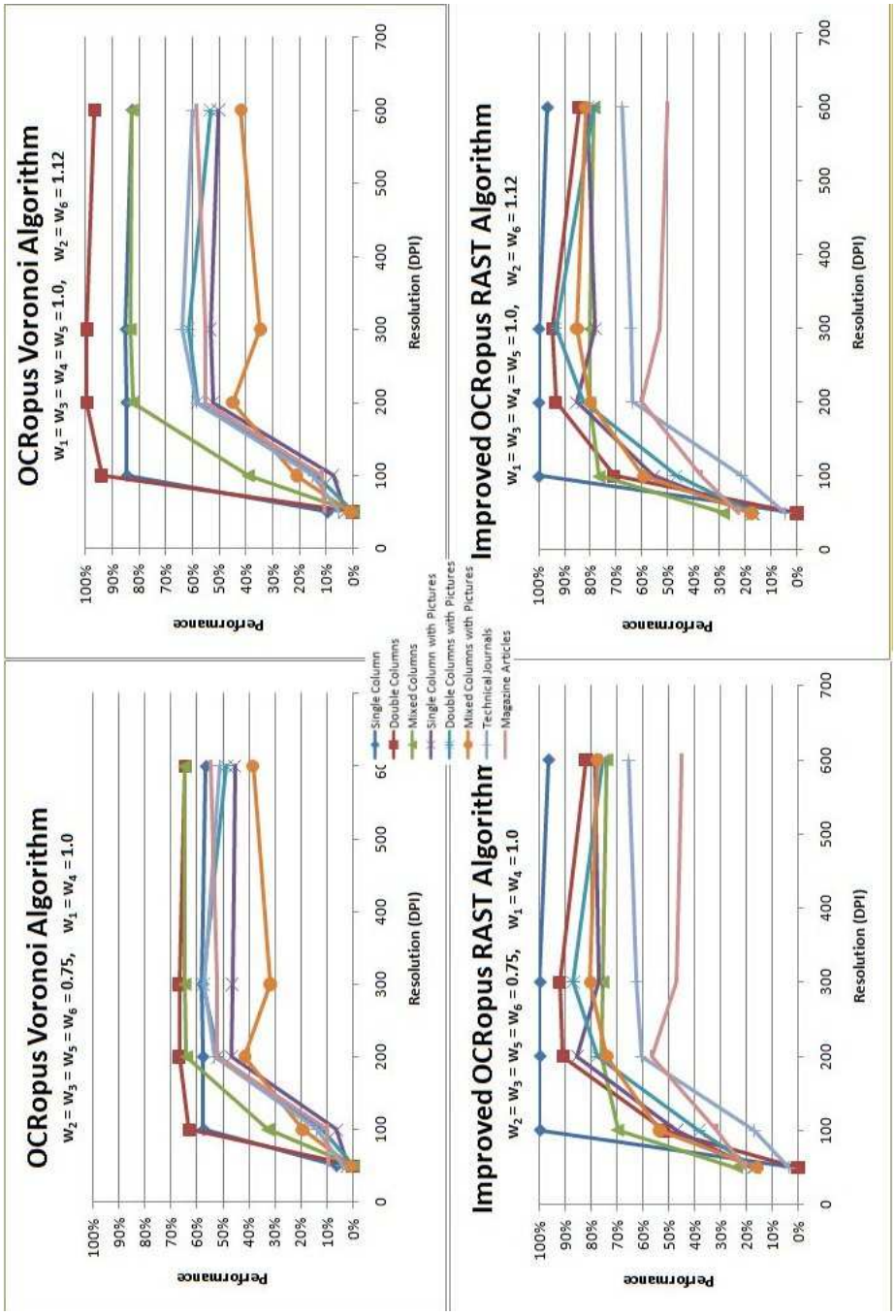


Figure 4.5: Performance of the Voronoi algorithm (top) and the improved RAST algorithm (bottom) with the ICDAR Page Segmentation Competition weights (left) and the weights compensated for segmentation of paragraphs (right).

and by the Vornonoi segmenter for one of the document images. Note that the entire document is one region in the ground truth, but the Voronoi algorithm assigns each space-separated paragraph its own region. The performance measurement returned by the comparison program for the RAST and Voronoi algorithms for this document were 100%/100% and 55%/81%, respectively, for the ICDAR and custom weights, demonstrating a higher level of performance with the custom weights.

Even though the Voronoi algorithm classified the regions correctly, it took a performance hit for segmenting these regions. Since the paragraphs are separated by spaces, though, they should have been separated in the ground truth as well, but the author did not know this at the time it was created. Therefore, this drop in performance can be attributed to the format of the ground truth rather than the Voronoi algorithm.

Compared to the RAST algorithm, in terms of overall metrics, Voronoi did not perform as well. With respect to resolution, the Voronoi algorithm performed essentially the same at 200 and 300 DPI with a small drop at 600 DPI. Also, while the two column text-only documents segmented at close to 100% accuracy, the Voronoi algorithm did not segment the single column and mixed column documents as well, ranging from 80% to 90%. While the weights of the comparison program were chosen to minimize the performance degradation for this reason, it did not compensate fully for all of the classes of documents.

Examining the results of the document classes that included half-tone images, all of them had similar performance measurements with the exception of the mixed columns class. In this case, the lower segmentation accuracy was either minor or could be attributed to non-Manhattan layouts. Figures 4.8 and 4.9 show the segmentation of a non-Manhattan layout (i.e., it does not have a Manhattan geometry) at 300 and

Zuerst kamen zehn Soldaten *Zuerst kamen zehn Soldaten* **9-1-09**

Alice in Wonderland (Times New Roman 11 pt 1 column)

Fünf und Sieben antworteten nichts, sahen aber Zwei an. Zwei fing mit leiser Stimme an: "Die Wahrheit zu gestehen, Fräulein, dies hätte hier ein rother Rosenstrauch sein sollen, und wir haben aus Versehen einen weißen gepflanzt, und wenn die Königin es gewahr würde, würden wir Alle geköpft werden, müssen Sie wissen. So, sehen Sie Fräulein, versuchen wir, so gut es geht, ehe sie kommt -; " In dem Augenblick rief Fünf, der ängstlich tiefer in den Garten hinein gesehen hatte: "Die Königin! die Königin!" und die drei Gärtner warfen sich sogleich flach auf's Gesicht. Es entstand ein Geräusch von vielen Schritten, und Alice blickte neugierig hin, die Königin zu sehen.

Zuerst kamen zehn Soldaten, mit Keulen bewaffnet, sie hatten alle dieselbe Gestalt wie die Gärtner, rechteckig und flach, und an den vier Ecken die Hände und Füße; danach kamen zehn Herren vom Hofe, sie waren über und über mit Diamanten bedeckt und gingen paarweise, wie die Soldaten. Nach diesen kamen die königlichen Kinder, es waren ihrer zehn, und die lieben Kleinen kamen lustig gesprungen Hand in Hand paarweise, sie waren ganz mit Herzen geschmückt. Darauf kamen die Gäste, meist Könige und Königinnen, und unter ihnen erkannte Alice das weiße Kaninchen; es unterhielt sich in etwas eiliger und aufgeregter Weise, lächelte bei Allem, was gesagt wurde und ging vorbei, ohne sie zu bemerken. Darauf folgte der Coeur-Bube, der die königliche Krone auf einem rothen Sammetkissen trug, und zuletzt in diesem großartigen Zuge kamen der Herzenskönig und die Herzenskönigin.

Alice wußte nicht recht, ob sie sich nicht flach auf's Gesicht legen müsse, wie die drei Gärtner; aber sie konnte sich nicht erinnern, je von einer solchen Sitte bei Festzügen gehört zu haben. "Und außerdem, wozu gäbe es überhaupt Aufzüge," dachte sie, "wenn alle Leute flach auf dem Gesichte liegen müßten, so daß sie sie nicht sehen könnten?" Sie blieb also stehen, wo sie war, und wartete.Als der Zug bei ihr angekommen war, blieben Alle stehen und sahen sie an, und die Königin fragte strenge: "Wer ist das?" Sie hatte den Coeur-Buben gefragt der statt aller Antwort nur lächelte und Kratzfüße machte. "Schafskopf!" sagte die Königin, den Kopf ungeduldig zurückwerfend, und zu Alice gewandt fuhr sie fort: "Wie heißt du, Kind?""Mein Name ist Alice, Euer Majestät zu dienen!" sagte Alice sehr höflich; aber sie dachte bei sich: "Ach was, es ist ja nur ein Pack Karten. Ich brauche mich nicht vor ihnen zu fürchten!""Und wer sind diese drei?" fuhr die Königin fort, indem sie auf die drei Gärtner zeigte, die um den Rosenstrauch lagen; denn natürlich, da sie auf dem Gesichte lagen und das Muster auf ihrer Rückseite dasselbe war wie für das ganze Pack, so konnte sie nicht wissen, ob es Gärtner oder Soldaten oder Herren vom Hofe oder drei von ihren eigenen Kindern waren."Woher soll ich das wissen?" sagte Alice, indem sie sich selbst über ihren Muth wunderte. "Es ist nicht meines Amtes."Die Königin wurde purpurroth vor Wuth, und nachdem sie sie einen Augenblick wie ein wildes Thier angestarrt hatte, fing sie an zu brüllen: "Ihren Kopf ab! ihren Kopf -; ""Unsinn!" sagte Alice sehr laut und bestimmt, und die Königin war still.Der König legte seine Hand auf ihren Arm und sagte milde: "Bedenke, meine Liebe, es ist nur ein Kind!"Die Königin wandte sich ärgerlich von ihm ab und sagte zu dem Buben: "Dreh' sie um!"

Der Bube that es, sehr sorgfältig, mit einem Fuße."Steht auf!" schrie die Königin mit durchdringender Stimme, und die drei Gärtner sprangen sogleich auf und fingen an sich zu verneigen vor dem König, der Königin, den königlichen Kindern, und Jedermann."Laßt das sein!" eiferte die Königin. "Ihr macht mich schwindlig." Und dann, sich nach dem Rosenstrauch umdrehend, fuhr sie fort: "Was habt ihr hier gethan?""Euer Majestät zu dienen," sagte Zwei in sehr demüthigem Tone und sich auf ein Knie niederlassend, "wir haben versucht -; ""Ich sehe!" sagte die Königin, die unterdessen die Rosen untersucht hatte. "Ihre Köpfe ab!" und der Zug

Figure 4.6: Ground truth of a single text-only document image.

Figure 4.7: Voronoi text segments of a single text-only document image.

600 DPI, respectively. The difference between the two lies in the bottom region. The higher resolution document contains several text regions; whereas the lower does not contain any. It only contains two non-text regions in this area. Since the extension of the Voronoi algorithm did not address non-Manhattan layouts, the results of this particular document image can be ignored.

For the remaining document classes containing half-tone images, three types of errors dominate: one can be attributed to the data, another to Kise's Voronoi algorithm, and the third to the text classification algorithm. Starting with the first, a number of the documents contain half-tone images in very close proximity to text, such as that shown in Figure 4.10. For documents scanned at a resolution of 300 DPI, Kise's Voronoi algorithm failed to separate the images from text when they were separated by 23 or fewer pixels. The height of a tall letter at this resolution is 28 pixels, so if the image were positioned within this distance, it might not be placed into its own region. After the Voronoi regions were defined, it was impossible for the extension of the algorithm to further segment and classify them correctly.

The second concern is similar to the first in that its root cause can be traced to Kise's Voronoi algorithm. As mentioned in Section 3.4, the most frequently occurring zoning error is the oversegmentation of text. This can be seen in titles, headers, footers and occasionally in parts of outlying sentences in paragraphs. The title shown in Figure 3.7 illustrates the phenomenon. Since this problem relates more to reading order than region classification, it was not addressed in this thesis.

