UNIVERSITY OF OSLO
Department of Informatics

High precision text extraction from PDF documents

Øyvind Raddum Berg

Friday 29th April, 2011

Acknowledgments

It is a pleasure to thank the many people who made this thesis possible.

Firstly, I would like to thank my advisor, Stephan Oepen, for being a great source of useful feedback, for being flexible, and for being easy to work with. Secondly, i thank Johan Benum Evensberget for getting me started, and Jonathon Read for patiently testing unfinished code.

I'll forever be indebtful to all my friends and fellow students, both to those who helped and motivated me with my thesis work, and to those who helped me avoid it.

A sincere *thank you* to my family for their love and support. Without your help this thesis would have remained but a dream. A special thank you goes to *Far* for always believing in me, and to Egil for always having been an inspiration.

Finally, to my bestest Regine. Thank you for your love, your patience and your smiles. *May the sun keep shining forever.*

Contents

Chapter 1

Introduction

1.1 Motivation

One of the great advances in the digital era has been to enable us to store vast amounts of documents electronically. The advantages electronic documents have over documents stored on paper are plentiful: easy storage, little maintenance, efficient retrieval and sharing. Another firm advantage is that electronic documents have an explicit internal structure which is easily accessible. This structure can be physical (*e.g.* pages, columns, paragraphs, tables, *etc*), logical (*e.g.* titles, abstracts, sections, *etc*) or both. This structural information can be very useful for locating information contained within the document, for Information Retrieval, or for helping for example visually impaired access the information.

It is indeed curious then, that the Portable Document Format (PDF) standard, one of the most used document standards for electronic document archiving, foregoes this, and does not normally represent information in a manner which preserves structure or semantics in a computer-understandable way. PDF documents are primarily organized at a comparatively low level of typesetting instructions.

This thesis concerns itself with trying to recreate some notion of structure from the information which is intact, with the primary goal of analysing scholarly literature.

Highlighting the need for this strctured information, a standard for embedding it in PDF files has more recently been developed under the name PDF/Universal Accessibility, or “Tagged PDF” (PDF/UA). However, because of the quantity of documents created without this information, trying to recreate it remains a very important problem to solve.

1.2 Background

1.2.1 The PDF standard

PDF is a document standard which is the *de facto* standard for printable documents on the web, as well as frequently used for document archival and exchange. It was created by Adobe in 1993, and subsequently officially released as the open standard ISO 32000-1:2008 (2008). The file format was originally created to represent documents in a fixed-layout manner, while being independent from the platform and application used to view it.

PDF can actually be seen as something of a composite standard, which unifies at least three technologies, which together constitute a way to create documents:

1. A simplified subset of the PostScript page description programming language which leaves out general programming constructs like loops and branches, but includes all graphical operations to draw the layout, text and images.
2. A font embedding system which allows a document to carry the fonts it needs to make sure it will display as it was designed.
3. A structured storage system which stores objects of data, images or fonts inside a PDF document.

All the data objects are represented in a visually oriented way, as a sequence of operators which, when interpreted by a PDF parser, will draw the document on a page canvas. This is a logical approach considering its roots as a PostScript successor and its orientation towards desktop publishing applications, but the implications are unfortunate for anyone who wants access to the text in a structured manner.

Interpretation of these operators will provide us with all the individual characters, as well as their formatting and position on the page. However, they generally do not convey information about higher level text units such as words, lines or columns, that kind of information is left implicit by whitespace. Furthermore, the fragments comprising the text on a page may consist of an individual character, a part of a word, a whole line, or any combination thereof. Complicating matters further, there are no rules governing the order in which text is encoded in the document. For example, to produce a page with a two-column layout, the page could be drawn by first drawing the first line of the left column, then the first on the right, then the second on the left *etc.* Obtaining text in the logical reading order obviously requires that the text in the left column be processed before the one on the right, so a naïve output based on the encoded order of the text might produce undesirable results.

Since the standard is now open and free for anyone to use, we are fortunate to have several mature, open source libraries for working with PDF documents to handle all the low-level parsing. For this project we will use PDFBox (n.d.).

1.3 Problem description

In order to be able to perform many operations on the extracted content, it is vital to have obtained a logical structure of the text. Since we are working with principally visually oriented data, it was decided to pursue a method to approximate this structure by using all the visual clues and information we have available.