The third issue identified was that some text, namely italicized and bold text, tended to be classified as images rather than text. This was due to the fact that the bounding boxes of the characters overlapped so were omitted from the zones and not considered as text. Therefore, by default they were classified non-text. While this

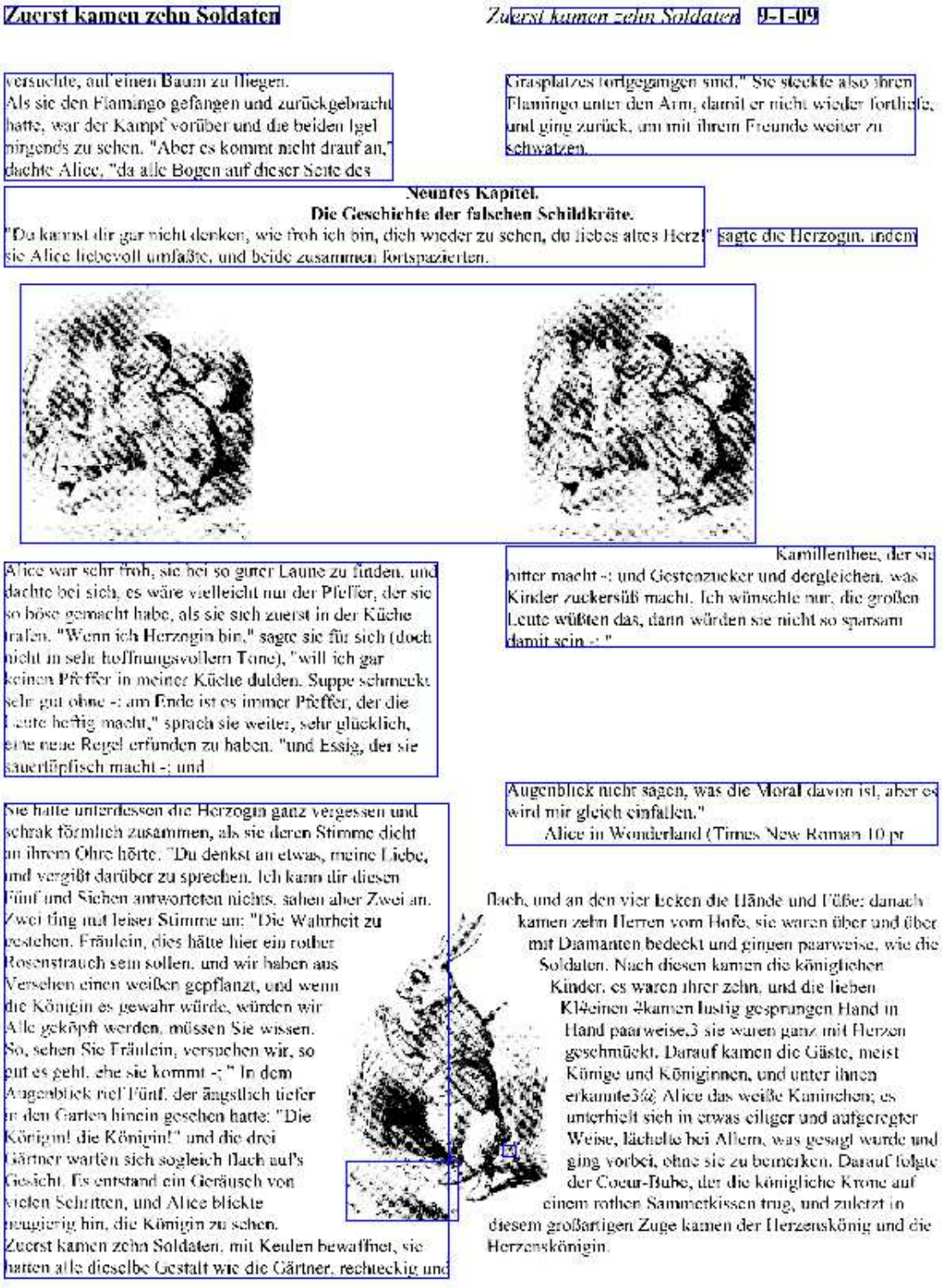


Figure 4.8: Voronoi segmentation of a mixed column document with pictures at 300 DPI. The lowest regions were classified as graphics. The accuracy of the segmentation was 37%.

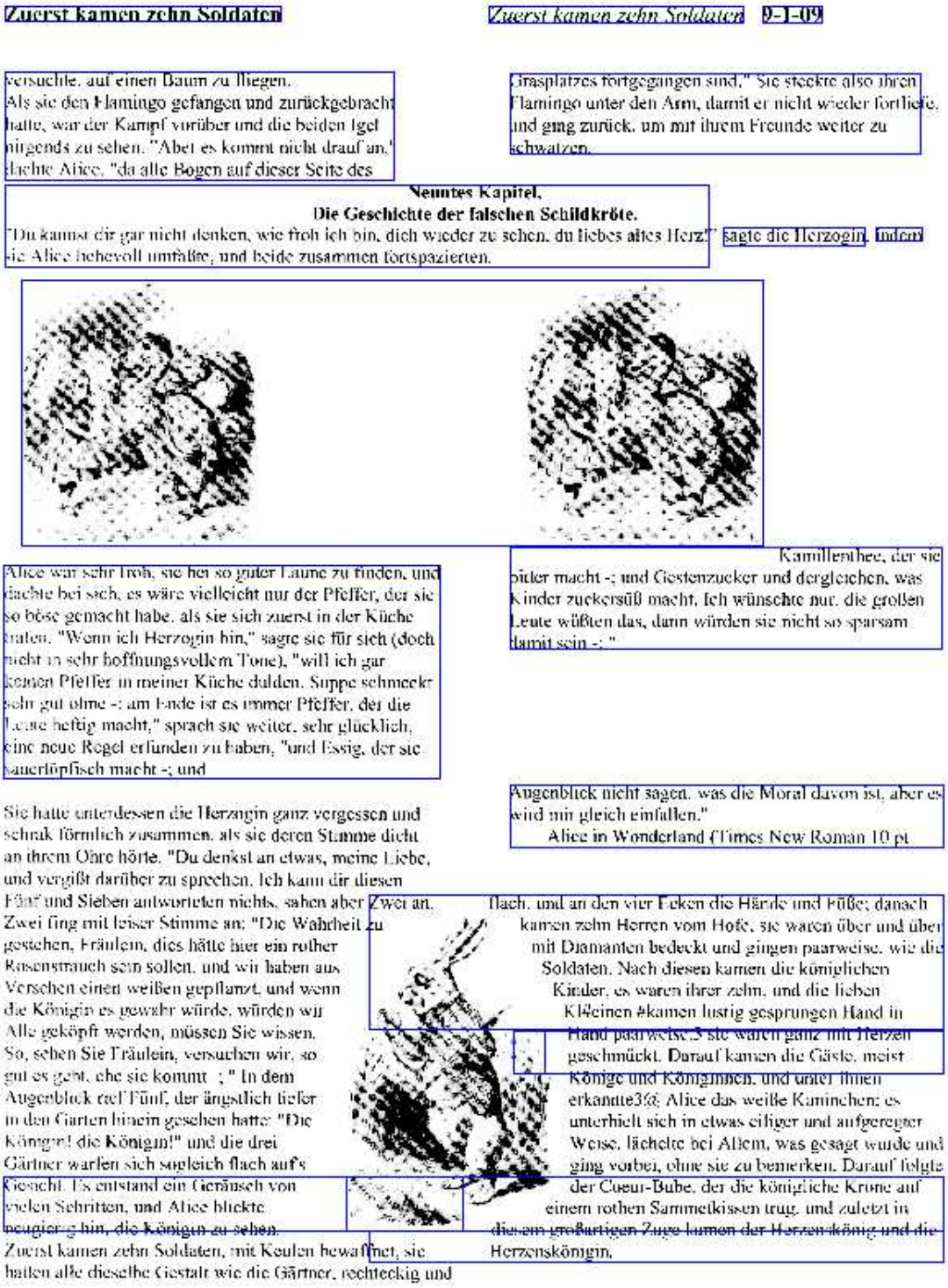


Figure 4.9: Voronoi segmentation of a mixed column document with pictures at 600 DPI. Most of the lowest regions were classified as text. The accuracy of the segmentation was 53%.

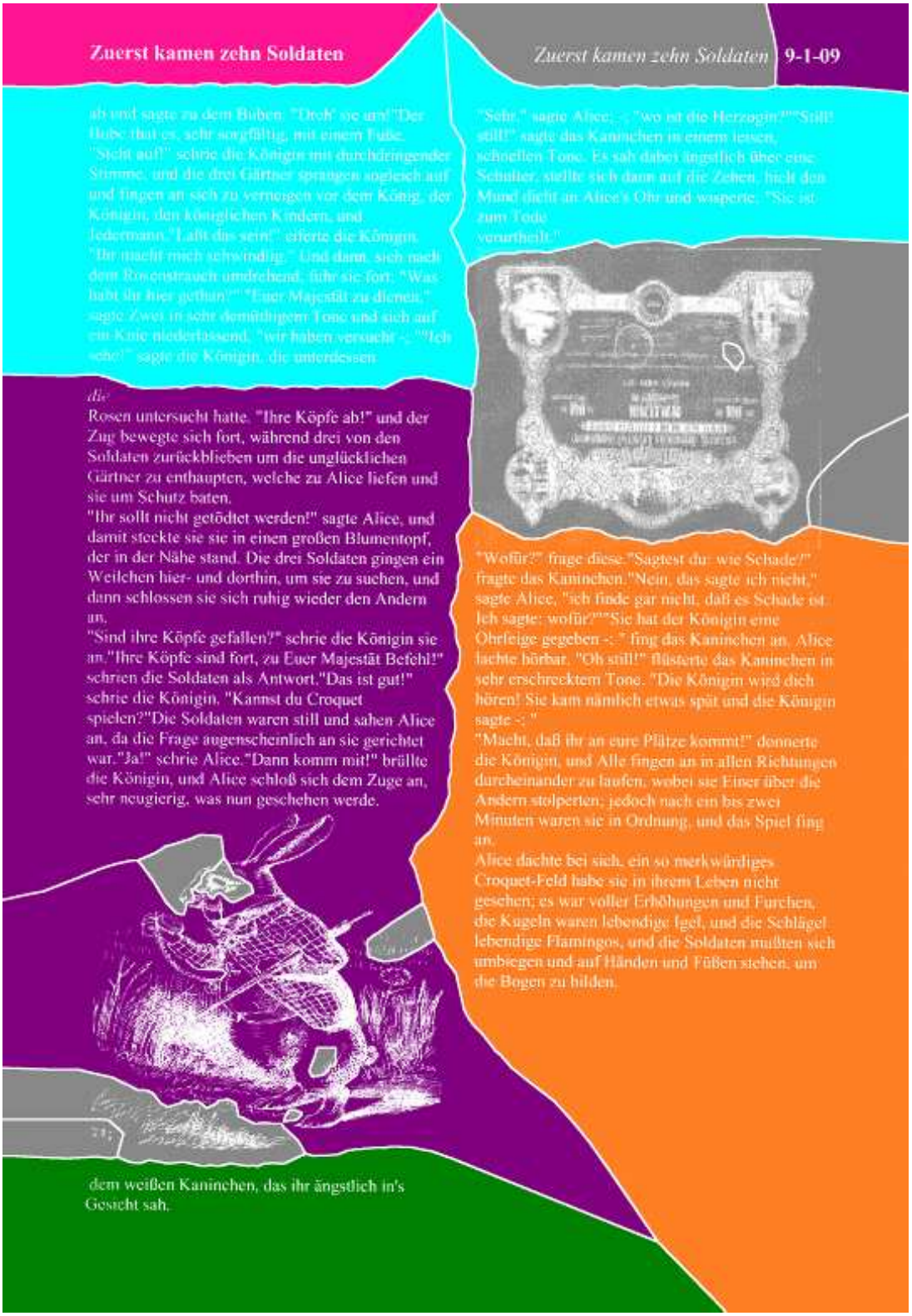


Figure 4.10: Document image where the picture is placed too close to the text to allow for correct Voronoi zoning. Note the purple text section merged with the rabbit.

was a problem sometimes, unless the entire block was italicized, it did not have a substantial impact on the performance. To fix this problem, it might be possible to add an overlap tolerance to the algorithm so that these letters are not dropped.

4.4 Commercial Package

Following the completion of the page segmentation algorithms, a commercial OCR program was evaluated for comparison. ABBYY's Fine Reader Engine 9.0 is a comprehensive layout analysis package, which not only includes image processing and layout analysis commands, but table, barcode, text-type recognition (i.e., direction, italics, underlining, etc.), and synthesis (i.e., hyperlinks, bullets, background, and text color, etc.) commands. Additionally, it can recognize 186 languages and can produce output in nine different formats.

The set of test documents described in Section 4.1 were analyzed by Fine Reader and output in XML format. Since the tags of this format did not match that of the ground truth, a program was written to convert these files to a matchable format. These were then compared to the ground truth using the program described in Section 3.1.

Figure 4.12 shows the performance of Fine Reader alongside the improved RAST and Voronoi algorithms. Comparing the two different weight classes, the performance is only slightly higher for the customized weights. Therefore, customizing the weights benefitted the Voronoi algorithm the most. This is because it segmented the text more than the other two algorithms as shown in Figures 4.6, 4.7, and 4.11, and placing higher weights on the one-to-many ground truth-to-detected region parameters results in a larger performance gain for highly segmented detected regions. As noted in

Section 4.3, though, the text of the ground truth was undersegmented as a whole.

For both cases, the performance of all classes is between 70% and 85% for all resolutions, including 100 DPI. Neither RAST nor Voronoi were able to segment as accurately at 100 DPI. At 50 DPI, the performance drops 5-10% for Fine Reader; whereas, for the other two algorithms, it essentially drops to zero. Not only does Fine Reader have a flatter response as a function of resolution, but it also has a tighter response in that all of the classes were segmented with approximately the same accuracy.

There were a couple of anomalies, though. At 50 and 100 DPI for the single column and double column classes, the performance dropped to zero. This was because the regions were classified as pictures rather than text. Also, the single column class only performed at approximately 50% throughout the range of resolutions due to the same reason: the Voronoi algorithm had a lower performance than RAST; the paragraphs were broken into individual regions, but were only represented by one region in the ground truth.

Examining the output, the predominant error appeared to be overlapping regions, which depending on how you define the ground truth, could not even be considered an error. Figure 4.13 shows one such example. Note the overlapping text and image regions. So, rather than break up the regions, Fine Reader simply overlaps them.

Zuerst kamen zehn Soldaten

Zuerst kamen zehn Soldaten **9-1-09**

Alice in Wonderland (Times New Roman 11 pt 1 column)

Fünf und Sieben antworteten nichts, sahen aber Zwei an. Zwei fing mit leiser Stimme an: "Die Wahrheit zu gestehen, Fräulein, dies hätte hier ein rother Rosenstrauch sein sollen, und wir haben aus Versehen einen weißen gepflanzt, und wenn die Königin es gewahr würde, würden wir Alle geköpft werden, müssen Sie wissen. So, sehen Sie Fräulein, versuchen wir, so gut es geht, ehe sie kommt -; " In dem Augenblick rief Fünf, der ängstlich tiefer in den Garten hinein gesehen hatte: "Die Königin! die Königin!" und die drei Gärtner warfen sich sogleich flach auf's Gesicht. Es entstand ein Geräusch von vielen Schritten, und Alice blickte neugierig hin, die Königin zu sehen.

Zuerst kamen zehn Soldaten, mit Keulen bewaffnet, sie hatten alle dieselbe Gestalt wie die Gärtner, rechteckig und flach, und an den vier Ecken die Hände und Füße; danach kamen zehn Herren vom Hofe, sie waren über und über mit Diamanten bedeckt und gingen paarweise, wie die Soldaten. Nach diesen kamen die königlichen Kinder, es waren ihrer zehn, und die lieben Kleinen kamen lustig gesprungen Hand in Hand paarweise, sie waren ganz mit Herzen geschmückt. Darauf kamen die Gäste, meist Könige und Königinnen, und unter ihnen erkannte Alice das weiße Kaninchen; es unterhielt sich in etwas eiliger und aufgeregter Weise, lächelte bei Allem, was gesagt wurde und ging vorbei, ohne sie zu bemerken. Darauf folgte der Coeur-Bube, der die königliche Krone auf einem rothen Sammetkissen trug, und zuletzt in diesem großartigen Zuge kamen der Herzenskönig und die Herzenskönigin.

Alice wußte nicht recht, ob sie sich nicht flach auf's Gesicht legen müsse, wie die drei Gärtner; aber sie konnte sich nicht erinnern, je von einer solchen Sitte bei Festzügen gehört zu haben. "Und außerdem, wozu gäbe es überhaupt Aufzüge," dachte sie, "wenn alle Leute flach auf dem Gesichte liegen müßten, so daß sie sie nicht sehen könnten?" Sie blieb also stehen, wo sie war, und wartete.Als der Zug bei ihr angekommen war, blieben Alle stehen und sahen sie an, und die Königin fragte strenge: "Wer ist das?" Sie hatte den Coeur-Buben gefragt der statt aller Antwort nur lächelte und Kratzfüße machte. "Schafskopf!" sagte die Königin, den Kopf ungeduldig zurückwerfend; und zu Alice gewandt fuhr sie fort: "Wie heißt du, Kind?""Mein Name ist Alice, Euer Majestät zu dienen!" sagte Alice sehr höflich; aber sie dachte bei sich: "Ach was, es ist ja nur ein Pack Karten. Ich brauche mich nicht vor ihnen zu fürchten!""Und wer sind diese drei?" fuhr die Königin fort, indem sie auf die drei Gärtner zeigte, die um den Rosenstrauch lagen; denn natürlich, da sie auf dem Gesichte lagen und das Muster auf ihrer Rückseite dasselbe war wie für das ganze Pack, so konnte sie nicht wissen, ob es Gärtner oder Soldaten oder Herren vom Hofe oder drei von ihren eigenen Kindern waren."Woher soll ich das wissen?" sagte Alice, indem sie sich selbst über ihren Muth wunderte. "Es ist nicht meines Amtes."Die Königin wurde purpurroth vor Wuth, und nachdem sie sie einen Augenblick wie ein wildes Thier angestarrt hatte, fing sie an zu brüllen: "Ihren Kopf ab! ihren Kopf -; ""Unsinn!" sagte Alice sehr laut und bestimmt, und die Königin war still.Der König legte seine Hand auf ihren Arm und sagte milde: "Bedenke, meine Liebe, es ist nur ein Kind!"Die Königin wandte sich ärgerlich von ihm ab und sagte zu dem Buben: "Dreh' sie um!"

Der Bube that es, sehr sorgfältig, mit einem Fuße."Steht auf!" schrie die Königin mit durchdringender Stimme, und die drei Gärtner sprangen sogleich auf und fingen an sich zu verneigen vor dem König, der Königin, den königlichen Kindern, und Jedermann."Laßt das sein!" eiferte die Königin. "Ihr macht mich schwindlig." Und dann, sich nach dem Rosenstrauch umdrehend, fuhr sie fort: "Was habt ihr hier gethan?""Euer Majestät zu dienen," sagte Zwei in sehr demüthigem Tone und sich auf ein Knie niederlassend, "wir haben versucht -; ""Ich sehe!" sagte die Königin, die unterdessen die Rosen untersucht hatte. "Ihre Köpfe ab!" und der Zug

Figure 4.11: Fine Reader text segments of a single text-only document image.

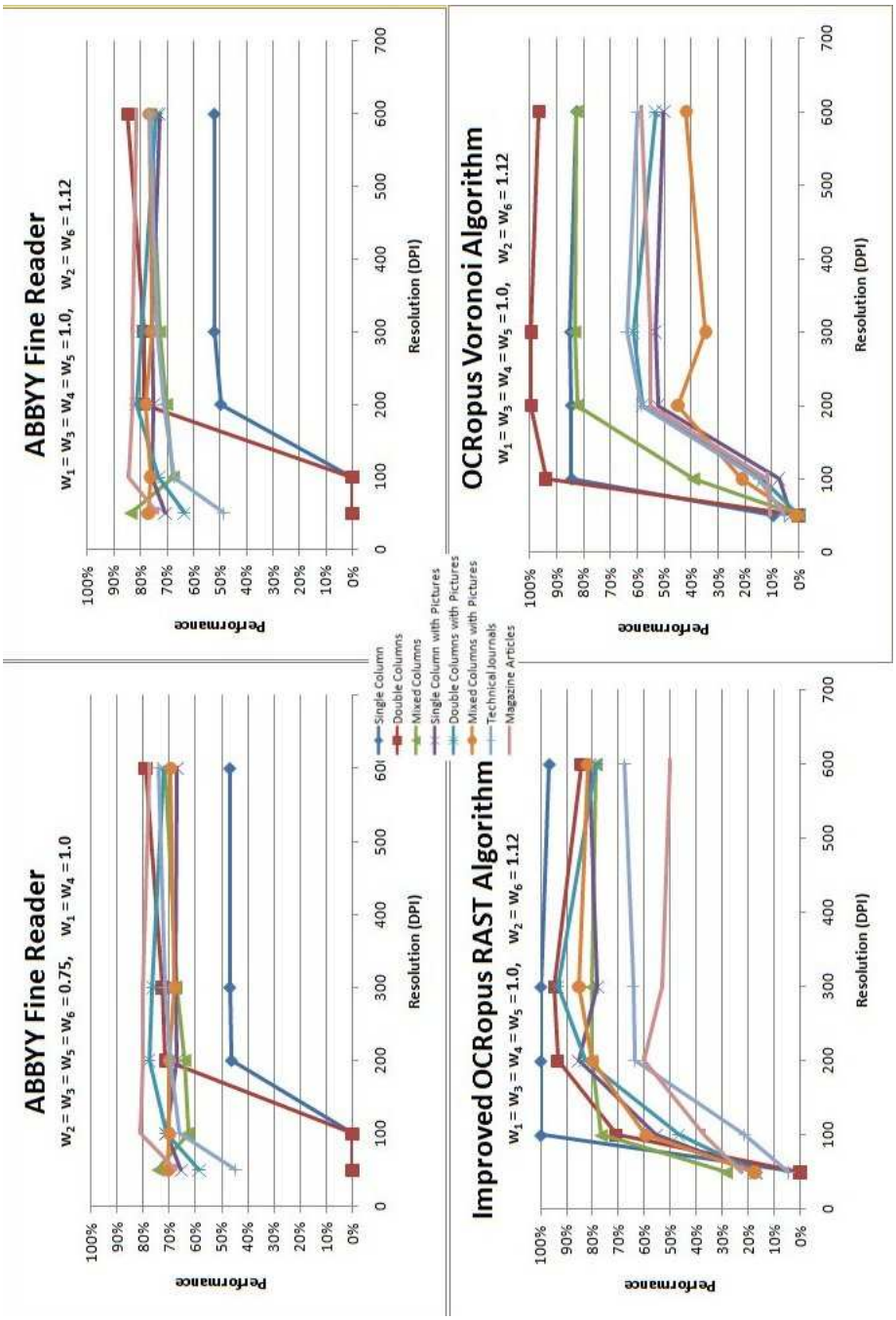


Figure 4.12: Performance of ABBYY's Fine Reader Engine 9.0 (bottom), the extended Voronoi algorithm (middle) and the improved (top) RAST algorithm.

Figure 4.13: Example of Fine Reader segmentation. Note the overlapping image and text boxes.

CHAPTER 5

CONCLUSION

In the interest of digitizing historical documents at a low cost, open source layout analysis programs were researched in the literature and on the Internet. A package under development called OCRopus, which contains a hardware solution for obtaining the images (i.e., a digital camera assembly) and a software solution for processing them, was deemed the most advanced available. In its current state, while the image processing capabilities were well developed, the page segmentation functionality was limited to text-only documents and was optimized for a resolution of 300 DPI. Therefore, the goal of this thesis was to improve its page segmentation performance, so that camera acquired images of historical documents with layouts similar to the Bavarian manuscripts of interest could be analyzed and converted to text.

After modifying the program to generate output in XML format, as well as writing a program to compare detected regions to ground truth, two page segmentation algorithms in OCRopus were evaluated. The first one was the default algorithm called RAST, which was designed for text-only documents. When tested on documents that contained non-text areas as well, it tended to classify regions within them as both text and non-text. Entities such as graphs and tables, on the other hand, tended to be divided up into both types of regions. The end result was that OCRopus output a series of errors along with whatever text it was able to recognize, thus rendering the

output illegible.

The second algorithm, based on the Voronoi method, was less mature than RAST in that it segmented the page into regions, but did not classify them. Therefore, this algorithm did not support OCR so could not even process text-only documents. In terms of segmentation ability, it worked fairly well, but tended to oversegment non-text areas as well as text typed in large fonts.

The RAST algorithm was modified in a number of different ways to improve its performance. First of all, it was discovered that the minimum length parameter used to define the text lines was not resolution independent so was changed to a multiple of the average character box width; however, the calculation of the average box width itself was found to be inaccurate, so an algorithm was developed to find its true value. This parameter was extracted from a peak of the histogram of the bounding boxes of the presumed characters. By smoothing the histogram iteratively until it assumed the targeted shape, the correct value could be extracted. Using this value, RAST was able to create text lines more accurately.

After the column dividers were found, the algorithm was expanded to merge and classify the regions correctly. The first two functions served the purpose of keeping track of pixels that had been lost previously to ensure that they are now classified as non-text. The next major function reclassifies text lines that overlap other text lines as non-text because text lines do not overlap. Then, the algorithm loops through a series of three functions that merge text lines that overlap non-text, non-text rectangles that overlap other non-text rectangles, and non-text rectangles in close proximity to each other until no new non-text rectangles are created. In this way, non-text areas such as figures are more accurately classified.

As for the Voronoi method, it did not have any classification algorithm, so one was

developed and implemented. It utilizes the same character box extraction method as RAST, then employs a smoothing algorithm to find the locations of text lines. After this is done, it examines the density of the boxes along the text lines. If enough lines have densities above a certain threshold, the region is considered text.

Following the classification of the regions, the oversegmentation of non-text regions was addressed. This was done in a recursive manner where a non-text region was selected and its neighboring non-text regions were relabeled with its zone number. After all of the non-text regions have been examined, the rectangles they form are considered. If any overlap, they are merged so that segmented regions do not overlap.

With this added functionality, the RAST and Voronoi algorithms are now capable of processing mixed-content layouts, making the digitization of standard format historical documents by low-budget organizations feasible. Once the improvements were implemented, they were tested on a set of test images. For the RAST algorithm, the performance of the hand-made documents with half-tone images improved an average of 40%, the technical document class 25%, and the magazine class 15% resulting in final overall accuracies of 90%, 65%, and 55%, respectively. While only the first six classes met the goal of the thesis, the other two consisted of more sophisticated content than would typically be included in an historic document, so is not considered as relevant.

The primary errors were caused by the oversegmentation of text areas due to unusually long or short text lines, the merging of short columns due to the constraints used for the definition of column dividers, and the merging of text and non-text regions due to non-Manhattan layouts, very narrow columns, and stylized text being classified as non-text.

The performance of the Voronoi algorithm was similar to RAST for the text-only

documents, but was lower for the documents containing half-tone images, graphs, and tables. The double column text-only class fared the best at 95% and the double and mixed column text-only classes at 85%. The rest of the classes, with the exception of the mixed columns with half-tone images, performed between 50% and 65%. Due to some anomalies in the files, the mixed column class was only segmented with an accuracy of 40%.

So, the Voronoi implementation only met the stated goal of the thesis for one of the classes; however, two other classes came close. As for the remainder, the factors impacting the performance of these documents included the layouts of the documents themselves, so that, in some cases, figures were so close to text that the kernel of the Voronoi algorithm merged them. Also, the algorithm tended to oversegment large text, breaking it up into separate zones. Additionally, italicized and bold text was classified as non-text. Since these errors stemmed from either the non-standard spacing of hand-made documents or the sophisticated layout of modern documents, it is likely that this method would perform better on the historical documents of interest.

Finally, the commercial package, Fine Reader, developed by ABBYY, was evaluated using the same set of test documents. With the exception of the single column class, Fine Reader performed more consistently for all classes and all resolutions than the OCRopus algorithms, with an accuracy of 70% to 85%. As for the single column class, its performance was lower because Fine Reader segmented the paragraphs; whereas, it was not segmented in the ground truth. While Fine Reader demonstrated a more consistent level of performance for all classes, it did not meet the 90% goal of the thesis either.

While the RAST and Voronoi algorithms performed well, there remain areas in

which they could be improved. Namely, the robustness of RAST could be increased so that it can process text lines of varying widths as well as short and/or narrow columns. The processing of stylized text and, for Voronoi, italicized and bolded text, could also be fixed. Also, the Voronoi algorithm could be enhanced by merging segmented titles and classifying italicized text properly. Finally, since the documents of interest were not available for this thesis, a true measurement of the performance of these algorithms could be obtained if images of the manuscripts were captured and processed.

In conclusion, the improved RAST algorithm compares well to a widely used commercial program in the case of documents that contain half-tone images rather than graphs and tables. The Voronoi algorithm did not perform as well as Fine Reader (by approximately 20%), but if the documents contain ample space between the figures and text, and there is no italicized or bolded text, it might perform adequately. Therefore, depending on the type of layout being digitized, either algorithm could potentially be employed.

REFERENCES

[1] "Printing Press." *Wikipedia*, http://en.wikipedia.org/wiki/Printing_press

[2] "Bavarian Traditional Clothing Culture Center and Archive." http://trachtenverband-bayern.de/hdbtt/

[3] "Image Scanner." *Wikipedia*, http://en.wikipedia.org/wiki/Image_scanner

[4] Breuel, T.M. "The OCRopus Open Source OCR System." *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 6815, 2008, pg. 68150F (15 pages). The code is hosted by Google at http://code.google.com/p/ocropus and documentation can be found at http://ocrocourse.iupr.com/.

[5] Silva, G.P. and Lins, R.D. "PhotoDoc: A Toolbox for Processing Document Images Acquired Using Portable Digital Cameras." *Camera-Based Document Analysis and Recognition (CBDAR2007)*, 2007, pp 107-113.

[6] "Tesseract." *Ray Smith*, http://code.google.com/p/tesseract-ocr/

[7] "Decapod." *Image Understanding Pattern Recognition (IUPR) Research Group at the DFK*, http://code.google.com/p/ocropus

[8] Droettboom, M., Fujinaga, I., MacMillan, K., Chouhury, G.S., DiLauro, T., Patton, M., Anderson, T. "Using the Gamera Framework for the Recognition of Cultural Heritage Materials." *Journal Conference of Digital Libraries (JDCL'02)*, July 13-17, 2002. For more information check http://gamera.informatik.hsnr.de/

[9] Shi, Z. and Govindaraju, V. "Dynamic Local Connectivity and Its Application to Page Segmentation." *Proceedings of the 1st ACM workshop on Hardcopy Document Processing (HDP-04)*, November 12, 2004, pp. 47-52.

[10] Nagy, G., Seth S. and Viswanathan, M. "A Prototype Document Image Analysis System for Technical Journals." *Computer*, vol. 25(7), July 1992, pp. 10-22.

[11] Wong, K.Y., Casey, R.G. and Wahl, R.M. "Document Analysis System." *IBM Journal of Research and Development*, vol 26(6), November 1982, pp. 647-656.

[12] O'Gorman, L. "The Document Spectrum for Page Layout Analysis." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 15(11), November 1993, pp. 1162-1173.

[13] Kise, K., Sato, A. and Iwata, M. "Segmentation of Page Images Using the Area Voronoi Diagram." *Computer Vision and Image Understanding*, vol. 70(3), June 1998, pp. 370-382.

[14] "Document Attribute Format Specification (DAFS)." `http://cool.conservation -us.org/bytopic/imaging/std/dafsdrft.html`

[15] Lee, C.H. and Kanungo, T. "The Architecture of TrueViz: A GroundTRUth/Metadata Editing and VIsualiZing ToolKit." *Pattern Recognition*, vol. 36(2003), pp. 811-825. The software itself is available at `http://www.kanungo.com/software/software.html#trueviz`

[16] Mao, S. and Kanungo, T.,"PSET: A Page Segmentation Evaluation Toolkit." *Proceedings of Document Analysis Systems*, Rio de Janeiro, Brazil, 2000.

[17] Mao, S. and Kanungo, T., "Software Architecture of PSET: A Page Segmentation Evaluation Toolkit." *International Journal on Document Analysis and Recognition (IJDAR)*, vol. 4(3), 2002, pp. 205-217.

[18] Shafait, F., Keysers, D. and Breuel, T.M. "Performance Evaluation and Benchmarking of Six -Page Segmentation Algorithms." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30(6), June 2008, pp. 941-954.

[19] Antonacopoulos, A., Gatos, B. and Bridson, D. "ICDAR2007 Page Segmentation Competition." *Proceedings of the 9th International Conference on Document Analysis and Recognition (ICDAR2007)*, Curtiba, Brazil, September 2007, IEEE Computer Society Press, pp. 1279-1283.

[20] Breuel, T.M. "A Practical, Globally Optimal Algorithm for Geometric Matching under Uncertainty." *Electronic Notes in Theoretical Computer Science*, vol. 46, 2001, pp. 1-15.

[21] Email exchange with Dr. Breuel, March 4, 2010.

[22] Breuel, T.M. "Two Geometric Algorithms for Layout Analysis." *Document Analysis Systems*, August 2002, pp. 188-199.

[23] "ABBYY Fine Reader". `http://www.frakturschrift.com/`

[24] Phillips, I.T. and Chhabra, A.K. "Empirical Performance Evaluation of Graphics Recognition Systems." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 21(9), September 1999, pp. 849-870.

[25] Solter, N.A. and Kleper, S.J. *Professional C++*, Wiley Publishing, Inc., 2005, pp 717-721.

[26] Andersen, T. and Zhang, W. "Features for Neural Net Based Region Identification of Newspaper Documents." *Proceedings of the Seventh International Conference on Document Analysis and Recognition (ICDAR'03)*, August 2003, pp 403-407.

[27] Alginahi, Y., Fekri, D. and Sid-Ahmed, M.A. "A Neural-Based Page Segmentation System." *Journal of Circuits, Systems and Computers*, vol. 14(1), 2005, pp 109-122.

[28] Keysers, D., Shafait, F. and Breuel, T.M. "Document Image Zone Classification - a simple high-performance approach." *Second International Conference on Computer Vision Theory and Applications*, Barcelona, Spain, March 2007, pp. 4451.

[29] Gonzales, R.C. and Woods, R.E. *Digital Image Processing*, Pearson Prentice Hall, 2008, pp. 635-639.

APPENDIX A

COMPARISON PROGRAM

A.1 README File

```
Zone Comparison Program

The file structure is:
    runZoneComp    executable
    ZoneComp.cpp   main program source file
    Rect.cpp       class source file
    Rect.hpp       class header file

Description:
    This program reads two xml files which list the page segmentation
zones of a document where the zones are categorized as "Text" or
"Non-text".  One of the input documents is the "ground truth" which
means it contains the true and accurate zone information of the
document; whereas the other file contains the zones as detected by
a page segmentation program.  ZoneComp then compares the two and
returns a metric of how well they match which is a measurement of
how well the page segmenter performed.

The metric is described in the following papers:
    A. Antonacopoulos, B. Gatos and D. Bridson, "ICDAR2007 Page
Segmentation Competition,"  Proceedings of the 9th International
Conference on Document Analysis and Recognition, Curitiba, Brazil,
September 2007, IEEE Computer Society Press, pp. 1279-183.

    I. Phillips and A. Chhabra, "Empirical Performance Evaluation of
Graphics Recognition Systems,"  IEEE Transaction on Pattern Analysis
and Machine Intelligence, Vol. 21, No. 9, pp. 849-870, Sept. 1999.

Note that if either XML file contains a document type tag like
```

```
<!DOCTYPE Page SYSTEM "Trueviz.dtd"> at the top of the page it needs
to be removed first.

To build the program type
    >make

The usage is
    >runZoneComp
        <-g name of ground truth xml file>
        <-d name of detected xml file>
        <[-r rejection threshold]>
        <[-a acceptance threshold]>
        <[-v for verbose]>

Sample program output in default mode is:
    Reporting results for the 1colpic300_2.xml
    Segmentation Metric =  1.00

Sample program output in verbose mode is:
    Reporting results for the 1colpic300_2.xml

    The number of one-to-one matches for the text region is 2.
    The number of one-to-one matches for the non-text region is 1.
    The number of d_one-to-many matches for the text region is 0.
    The number of d_one-to-many matches for the non-text region is 0.
    The number of g_many-to-one matches for the text region is 0.
    The number of g_many-to-one matches for the non-text region is 0.
    The number of g_one-to-many matches for the text region is 0.
    The number of g_one-to-many matches for the non-text region is 0.
    The number of d_many-to-one matches for the text region is 0.
    The number of d_many-to-one matches for the non-text region is 0.

    The text detection rate =  1.00
    The text recognition accuracy =  1.00
    The text region metric =  1.00

    The non-text detection rate =  1.00
    The non-text recognition accuracy =  1.00
    The non-text region metric =  1.00

    Segmentation Metric =  1.00
```

A.2 Code Documentation

A.2.1 Main Program Functions

This function prints the command line usage of the program:

void printUsage(const char* progName)

progName is the name of this program

This function initializes the XML platform:

void initializeXMLplatform()

This function checks the status of an input file
and returns the pointer its XML parser:

XercesDOMParser* checkFile(const char* fileName)

fileName is the name of the XML file to check

This function parses the xml documents:

void parseDoc(DOMNodeList* zoneList,
vector vector Rect zone, int numZones)

zoneList is the list of zones

zone is the list of rectangles

numZones is the number of zones

This function calculates the match scores of the documents:

void calculate_Match_Scores (vector vector Rect gtZone,
vector vector Rect dtZone,

vector vector vector float match_score,

vector vector vector int match_score_thres)

gtZone is the ground truth list of zones

dtZone is the etected list of zones

match_score the 2D vector (array) which holds the match scores

match_score_thres is the 2D vector (array)

which holds the thresholded match scores

This function calculates the G-Profile and the D-Profile:

void calculate_G_And_D_Profiles(

vector vector vector float match_score,

vector vector vector int match_score_thres,

vector vector int G_profile,

vector vector int D_profile)

match_score the 2D vector (array) which holds the match scores

match_score_thres the 2D vector (array) which holds

the thresholded match scores

G_profile the array which holds the Ground Truth profile

D_profile the array which holds the Detected profile

This function prints the G-Profile and the D-Profile:

void print_G_And_D_Profiles(

vector vector int G_profile,

vector vector int D_profile)

G_profile the array which holds the Ground Truth profile

D_profile the array which holds the Detected profile

This function computes the straight forward one-to-one matches:

void compute_one2one_Matches_Easy(

vector vector vector float match_score,

vector vector vector int match_score_thres,

vector vector int G_profile,

vector vector int D_profile)

match_score is the 2D vector (array) which holds the match scores

match_score_thres the 2D vector (array) which holds

the thresholded match scores

G_profile the array which holds the Ground Truth profile

D_profile the array which holds the Detected profile

This function computes the one-to-one matches by resolving the many-to-one detected conflicts for the first two cases

void compute_one2one_Matches_Resolving_

D_many2one_Conflicts_part1(

vector vector vector float match_score,

vector vector vector int match_score_thres,

vector vector int G_profile,

vector vector int D_profile)

match_score the 2D vector (array) which holds the match scores

match_score_thres the 2D vector (array) which holds

the thresholded match scores

G_profile the array which holds the Ground Truth profile

D_profile the array which holds the Detected profile

This function computes the one-to-one matches by resolving
the many-to-one detected conflicts for the third case:

void compute_one2one_Matches_Resolving_
D_many2one_Conflicts_part2(
vector vector vector float match_score,
vector vector vector int match_score_thres,
vector vector int G_profile,
vector vector int D_profile)

match_score the 2D vector (array) which holds the match scores

match_score_thres the 2D vector (array) which holds
the thresholded match scores

G_profile the array which holds the Ground Truth profile

D_profile the array which holds the Detected profile)

This function computes the one-to-one matches by resolving the one-to-many
detected conflicts where the D-Profile is greater than or equal to two

void compute_one2one_Matches_Resolving_
D_one2many_Conflicts_part1(
vector vector vector float match_score,
vector vector vector int match_score_thres,
vector vector int G_profile,
vector vector int D_profile)

match_score the 2D vector (array) which holds the match scores

match_score_thres the 2D vector (array) which holds

the thresholded match scores

G_profile the array which holds the Ground Truth profile

D_profile the array which holds the Detected profile

This function computes the one-to-one matches by resolving the one-to-many detected conflicts where the G-Profile is greater than or equal to two:

void compute_one2one_Matches_Resolving_

D_one2many_Conflicts_part2(

vector vector vector float match_score,

vector vector vector int match_score_thres,

vector vector int G_profile,

vector vector int D_profile)

match_score the 2D vector (array) which holds the match scores

match_score_thres the 2D vector (array) which holds

the thresholded match scores

G_profile the array which holds the Ground Truth profile

D_profile the array which holds the Detected profile

This function computes the partial Detected one-to-many matches and the partial Ground truth many-to-one matches:

void compute_D_one2many_Matches(

vector vector vector float match_score,

vector vector vector int match_score_thres,

vector vector int G_profile,

vector vector int D_profile)

match_score the 2D vector (array) which holds the match scores

match_score_thres the 2D vector (array) which holds

the thresholded match scores

G_profile the array which holds the Ground Truth profile

D_profile the array which holds the Detected profile

This function computes the partial Ground truth one-to-many matches and the partial Detected many-to-one matches

void compute_G_one2many_Matches(

vector vector vector float match_score,

vector vector vector int match_score_thres,

vector vector int G_profile,

vector vector int D_profile)

match_score the 2D vector (array) which holds the match scores

match_score_thres the 2D vector (array) which holds

the thresholded match scores

G_profile the array which holds the Ground Truth profile

D_profile the array which holds the Detected profile

This function calculates the detection rates:

void calculate_Performance(vector vector Rect gtZone,

vector vector Rect dtZone)

gtZone the ground truth list of zones

dtZone the detected list of zones

A.2.2 Rect Class Constructor and Functions

This constructor creates a zero area rectangle at (0,0) coordinates:

Rect::Rect()

This constructor creates a rectangle with the given coordinates:

Rect::Rect(int xmin, int xmax, int ymin, int ymax)

xmin the coordinate of the minimum x value

xmax the coordinate of the maximum x value

ymin the coordinate of the minimum y value

ymax the coordinate of the maximum y value

This method prints the coordinates of a rectangle:

void Rect::print()

This method returns the x-value of the left side (minimum x) of a rectangle:

int Rect::getLeft()

This method returns the x-value of the right side (maximum x) of a rectangle:

int Rect::getRight()

This method returns the y-value of the top (minimum y) of a rectangle:

int Rect::getTop()

This method returns the y-value of the bottom (maximum y) of a rectangle:

int Rect::getBottom()

This method sets the coordinates of a rectangle:

void Rect::setCoords(int xmin, int xmax, int ymin, int ymax)

xmin the coordinate of the minimum x value

xmax the coordinate of the maximum x value

ymin the coordinate of the minimum y value

ymax the coordinate of the maximum y value

This method sets the type or class of a rectangle:

void Rect::setType(int inType)

inType the class of the rectangle (i.e. Text or Non-text)

This method returns the type or class of a rectangle:

int Rect::getType()

This method calculates and returns the match score of two rectangles:

float Rect::getMatchScore(Rect otherRect)

otherRect the rectangle to compare to

APPENDIX B

XML OUTPUT

B.1 get-text-columns of ocr-detect-columns.cc

```
void get_text_columns(rectarray &textcolumns,
                      rectarray &textlines,
                      rectarray &gutters,
                      rectarray &graphics){   <--- graphics array now passed

    if(!textlines.length())  return;

    if(!gutters.length()){
        rectangle column = rectangle(textlines[0]);
        rectangle tempcolumn = column;
        for(int i=1; i<textlines.length(); i++){
            tempcolumn.include(textlines[i]);
            bool crosses_graphics = false;                new graphics code
            for(int j=0; j<graphics.length(); j++){                    |
                if (tempcolumn.fraction_covered_by(graphics[j])>0)     |
                    crosses_graphics = true;                           |
            }                                                          |
            if (crosses_graphics){                                     |
                textcolumns.push(column);                              |
                column = rectangle(textlines[i]);                      |
                tempcolumn = column;                                   V
            } else{
                column.include(textlines[i]);
            }
        }
        textcolumns.push(column);
        return;
    }
```

```
    rectangle column = rectangle(textlines[0]);
    rectangle tempcolumn =
    rectangle(textlines[0].dilated_by(-10,-2,-10,-2));
    for(int i=1; i<textlines.length(); i++){
        tempcolumn.include(textlines[i].dilated_by(-10,-2,-10,-2));
        bool intersects_gutter = false;
        bool gutter_penetrating_from_below = false;
        bool gutter_penetrating_from_above = false;
        for(int j=0; j<gutters.length(); j++){
            point top    = point(gutters[j].xcenter(),gutters[j].y1) ;
            point bottom = point(gutters[j].xcenter(),gutters[j].y0) ;
            if(tempcolumn.overlaps(gutters[j])){
                intersects_gutter = true;
                if(textlines[i].contains(top))
                    gutter_penetrating_from_below = true;
                if(textlines[i].contains(bottom))
                    gutter_penetrating_from_above = true;
                break;
            }
        }
        bool crosses_graphics = false;                more new graphics code
        for(int j=0; j<graphics.length(); j++){                       |
            if (tempcolumn.fraction_covered_by(graphics[j])>0)        |
                crosses_graphics = true;                              V
        }
        if (((intersects_gutter) || (crosses_graphics))
               && !gutter_penetrating_from_below){
            textcolumns.push(column);
            column = rectangle(textlines[i]);
            if(!gutter_penetrating_from_above)
                tempcolumn=rectangle(textlines[i].dilated_by(-10,-2,-10,-2));
            else
                tempcolumn=rectangle();
        } else{
            column.include(textlines[i]);
        }
    }
    textcolumns.push(column);   <-------- Push command added.
}
```

B.2 Functions of ocr-hps-output.cc

```
void hps_dump_preamble(FILE *output) {
    fprintf(output, "<!DOCTYPE html\n");
    fprintf(output, "   PUBLIC \"-//W3C//DTD XHTML 1.0 Transitional//EN\n");
    fprintf(output,
      "   http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\">\n");
}

void hps_dump_head(FILE *output) {
    fprintf(output, "<head>\n");
    fprintf(output,
      "<meta name=\"ocr-capabilities\" content=\"ocr_line ocr_page\" />\n");
    fprintf(output, "<meta name=\"ocr-langs\" content=\"en\" />\n");
    fprintf(output, "<meta name=\"ocr-scripts\" content=\"Latn\" />\n");
    fprintf(output, "<meta name=\"ocr-microformats\" content=\"\" />\n");
    fprintf(output,
      "<meta http-equiv=\"Content-Type\" content=\"text/html;charset=utf-8\" />");
    fprintf(output, "<title>OCR Output</title>\n");
    fprintf(output, "</head>\n");
}

void hps_dump_regions(FILE *output, rectarray &textArray,
                      rectarray &graphArray, int imageHeight)
{
    fprintf(output, "<Page>\n");

    for(int i=0; i<textArray.length(); i++)
    {
        int x0 = textArray[i].x0;
        int y0 = imageHeight - textArray[i].y1;
        int x1 = textArray[i].x1;
        int y1 = imageHeight - textArray[i].y0;

        fprintf(output, "<Zone>\n");

        fprintf(output, "<ZoneCorners>\n");
        fprintf(output, "<Vertex x=\"%d\" y=\"%d\">\n", x0, y0);
        fprintf(output, "</Vertex>\n");
        fprintf(output, "<Vertex x=\"%d\" y=\"%d\">\n", x1, y0);
        fprintf(output, "</Vertex>\n");
        fprintf(output, "<Vertex x=\"%d\" y=\"%d\">\n", x1, y1);
```

```
            fprintf(output, "</Vertex>\n");
            fprintf(output, "<Vertex x=\"%d\" y=\"%d\">\n", x0, y1);
            fprintf(output, "</Vertex>\n");
            fprintf(output, "</ZoneCorners>\n");

            fprintf(output, "<Classification>\n");
            fprintf(output, "<Category Value=\"Text\">\n");
            fprintf(output, "</Category>\n");
            fprintf(output, "</Classification>\n");

            fprintf(output, "</Zone>\n");
        }
        for(int i=0; i<graphArray.length(); i++)
        {
            int x0 = graphArray[i].x0;
            int y0 = imageHeight - graphArray[i].y1;
            int x1 = graphArray[i].x1;
            int y1 = imageHeight - graphArray[i].y0;

            fprintf(output, "<Zone>\n");

            fprintf(output, "<ZoneCorners>\n");
            fprintf(output, "<Vertex x=\"%d\" y=\"%d\">\n", x0, y0);
            fprintf(output, "</Vertex>\n");
            fprintf(output, "<Vertex x=\"%d\" y=\"%d\">\n", x1, y0);
            fprintf(output, "</Vertex>\n");
            fprintf(output, "<Vertex x=\"%d\" y=\"%d\">\n", x1, y1);
            fprintf(output, "</Vertex>\n");
            fprintf(output, "<Vertex x=\"%d\" y=\"%d\">\n", x0, y1);
            fprintf(output, "</Vertex>\n");
            fprintf(output, "</ZoneCorners>\n");

            fprintf(output, "<Classification>\n");
            fprintf(output, "<Category Value=\"Non-text\">\n");
            fprintf(output, "</Category>\n");
            fprintf(output, "</Classification>\n");

            fprintf(output, "</Zone>\n");
        }

        fprintf(output, "</Page>\n");
    }
```

APPENDIX C

RAST UPGRADE

C.1 Excerpts of ocr-layout/ocr-layout-rast.cc

```
void SegmentPageByRAST::segmentInternal(intarray &visualization,
                                        intarray &image,
                                        bytearray &in_not_inverted,
                                        bool need_visualization,
                                        rectarray &extra_obstacles) {

    const int zero   = 0;
    const int yellow = 0x00ffff00;
    bytearray in;
    copy(in, in_not_inverted);
    make_page_binary_and_black(in);

    // Do connected component analysis
    intarray charimage;
    copy(charimage,in);
    label_components(charimage,false);

    // Clean non-text and noisy boxes and get character statistics
    rectarray bboxes;
    bounding_boxes(bboxes,charimage);
    if(bboxes.length()==0){
        makelike(image,in);
        fill(image,0x00ffffff);
        return ;
    }

    autodel<CharStats> charstats(make_CharStats());
    charstats->getCharBoxes(bboxes);
    charstats->calcCharStats();
```

```
rectarray cboxes;
for(int i=0; i<charstats->char_boxes.length(); i++) {
    cboxes.push(charstats->char_boxes[i]);
}

// Compute Whitespace Cover
autodel<WhitespaceCover> whitespaces(
    make_WhitespaceCover(0,0,in.dim(0),in.dim(1)));
rectarray whitespaceboxes;
whitespaces->compute(whitespaceboxes,charstats->char_boxes);

// Find whitespace column separators (gutters)
autodel<ColSeparators> whitespace_obstacles(make_ColSeparators());
rectarray gutters, column_candidates;
whitespace_obstacles->
    findGutters(column_candidates, whitespaceboxes, *charstats);
whitespace_obstacles->filterOverlaps(gutters, column_candidates);

// Separate horizontal/vertical rulings from graphics
rectarray graphics;
rectarray hor_rulings;
rectarray vert_rulings;
autodel<ExtractRulings> rulings(make_ExtractRulings());
rulings->analyzeObstacles(hor_rulings,vert_rulings,graphics,
                          extra_obstacles,charstats->boxHeight);
rulings->analyzeObstacles(hor_rulings,vert_rulings,graphics,
                          charstats->large_boxes,charstats->boxHeight);

// Add whitespace gutters and the user-supplied obstacles
// to a list of obstacles
rectarray textline_obstacles;
for(int i=0;i<gutters.length();i++)
    textline_obstacles.push(gutters[i]);
for(int i=0;i<extra_obstacles.length();i++)
    textline_obstacles.push(extra_obstacles[i]);
for(int i=0;i<vert_rulings.length();i++)
    textline_obstacles.push(vert_rulings[i]);

 // Extract textlines
narray<TextLine> textlines;
autodel<CTextlineRAST> ctextline(make_CTextlineRAST());
ctextline->min_q     = 2.0;
```

```
ctextline->min_count = 2;
ctextline->min_length= (int) 2*charstats->boxWidth;
ctextline->max_results = max_results;
ctextline->min_gap = 3*charstats->boxWidth;
ctextline->extract(textlines,textline_obstacles,graphics,charstats);

// Capture the connected components that were rejected as characters.
rectarray rejected_cboxes;
for (int i=0; i<cboxes.length(); i++) {
    bool overlap = false;
    for (int j=0; j<textlines.length(); j++) {
        rectangle textline_box = textlines[j].bbox;
        if (cboxes[i].fraction_covered_by(textline_box)>0)
            overlap = true;
    }
    if (!overlap) {
        rejected_cboxes.push(cboxes[i]);
    }
}

// Merge the rejects then place them into the graphics array.
rectarray char_graphics;
int textlineHeight = charstats->boxHeight;
int dilation = 0.25*textlineHeight;
bool merged = closeRects(rejected_cboxes, char_graphics, dilation, dilation);
while (merged) {
    rectarray mBoxes;
    merged = closeRects(char_graphics, mBoxes, dilation, dilation);
    if (merged)     { char_graphics = mBoxes; }
}
for (int i=0; i<char_graphics.length(); i++) {
    graphics.push(char_graphics[i]);
}

// Of the small connected components, select those which do not
// overlap any textlines for further processing.
rectarray small_graphics;
for (int i=0; i<charstats->small_boxes.length(); i++) {
    bool overlap = false;
    for (int j=0; j<textlines.length(); j++) {
        if (textlines[j].bbox.fraction_covered_by(charstats->small_boxes[i])>0)
            overlap = true;
    }
```

```
    if (!overlap)
        small_graphics.push(charstats->small_boxes[i]);
}

// Now select the very small graphics rectangles.
int min_area = 0.1*textlineHeight*textlineHeight;
for (int i=0; i<graphics.length(); i++) {
    if (graphics[i].area()<min_area) {
        bool overlap = false;
        for (int j=0; j<textlines.length(); j++) {
            if (textlines[j].bbox.fraction_covered_by(graphics[i])>0)
                overlap = true;
        }
        if (!overlap)
            small_graphics.push(graphics[i]);
    }
}

// Merge all of the small graphics components which correspond
// to isolated pixels or gray areas in images then add them to
// the graphics array.
rectarray small_boxes;
dilation = 0.25*textlineHeight;
merged = closeRects(small_graphics, small_boxes, dilation, dilation);
while (merged) {
    rectarray mGraphics;
    merged = closeRects(small_boxes, mGraphics, dilation, dilation);
    if (merged)     { small_boxes = mGraphics; }
}
for (int i=0; i<small_boxes.length(); i++) {
 rectangle box = small_boxes[i];
    if ((box.width() < textlineHeight) && (box.height() > 10 * textlineHeight))
        continue;
    graphics.push(box);
}

// Merge overlapping text line boxes and insert them into the graphics array.
bool mergedArrays = true;
while (mergedArrays) {
    narray<TextLine> onlyTextlines;
    mergedArrays = mergeText(textlines, onlyTextlines, graphics);
    if (mergedArrays)   { textlines = onlyTextlines; }
}
```

```
// Clean up the graphics array by removing any little rectangles
// that might have been created while processing the gray areas.
min_area = textlineHeight*textlineHeight;
rectarray filtered_graphics2;
for (int i=0; i<graphics.length(); i++) {
    if (graphics[i].area()>min_area)
        filtered_graphics2.push(graphics[i]);
}
graphics = filtered_graphics2;

// Move textlines that overlap graphics to the graphics array,
// merge overlapping graphics boxes into megagraphics boxes
// then merge nearby graphics boxes.
// Continue doing this until no textlines overlap graphics
bool updated = true;
dilation = 1.4*textlineHeight;
    // 1.6 merges graph axis titles, but also figures
    // whereas 1.4 doesn't merge figures.
while (updated) {
    narray<TextLine> onlyTextlines;
    updated = mergeTextAndGraphics(
                  textlines, onlyTextlines, graphics, dilation);
    if (updated)    { textlines = onlyTextlines; }
}

// Sort textlines in reading order
autodel<ReadingOrderByTopologicalSort>
reading_order(make_ReadingOrderByTopologicalSort());
reading_order->sortTextlines(
           textlines,gutters,hor_rulings,vert_rulings,*charstats);

rectarray textcolumns;
rectarray paragraphs;
rectarray textline_boxes;
for(int i=0, l=textlines.length(); i<l; i++)
    textline_boxes.push(textlines[i].bbox);

// Group textlines into text columns
// Since vertical rulings have the same role as whitespace gutters,
// add them to vertical separators list as long as they are true gutters.
rectarray vert_separators;
for(int i=0,l=vert_rulings.length(); i<l; i++){
```

```
        vert_separators.push(vert_rulings[i]);
    }
    for(int i=0,l=gutters.length(); i<l; i++){
        bool overlap = false;
        for(int j=0; j<graphics.length(); j++) {
            if (gutters[i].fraction_covered_by(graphics[j])>0)
                overlap = true;
        }
        if (!overlap)
            vert_separators.push(gutters[i]);
    }

    get_text_columns(textcolumns, textline_boxes, vert_separators, graphics);

    FILE *output = stdout;
    //hps_dump_preamble(output);
    //hps_dump_head(output);
    hps_dump_regions(output, textcolumns, graphics, in_not_inverted.dim(1));
```

C.2 Excerpts of ocr-layout/ocr-char-stats.cc

```
/**
* @brief This function finds the major peaks of a histogram which
*  have two consecutive lower and higher points to each side.
* @param locations the array in which to place the peak locations
* @param a the histogram
* @param minsize the locatin on the histogram to start examining
* @param maxsize the locatin on the histogram to stop examining
* @param sigma the amount of smoothing to apply
*/
static void major_peaks(intarray &locations, floatarray &a,
int minsize, int maxsize, float sigma)
{
    locations.clear();
    floatarray v;
    copy(v, a);
    if (sigma>0)
        gauss1d(v, sigma);
    int start = max(2, minsize);
    int stop = min(v.length()-3, maxsize);
```

```
    float maxValue = 0;
    for (int i=start; i<stop; i++) {
    if (v[i] < 1) { v[i] = 0; }
        if (((v[i]>v[i-1]) && (v[i-1]>v[i-2])) &&
            ((v[i]>v[i+1]) && (v[i+1]>v[i+2])) &&
             (v[i] > 0.05 * maxValue))
        {
            locations.push(i);
            if (locations.length() == 1)
                maxValue = v[i];
}
    }
}

/**
* @brief This function determines the value of the rightmost peak of
*  a histogram by iteratively smoothing it until no more than
*  the given number of peaks remain.
* @param hist the histogram
* @param peakNumber the desired number of peaks
*/
static int get_hist_peak(floatarray &hist, int peakNumber)
{
int start = 2;
int stop = hist.length();
int numPeaks = 0;
int smooth = 0;
int peak = 0;
bool needsSmoothing = true;

while ((needsSmoothing) && (smooth < 15)) {
intarray modes;
major_peaks(modes, hist, start, stop, smooth);

if ((numPeaks = modes.length()) == peakNumber)
{ // return the value of the peak of choice
peak = modes(peakNumber-1);
needsSmoothing = false;
}
else if (numPeaks == 0)
{ // no peaks were found so more smoothing is needed
needsSmoothing = true;
}
```

```
else if (numPeaks < peakNumber)
{ // too few peaks were found, take the rightmost
peak = modes(numPeaks-1);
needsSmoothing = false;
}
smooth++;
}
if (smooth == 15) { peak = 0; }
return peak;
}
```

C.3 Excerpts of ocr-layout/ocr-layout-manip.cc

```
/**
* @brief This function merges text lines then puts them into the graphics array.
* @param textArray the input text line array
* @param newTextArray the output text line array
* @param graphicsArray the graphics array
* @return true if text lines were moved to the graphics array
*/
bool mergeText(narray<TextLine> &textArray,
               narray<TextLine> &newTextArray,
               rectarray &graphicsArray)
{
int i, j, numBoxes = textArray.length();
    int mergeStatus[numBoxes];
    bool arraysMerged=false;
    for(i=0; i<numBoxes; i++) { mergeStatus[i] = 0; }
    for(i=0; i<numBoxes-1; i++) {
        j=i+1;
        if (mergeStatus[i]==0) {
            while ((j<numBoxes) && (mergeStatus[j]==0)) {
                rectangle tlBbox_i = textArray[i].bbox;
                rectangle tlBbox_j = textArray[j].bbox;
                if (tlBbox_i.fraction_covered_by(tlBbox_j)>0) {
                    float tlBbox_i_height = tlBbox_i.height();
                    float tlBbox_j_height = tlBbox_j.height();
                    float diff = abs(tlBbox_i_height - tlBbox_j_height);
                    float sum = tlBbox_i_height + tlBbox_j_height;
                    float ratio;
                    if (tlBbox_i_height < tlBbox_j_height)
```

```
                        ratio = tlBbox_i_height / tlBbox_j_height;
                    else
                        ratio = tlBbox_j_height / tlBbox_i_height;

                    if (ratio < 0.7)
                    {   // we've got a large and small rectangle
                        rectangle combinedRect = tlBbox_i.inclusion(tlBbox_j);
                        graphicsArray.push(combinedRect);
                        mergeStatus[i] = 1;
                        mergeStatus[j] = 1;
                        arraysMerged = true;
                    } else
                    {   // we probably have two text lines, check the overlap
                        if (diff/sum > 0.15) {
                            rectangle combinedRect = tlBbox_i.inclusion(tlBbox_j);
                            graphicsArray.push(combinedRect);
                            mergeStatus[i] = 1;
                            mergeStatus[j] = 1;
                            arraysMerged = true;
                        }
                    }
                }
                j++;
            }
        }
    }
    for(i=0; i<numBoxes; i++)
        if (mergeStatus[i]==0)  { newTextArray.push(textArray[i]); }
    return arraysMerged;
}

/**
* @brief This function merges overlapping rectangles.
* @param currentArray the input array
* @param newArray the output array
* @return true if rectangles were merged
*/
bool mergeRects(rectarray &currentArray, rectarray &newArray)
{
    int i, j, numBoxes = currentArray.length();
    int mergeStatus[numBoxes];
    bool lastMerged=false, arraysMerged=false;
    if (numBoxes==0)    { return false; }
```

```
    for(i=0; i<numBoxes; i++) { mergeStatus[i] = 0; }
    for(i=0; i<numBoxes-1; i++) {
        j=i+1;
        if (mergeStatus[i]==0) {
            while ((j<numBoxes) && (mergeStatus[j]==0)) {
                if (currentArray[i].fraction_covered_by(currentArray[j])>0) {
                    rectangle combinedRect =
                           currentArray[i].inclusion(currentArray[j]);
                    newArray.push(combinedRect);
                    mergeStatus[i] = 1;
                    mergeStatus[j] = 1;
                    arraysMerged = true;
                    if (j == numBoxes-1)    { lastMerged = true; }
                }
                j++;
            }
        }
        if (mergeStatus[i]==0)     { newArray.push(currentArray[i]); }
    }
    if (!lastMerged)    { newArray.push(currentArray[i]); }
    return arraysMerged;
}

/**
* @brief This function closes rectangles by dilating, merging then eroding them.
* @param currentArray the input array
* @param newArray the output array
* @param x_dilation the horizontal dilation
* @param y_dilation the vertical dilation
* @return true if rectangles were merged
*/
bool closeRects(rectarray &currentArray,
                rectarray &newArray,
                int x_dilation,
                int y_dilation)
{
    int i, j, numBoxes = currentArray.length();
    rectarray dilatedArray;
    int mergeStatus[numBoxes];
    bool lastMerged=false, arraysMerged=false;

    if (numBoxes==0)    { return false; }
```

```
    for(i=0; i<numBoxes; i++)
    {
        dilatedArray.push(currentArray[i].dilated_by(
               x_dilation, y_dilation, x_dilation, y_dilation));
        mergeStatus[i] = 0;
    }
    for(i=0; i<numBoxes-1; i++) {
        j=i+1;
        if (mergeStatus[i]==0) {
            while ((j<numBoxes) && (mergeStatus[j]==0)) {
                if (dilatedArray[i].fraction_covered_by(dilatedArray[j])>0) {
                    rectangle combinedRect =
                           dilatedArray[i].inclusion(dilatedArray[j]);
                    newArray.push(combinedRect.dilated_by(
                          -x_dilation, -y_dilation, -x_dilation, -y_dilation));
                    mergeStatus[i] = 1;
                    mergeStatus[j] = 1;
                    arraysMerged = true;
                    if (j == numBoxes-1)    { lastMerged = true; }
                }
                j++;
            }
        }
        if (mergeStatus[i]==0)   { newArray.push(currentArray[i]); }
    }
    if (!lastMerged)            { newArray.push(currentArray[i]); }
    return arraysMerged;
}

/**
* @brief This function closes rectangles by dilating, merging then eroding them.
* @param textArray the input text line array
* @param newTextArray the output text line array
* @param dilation the graphics dilation
* @return true if rectangles were added to the new array
*/
bool mergeTextAndGraphics(narray<TextLine> &currentTextArray,
                          narray<TextLine> &newTextArray,
                          rectarray &graphicsArray,
                          int dilation)
{
    bool update = false;
```

```
    // Move textlines that overlap graphics to the graphics array
    bool overlap;
    for(int i=0; i<currentTextArray.length(); i++) {
        overlap = false;
        rectangle tlBbox = currentTextArray[i].bbox;
        for(int j=0; j<graphicsArray.length(); j++) {
            if (tlBbox.fraction_covered_by(graphicsArray[j])>0)
                overlap = true;
        }
        if (overlap) {
            graphicsArray.push(tlBbox);
            update = true;
        }
        else
            newTextArray.push(currentTextArray[i]);
    }

    // Merge overlapping graphics boxes into megagraphics boxes
    bool mergedArrays = true;
    while (mergedArrays) {
        rectarray mGraphics;
        mergedArrays = mergeRects(graphicsArray, mGraphics);
        if (mergedArrays) {
            graphicsArray = mGraphics;
            update = true;
        }
    }

    // Merge nearby graphics boxes
    mergedArrays = true;
    while (mergedArrays) {
        rectarray mGraphics;
        mergedArrays = closeRects(graphicsArray, mGraphics, dilation, dilation);
        if (mergedArrays) {
            graphicsArray = mGraphics;
            update = true;
        }
    }
    return update;
}
```

APPENDIX D

VORONOI UPGRADE

D.1 Excerpts of ocr-voronoi/ocr-voronoi-ocropus.cc

```
// Color the Voronoi zones and lines
intarray voronoi_zones, voronoi_lines;
makelike(voronoi_zones, voronoi_diagram_image);
makelike(voronoi_lines, voronoi_diagram_image);
for (int i=0; i<voronoi_diagram_image.length1d(); i++){
    if (voronoi_diagram_image.at1d(i)==0x00ffffff ||
        voronoi_diagram_image.at1d(i)==0) {
        // black or white pixels
        voronoi_zones.at1d(i) = 1;
        voronoi_lines.at1d(i) = 0;
    }
    else {
        // blue pixels corresponding to the lines
        voronoi_zones.at1d(i) = 0;
        voronoi_lines.at1d(i) = 1;
    }
}

// Define the regions by extracting the connected components
// created above and color each differently
// The first zone is the lines.
int numZones = label_components(voronoi_zones,false);

// Now get the bounding boxes of the connected components
bytearray in;
copy(in, in_not_binary);
make_page_binary_and_black(in);

intarray charimage;
```

```
copy(charimage,in);
label_components(charimage,false);

rectarray bboxes;
bounding_boxes(bboxes,charimage);

autodel<CharStats> charstats(make_CharStats());
charstats->getCharBoxes(bboxes);
charstats->calcCharStats();

int numCharBoxes = charstats->char_boxes.length();
rectarray cBoxes;
for (int i=0; i<numCharBoxes; i++)
    cBoxes.push(charstats->char_boxes[i]);

int overlap[numCharBoxes];
for (int i=0; i<numCharBoxes; i++)
     overlap[i] = 0;

for (int i=0; i<numCharBoxes; i++) {
    for (int j=i+1; j<numCharBoxes; j++) {
        if (cBoxes[i].overlaps(cBoxes[j])) {
           overlap[i] = 1;
           overlap[j] = 1;
        }
    }
}

// Find the extreme points of the character boxes in each zone.
vector<int> wrap_around;
int xminText[numZones], xmaxText[numZones],
    yminText[numZones], ymaxText[numZones];
for (int z=0; z<numZones; z++) {
    xminText[z] = pageWidth;
    xmaxText[z] = 0;
    yminText[z] = pageHeight;
    ymaxText[z] = 0;
    wrap_around.push_back(0);
}

// Can only create one zone character box array at a time
// because of memory limitations.
rectarray printZone;
```

```
rectangle textRect[numZones];
bool textZone[numZones];
for (int z=1; z<numZones; z++) {
    rectarray zoneBoxes;
    for (int j=0; j<numCharBoxes; j++) {
        if (overlap[j] == 0) {
           rectangle box = cBoxes[j];
           int xmin = box.x0;
           int ymin = box.y0;
           if (z == voronoi_zones(xmin, ymin)) {
               zoneBoxes.push(box);
               int xmax = box.x1;
               int ymax = box.y1;
               if (xmin < xminText[z]) { xminText[z] = xmin; }
               if (xmax > xmaxText[z]) { xmaxText[z] = xmax; }
               if (ymin < yminText[z]) { yminText[z] = ymin; }
               if (ymax > ymaxText[z]) { ymaxText[z] = ymax; }
            }
          }
      }
      if (zoneBoxes.length() > 0) {
         if (xminText[z] < xmaxText[z])
             textRect[z] = rectangle(xminText[z], yminText[z],
                                     xmaxText[z], ymaxText[z]);
         else
             textRect[z] = rectangle();

         textZone[z] = is_text_block(zoneBoxes, wrap_around, z);
       }
       else
           textZone[z] = false;
}

// Create an array of the pixels of each zone.
vector<vector<Pixel> > vZone;
for (int z=0; z<numZones; z++)
    vZone.push_back( vector<Pixel>() );

for (int x=0,w=pageWidth;x<w;x++){
    for (int y=0,h=pageHeight;y<h;y++){
        Pixel pixel;
        pixel.x = x;
        pixel.y = y;
```

```
        vZone[voronoi_zones(x,y)].push_back(pixel);
    }
}

// Dilate the lines dividing the zones to get the perimeters.
intarray dilated_lines;
makelike(dilated_lines, voronoi_lines);
for (int x=0,w=pageWidth;x<w;x++)
    for (int y=0,h=pageHeight;y<h;y++)
        dilated_lines(x,y) = 0;

for (int x=1,w=pageWidth-1;x<w;x++){
    for (int y=1,h=pageHeight-1;y<h;y++){
         if (voronoi_lines(x,y) > 0){
             dilated_lines(x,y+1) = 1;
             dilated_lines(x+1,y+1) = 1;
             dilated_lines(x+1,y) = 1;
             dilated_lines(x+1,y-1) = 1;
             dilated_lines(x,y-1) = 1;
             dilated_lines(x-1,y-1) = 1;
             dilated_lines(x-1,y) = 1;
             dilated_lines(x-1,y+1) = 1;
          }
     }
}

// Create an array of the zone perimeters.
vector<vector<Pixel> > vPeri;
for (int z=0; z<numZones; z++)
    vPeri.push_back( vector<Pixel>() );

for (int z=1; z<numZones; z++) {
    for (int p=0; p<vZone[z].size(); p++) {
        int x = vZone[z][p].x;
        int y = vZone[z][p].y;
        if (dilated_lines(x,y) == 1)
           vPeri[z].push_back(vZone[z][p]);
    }
}

// Find the extreme points of each zone.
Pixel xminZone[numZones], xmaxZone[numZones],
      yminZone[numZones], ymaxZone[numZones];
```

```
for (int z=0; z<numZones; z++) {
    xminZone[z].x = pageWidth;
    xmaxZone[z].x = 0;
    yminZone[z].y = pageHeight;
    ymaxZone[z].y = 0;
}
for (int z=1; z<numZones; z++) {
    for (int p=0; p<vPeri[z].size(); p++) {
        int x = vPeri[z][p].x;
        int y = vPeri[z][p].y;
        if (x < xminZone[z].x) { xminZone[z].x = x; xminZone[z].y = y; }
        if (x > xmaxZone[z].x) { xmaxZone[z].x = x; xmaxZone[z].y = y; }
        if (y < yminZone[z].y) { yminZone[z].y = y; yminZone[z].x = x; }
        if (y > ymaxZone[z].y) { ymaxZone[z].y = y; ymaxZone[z].x = x; }
     }
}

// Create an array to label zones text or not.
vector<int> converted;
vector<int> imageMap;
for (int z=0; z<numZones; z++){
    converted.push_back(0);
    imageMap.push_back(z);
}

// Put the image-like zones into an array according to size.
vector<int> zoneBySize;
int firstIndex = 1; // start with one since zero is the lines
bool firstAdded = false;
while (!firstAdded) {
     if (!textZone[firstIndex]) {
         zoneBySize.push_back(firstIndex);
         firstAdded = true;
     }
     else
         firstIndex++;
}

for (int z=firstIndex+1; z<numZones; z++)
{
    if (!textZone[z])
    {
        int j = 0;
```

```
        int numPixels = vZone[z].size();
        bool foundPlace = false;
        while ((!foundPlace) && (j < zoneBySize.size()))
             if (numPixels < vZone[zoneBySize[j]].size())
                 foundPlace = true;
             else
                 j++;
        if (j == 0)
            zoneBySize.insert(zoneBySize.begin(), z);
        else if (j == zoneBySize.size())
            zoneBySize.push_back(z);
        else
            zoneBySize.insert(zoneBySize.begin()+j, z);
    }
}

// Create a vector of graphics
vector<int> graphics;   // the int value will correspond to the zone number

// Consider the smallest zone on the list.  If it's really small and
// is considered image-like, call it an image,
// find its neighbors, convert them to my zone, and so on.
for (int zi=0; zi<zoneBySize.size(); zi++)
{
     int currentNum = zoneBySize[zi];

     if ((!textZone[currentNum]) && (converted[currentNum] == 0))
     {
         bool contain = false;
         for (int g=0; g<graphics.size(); g++)
             if (graphics[g] == currentNum)  { contain = true; }
         if (!contain)   { graphics.push_back(currentNum); }
            converted[currentNum] = 1;

         int zoneCount[numZones];
         for (int z=1; z<numZones; z++)  { zoneCount[z] = 0; }
         getBorderingZones(voronoi_zones, vPeri, currentNum,
                           zoneCount, pageWidth, pageHeight);

         for (int z=1; z<numZones; z++) {
             if ((zoneCount[z] > 0) && (!textZone[z]) && (converted[z] == 0))
             {
                 convertZone(voronoi_zones, vZone, vPeri, converted, imageMap,
```

```
                                numZones, z, currentNum,
                                pageWidth, pageHeight, textZone, graphics);
                }
            }
        }
}

// Now start cleaning up the rectangular zones.
// Create an array of text segments.
int textSegIndex = 0;
int textMap[numZones];
rectarray textSegments0;
for (int z=1; z<numZones; z++) {
    if (converted[z] == 0) {
       textSegments0.push(textRect[z]);
       textMap[textSegIndex++] = z;
    }
}

// Create an array of image segments.
// One at a time like the text zones because of memory limitations.
int numImages = graphics.size();
int xminImage[numImages], xmaxImage[numImages],
    yminImage[numImages], ymaxImage[numImages];
for (int i=0; i<numImages; i++) {
    xminImage[i] = pageWidth;
    xmaxImage[i] = 0;
    yminImage[i] = pageHeight;
    ymaxImage[i] = 0;
}
rectarray imageSegments0;
for (int i=0; i<numImages; i++) {
    int imgZone = graphics[i];
    for (int x=0,w=pageWidth;x<w;x++){
        for (int y=0,h=pageHeight;y<h;y++){
            if (voronoi_zones(x,y) == imgZone) {
                if (voronoi_diagram_image(x,y) == 0) {
                    if (x < xminImage[i]) { xminImage[i] = x; }
                    if (x > xmaxImage[i]) { xmaxImage[i] = x; }
                    if (y < yminImage[i]) { yminImage[i] = y; }
                    if (y > ymaxImage[i]) { ymaxImage[i] = y; }
                }
            }
```

```
        }
    }
    imageSegments0.push(rectangle(xminImage[i], yminImage[i],
                                  xmaxImage[i], ymaxImage[i]));
}

// Identify text segments that are completely covered by image segments
// and delete them.
int notText0[textSegments0.length()];
for (int t=0; t<textSegments0.length(); t++)
    notText0[t] = 0;

for (int i=0; i<imageSegments0.length(); i++) {
    for (int t=0; t<textSegments0.length(); t++) {
        if (imageSegments0[i].includes(textSegments0[t]))
           notText0[t] = 1;
    }
}
rectarray textSegments1;
for (int t=0; t<textSegments0.length(); t++) {
    if (notText0[t] == 0)
        textSegments1.push(textSegments0[t]);
}

// Identify image segments that are completely covered by text zones,
// delete them and convert the text segment to an image.
int notImage0[imageSegments0.length()];
for (int i=0; i<imageSegments0.length(); i++)
    notImage0[i] = 0;

int notText1[textSegments1.length()];
for (int t=0; t<textSegments1.length(); t++)
    notText1[t] = 0;

rectarray tempTextZone;
for (int t=0; t<textSegments1.length(); t++) {
    int z = voronoi_zones(textSegments1[t].x0, textSegments1[t].y0);
    xminText[z] = pageWidth;
    xmaxText[z] = 0;
    yminText[z] = pageHeight;
    ymaxText[z] = 0;
    for (int p=0; p<vZone[z].size(); p++) {
        Pixel pixel = vZone[z][p];
```

```
        int x = pixel.x;
        int y = pixel.y;
        if (voronoi_diagram_image(x,y) == 0) {
           if (x < xminText[z]) { xminText[z] = x; }
           if (x > xmaxText[z]) { xmaxText[z] = x; }
           if (y < yminText[z]) { yminText[z] = y; }
           if (y > ymaxText[z]) { ymaxText[z] = y; }
         }
     }
     tempTextZone.push(rectangle(xminText[z], yminText[z],
                                 xmaxText[z], ymaxText[z]));
     for (int i=0; i<imageSegments0.length(); i++) {
         if (tempTextZone[t].includes(imageSegments0[i])) {
             notImage0[i] = 1;
             notText1[t] = 1;
         }
     }
}

rectarray imageSegments1;
for (int i=0; i<imageSegments0.length(); i++)
    if (notImage0[i] == 0)
       imageSegments1.push(imageSegments0[i]);

for (int t=0; t<textSegments1.length(); t++) {
    if (notText1[t] == 1) {
       imageSegments1.push(tempTextZone[t]);
    }
}
rectarray textSegments2;
for (int t=0; t<textSegments1.length(); t++)
    if (notText1[t] == 0)
        textSegments2.push(textSegments1[t]);


// Break image segments that cross column dividers and
// text segments that wrap around images.
rectarray newTextSegs, newImageSegs;
int brokenTextSegs[textSegments2.length()],
    brokenImageSegs[imageSegments1.length()];
for (int t=0; t<textSegments2.length(); t++)
     brokenTextSegs[t] = 0;
for (int i=0; i<imageSegments1.length(); i++)
```

```
     brokenImageSegs[i] = 0;

for (int t=0; t<textSegments2.length(); t++) {
    for (int i=0; i<imageSegments1.length(); i++) {
        if (textSegments2[t].overlaps(imageSegments1[i])) {
           if (wrap_around[textMap[t]] == 0) {
               breakImage(imageSegments1, textSegments2, newImageSegs, i, t);
               brokenImageSegs[i] = 1;
            } else {
               breakText(textSegments2, imageSegments1, newTextSegs, t, i);
               brokenTextSegs[t] = 1;
            }
         }
     }
}

int textOverlaps[newTextSegs.length()];
for (int i=0; i<newTextSegs.length(); i++)
     textOverlaps[i] = 0;

for (int i=0; i<newTextSegs.length(); i++) {
    for (int j=0; j<newTextSegs.length(); j++) {
         if ((i != j) && (newTextSegs[i].overlaps(newTextSegs[j]))) {
             if (newTextSegs[i].area() > newTextSegs[j].area())
                 textOverlaps[i] = 1;
             else
                 textOverlaps[j] = 1;
          }
     }
}

int imageOverlaps[newImageSegs.length()];
for (int i=0; i<newImageSegs.length(); i++)
    imageOverlaps[i] = 0;

for (int i=0; i<newImageSegs.length(); i++) {
    for (int j=0; j<newImageSegs.length(); j++) {
        if ((i != j) && (newImageSegs[i].overlaps(newImageSegs[j]))) {
            if (newImageSegs[i].area() > newImageSegs[j].area())
                imageOverlaps[i] = 1;
            else
                imageOverlaps[j] = 1;
         }
```

```
    }
}

rectarray finalTextSegments;
for (int t=0; t<textSegments2.length(); t++)
    if (brokenTextSegs[t] == 0)
        finalTextSegments.push(textSegments2[t]);
    for (int t=0; t<newTextSegs.length(); t++)
        if (textOverlaps[t] == 0)
            finalTextSegments.push(newTextSegs[t]);

rectarray finalImageSegments;
for (int i=0; i<imageSegments1.length(); i++)
    if (brokenImageSegs[i] == 0)
        finalImageSegments.push(imageSegments1[i]);
for (int i=0; i<newImageSegs.length(); i++)
    if (imageOverlaps[i] == 0)
        finalImageSegments.push(newImageSegs[i]);
```

D.2 Excerpts of ocr-voronoi/ocr-zone-manip.cc

```
void getBorderingZones(intarray &voronoi_zones, vector<vector<Pixel> >& vPeri,
                           int z, int* zoneCount, int pageWidth, int pageHeight)
{
   // Now tally the number of zones along the border.
   // For each pixel, venture in all four directions until the line is crossed.
   for (int p=0; p<vPeri[z].size(); p++)
   {
       int zoneEast=0, zoneWest=0, zoneNorth=0, zoneSouth=0;
       int stepEast=0, stepWest=0, stepNorth=0, stepSouth=0;

       int eastX = vPeri[z][p].x+1;
       int eastY = vPeri[z][p].y;
       while ((eastX < pageWidth) &&
              ((zoneEast = voronoi_zones(eastX, eastY)) == 0) &&
              (stepEast < 20)) {
             stepEast++; eastX++;
       }

       int westX = vPeri[z][p].x-1;
```

```
        int westY = vPeri[z][p].y;
        while ((westX >= 0) &&
              ((zoneWest = voronoi_zones(westX, westY)) == 0) &&
              (stepWest < 20))  {
            stepWest++; westX--;
        }

        int southX = vPeri[z][p].x;
        int southY = vPeri[z][p].y-1;
        while ((southY >= 0) &&
              ((zoneSouth = voronoi_zones(southX, southY)) == 0) &&
              (stepSouth < 20))  {
            stepSouth++; southY--;
        }

        int northX = vPeri[z][p].x;
        int northY = vPeri[z][p].y+1;
        while ((northY < pageHeight) &&
              ((zoneNorth = voronoi_zones(northX, northY)) == 0) &&
              (stepNorth < 20)) {
            stepNorth++; northY++;
        }

        // If the line was crossed in any of the directions
        // add that zone to the count.
        if ((stepEast > 0) && (stepEast < 20))
           zoneCount[zoneEast]++;
        else if ((stepWest > 0) && (stepWest < 20))
           zoneCount[zoneWest]++;
        else if ((stepSouth > 0) && (stepSouth < 20))
           zoneCount[zoneSouth]++;
        else if ((stepNorth > 0) && (stepNorth < 20))
           zoneCount[zoneNorth]++;

    }
}

void convertZone(intarray &voronoi_zones, vector<vector<Pixel> >& vZone,
                 vector<vector<Pixel> >& vPeri, vector<int>& converted,
                 vector<int>& mapped, int numZones, int thisZoneNum,
                 int newZoneNum, int pageWidth, int pageHeight,
                 bool* textZone, vector<int>& graphics)
{
```

```
   for (int p=0; p<vZone[thisZoneNum].size(); p++) {
       int pixelX = vZone[thisZoneNum][p].x;
       int pixelY = vZone[thisZoneNum][p].y;
       voronoi_zones(pixelX,pixelY) = newZoneNum;
   }
   converted[thisZoneNum] = 1;
   mapped[thisZoneNum] = newZoneNum;

   int zoneCount[numZones];
   for (int z=1; z<numZones; z++)  { zoneCount[z] = 0; }
   getBorderingZones(voronoi_zones, vPeri, thisZoneNum,
                     zoneCount, pageWidth, pageHeight);

   for (int z=1; z<numZones; z++)
   {
       if ((zoneCount[z] > 0) && (!textZone[z]) && (converted[z] ==0))
       {
           convertZone(voronoi_zones, vZone, vPeri, converted,
                       mapped, numZones, z, newZoneNum,
                       pageWidth, pageHeight, textZone, graphics);
       }
   }
}

void breakImage(rectarray &arrayToBreak, rectarray &breakerArray,
                rectarray &newArray, int arrayToBreak_index,
                int breakerArray_index)
{
rectangle rectToBreak = arrayToBreak[arrayToBreak_index];
rectangle breakerRect = breakerArray[breakerArray_index];
    rectangle overlap = rectToBreak.intersection(breakerRect);

    if ((overlap.x1 + overlap.x0)/2 > (rectToBreak.x1 + rectToBreak.x0)/2) {
       if ((overlap.y1 + overlap.y0)/2 > (rectToBreak.y1 + rectToBreak.y0)/2) {
          newArray.push(rectangle(rectToBreak.x0, rectToBreak.y0,
                                  overlap.x0-1, rectToBreak.y1));
          newArray.push(rectangle(overlap.x0, rectToBreak.y0,
                                  rectToBreak.x1, overlap.y0-1));
       }
       else {
          newArray.push(rectangle(rectToBreak.x0, rectToBreak.y0,
                                  overlap.x0-1, rectToBreak.y1));
          newArray.push(rectangle(overlap.x0, overlap.y1+1,
```

```
                                    rectToBreak.x1, rectToBreak.y1));
        }
    }
    else {
        if ((overlap.y1 + overlap.y0)/2 > (rectToBreak.y1 + rectToBreak.y0)/2) {
            newArray.push(rectangle(rectToBreak.x0, rectToBreak.y0,
                                    overlap.x1, overlap.y0-1));
            newArray.push(rectangle(overlap.x1+1, rectToBreak.y0,
                                    rectToBreak.x1, rectToBreak.y1));
        }
        else {
            newArray.push(rectangle(rectToBreak.x0, overlap.y1+1,
                                    overlap.x1, rectToBreak.y1));
            newArray.push(rectangle(overlap.x1+1, rectToBreak.y0,
                                    rectToBreak.x1, rectToBreak.y1));
        }
    }
}

void breakText(rectarray &arrayToBreak, rectarray &breakerArray,
   rectarray &newArray, int arrayToBreak_index, int breakerArray_index)
{
rectangle rectToBreak = arrayToBreak[arrayToBreak_index];
    rectangle breakerRect = breakerArray[breakerArray_index];
    rectangle overlap = rectToBreak.intersection(breakerRect);

    if ((overlap.x1 + overlap.x0)/2 > (rectToBreak.x1 + rectToBreak.x0)/2) {
        if (breakerRect.includes(rectToBreak.x1, rectToBreak.y1)) {
            newArray.push(rectangle(rectToBreak.x0, rectToBreak.y0,
                                    rectToBreak.x1, overlap.y0-1));
            newArray.push(rectangle(rectToBreak.x0, overlap.y0,
                                    overlap.x0-1, rectToBreak.y1));
        }
        else if (breakerRect.includes(rectToBreak.x1, rectToBreak.y0)) {
            newArray.push(rectangle(rectToBreak.x0, rectToBreak.y0,
                                    overlap.x0-1, overlap.y1));
            newArray.push(rectangle(rectToBreak.x0, overlap.y1+1,
                                    rectToBreak.x1, rectToBreak.y1));
        }
        else {
            newArray.push(rectangle(rectToBreak.x0, rectToBreak.y0,
                                    rectToBreak.x1, overlap.y0-1));
            newArray.push(rectangle(rectToBreak.x0, overlap.y0,
```

```
                                overlap.x0-1, overlap.y1));
            newArray.push(rectangle(rectToBreak.x0, overlap.y1+1,
                                rectToBreak.x1, rectToBreak.y1));
         }
      }
      else {
         if (breakerRect.includes(rectToBreak.x0, rectToBreak.y1)) {
            newArray.push(rectangle(rectToBreak.x0, rectToBreak.y0,
                                rectToBreak.x1, overlap.y0-1));
            newArray.push(rectangle(overlap.x1+1, overlap.y0,
                                rectToBreak.x1, rectToBreak.y1));
         }
         else if (breakerRect.includes(rectToBreak.x0, rectToBreak.y0)) {
            newArray.push(rectangle(rectToBreak.x0, overlap.y1+1,
                                rectToBreak.x1, rectToBreak.y1));
            newArray.push(rectangle(overlap.x1+1, rectToBreak.y0,
                                rectToBreak.x1, overlap.y1));
         }
         else {
            newArray.push(rectangle(rectToBreak.x0, rectToBreak.y0,
                                rectToBreak.x1, overlap.y0-1));
            newArray.push(rectangle(overlap.x1+1, overlap.y0,
                                rectToBreak.x1, overlap.y1));
            newArray.push(rectangle(rectToBreak.x0, overlap.y1+1,
                                rectToBreak.x1, rectToBreak.y1));
         }
      }
}
```

D.3 Excerpts of ocr-layout/ocr-char-stats.cc

```
bool is_text_block(rectarray &bboxes, vector<int>& wrap_around, int zone_num)
{
int i;
    int biggest_x = 0;
    int width_sum = 0;
    int height_sum = 0;
    floatarray y0;
    floatarray hist;

    // Collect the y0 values of the bounding boxes.
```

```
    for (i=0; i<bboxes.length(); i++) {
        y0.push(bboxes[i].y0);
        width_sum += bboxes[i].x1 - bboxes[i].x0;
        height_sum += bboxes[i].y1 - bboxes[i].y0;
        if (bboxes[i].x1 > biggest_x)
           biggest_x = bboxes[i].x1;
    }
    int avg_width = width_sum / bboxes.length();
    int avg_height = height_sum / bboxes.length();

// Create the y0 histogram.
    calc_hist(hist, y0);
    gauss1d(hist, 1.0);

    // Get its peaks.
    floatarray peak;
    for (i=2; i<hist.length()-2; i++)
    {
        if ((hist[i] > 1) &&
           ((hist[i] > hist[i-1]) && (hist[i] >= hist[i+1]))) {
            float temp = hist[i];
            peak.push(temp);
        }
    }
    if ((hist[i] > 1) && (hist[i] > hist[i-1])) {
        float temp = hist[i];
        peak.push(temp);
    }
    // If there are no peaks this is not a text block so return.
    if (peak.length() == 0)     { return false; }

    // Now create the peak histogram.
    floatarray peak_hist;
    calc_hist(peak_hist, peak);

// The average number of occurences dictating the peaks is ...
    int max_peak = 0;
    int avg_num_occurences = 2;
    for (i=peak_hist.length()-1; i>=0; i--) {
        if (peak_hist[i] > max_peak) {
            max_peak = peak_hist[i];
            if (i > 2)
                avg_num_occurences = i;
```

```
    }
}

// Now find the y-values given the peak threshold.
intarray line;
for (i=2; i<hist.length()-2; i++) {
    if ((hist[i] > (0.5 * avg_num_occurences)) &&
       ((hist[i] > hist[i-1]) && (hist[i] >= hist[i+1]))){
        line.push(i);
    }
}
if ((hist[i] > (0.5 * avg_num_occurences)) &&
    (hist[i] > hist[i-1])) {
    line.push(i);
}

int num_lines = line.length();
// If no lines were found it's not a text block so return false.
if (num_lines == 0)     { return false; }

// Now get the average separation and if it's too high return false.
int sum_line_seps = 0;
for (i=1; i<num_lines; i++)
    sum_line_seps += line[i] - line[i-1];
int avg_line_sep = sum_line_seps / num_lines;
if ((avg_line_sep / avg_height) > 5)        { return false; }

// Calculate the compacted widths (summation of box widths)
// and the x-range of the boxes.
int compacted_line_length[num_lines];
int xmin[num_lines], xmax[num_lines];
for (i=0; i<num_lines; i++) {
    compacted_line_length[i] = 0;
    xmin[i] = biggest_x;
    xmax[i] = 0;
}

for (i=0; i<bboxes.length(); i++) {
    int j = 0;
 bool line_found = false;
    while ((!line_found) && (j < num_lines)) {
        if ((bboxes[i].y0 > (line[j] - avg_width)) &&
            (bboxes[i].y0 < (line[j] + avg_width))) {
```

```
                line_found = true;
                compacted_line_length[j] += bboxes[i].x1 - bboxes[i].x0;
                if (bboxes[i].x0 < xmin[j])
                    xmin[j] = bboxes[i].x0;
                if (bboxes[i].x1 > xmax[j])
                    xmax[j] = bboxes[i].x1;
            }
            else
                j++;
        }
    }

    // Using these numbers calculate the density.
    float density[num_lines];
    int line_length[num_lines];
    int longest_line = 0;
    for (i=0; i<num_lines; i++){
        if ((line_length[i] = xmax[i] - xmin[i]) > longest_line)
            longest_line = line_length[i];
        if (line_length[i] > 0)
            density[i] = (float)compacted_line_length[i] / (float)line_length[i];
        else
            density[i] = 0;
    }

    // Adjust the number of lines if some have zero length
    // and tally how many are full length.
    int zero_length_lines = 0;
    int full_length[num_lines];
    for (i=0; i<num_lines; i++) {
        if (compacted_line_length[i] == 0)
            zero_length_lines++;
        if (line_length[i] > 0.8 * longest_line)
            full_length[i] = 1;
        else
            full_length[i] = 0;
    }
    int actual_num_lines = num_lines - zero_length_lines;

// Count the number of good lines.
    int num_good_lines = 0;
    for (i=0; i<num_lines; i++)
        if (density[i] > 0.5)   { num_good_lines++; }
```

```
    // If three in a row are not full length assume
    // it's a wrap around text block.
    for (i=2; i<num_lines; i++)
        if ((full_length[i] == 0) && (full_length[i-1] == 0)
            && (full_length[i-2] == 0)) {
            wrap_around[zone_num] = 1;
            i = num_lines;
    }

    // Return true or false depending on what fraction of the lines are good.
    switch (actual_num_lines) {
        case 2 : if (num_good_lines >= 1)   { return true; }
                 else                       { return false; }
        case 3 : if (num_good_lines >= 2)   { return true; }
                 else                       { return false; }
        case 4 : if (num_good_lines >= 3)   { return true; }
                 else                       { return false; }
        default : if (num_good_lines >= (0.8 * actual_num_lines))
                    return true;
                  else
                    return false;
    }
}
```