The data presented in a PDF file consists of streams of data; by placing less importance on the order of elements of within the streams, and more on the visual result obtained by "rendering" (or at least evaluating all the PDF operations) the file, the problem of making sense of these data is shifted slightly from what essentially amounts to stream-processing, into a domain related to *computer vision*.

This essentially corresponds to the same problem tackled by Optical Character Recognition (OCR) software, just without the need to perform the actual character recognition. Since OCR has been researched for a long time, one could reasonably hope that existing methods from this field could be adopted for this new context and benefit from the more plentiful information available.

The process of "understanding" a document in this context is called document layout analysis, a process which is frequently treated as two subsequent processes. First a page image is subject to a geometric layout analysis, the result of which is then used as input for a logical layout analysis to ultimately provide a logical representation of the content.

1.3.1 Geometric layout analysis

The goal of this analysis is to produce a hierarchical representation of a page in terms of *blocks* of homogenous content, which implicitly contains the spatial relationship between them. This is principally done through geometric and spatial information, *i.e.* the size and position of content. The resulting structure allows describing a page at different levels of detail, *e.g.* the content which corresponds to a linguistic word can thus be seen as the whole word itself, as part of a line of text, as a part of a block of text lines, *etc.*

This process of segmenting different parts of document content in this way is a well known problem, especially in the context of OCR software, and is normally called *geometric layout analysis*. This is the process other PDF text extraction projects have labeled boxing/deboxing, so named after the boxes content is grouped in. Even though we have more information when it comes to PDF documents than in an OCR context, the process remain essentially the same. A thorough, if a bit dated, overview can be found in Cattoni, Coianiz, and Messelodi (1998)

After having obtained all the content by parsing the PDF file, we find that the order in which it is received may not necessarily correspond to the logical order in which it should be read. To properly reorder the content, it is segmented into more manageable blocks, each representing a maximal region of the page with homogenous content.

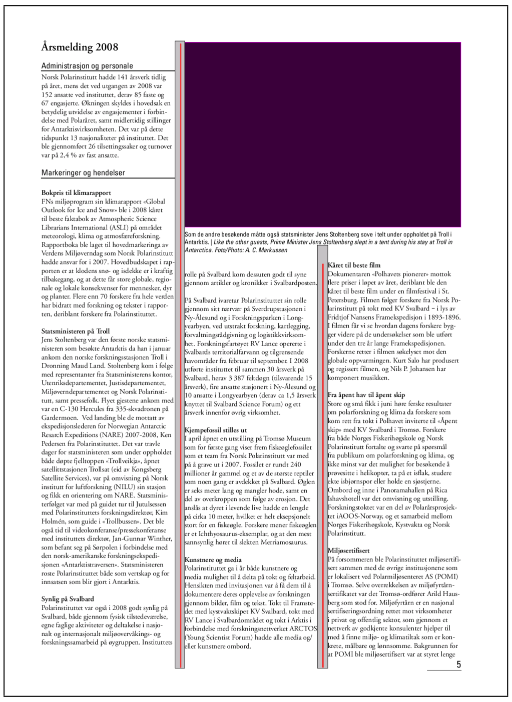
The main problem while performing this segmentation is essentially to decide the function of whitespace between content, *i.e.* whether it represents character spacing, word spacing, or if it divides two non-connected pieces of text. The reason why this is difficult is that the boundaries for these classes of spacing might overlap, for example when the distance separating two words might be bigger than that between two columns of text. In any case these boundaries will vary a great deal between documents. Also horizontal spacing gives some problems when it comes to deciding whether or not two text fragments are on the same line, consider for example $^{\text{superscript}}$, $_{\text{subscript}}$ and complex equations. Although this is more thoroughly done in the logical layout analysis phase, some floating text like sidenotes and graphic figures, *etc* should also be separated out early.

Literature on the subject suggests three classes of approaches for this kind of analysis, top-down, bottom-up and hybrid. Although these categories do not seem to be clear-cut and without overlap, they still provide a useful way to differentiate the many algorithms.

Bottom-up algorithms start from individual pixels, construct *connected components* from the pixels which constitute characters, and then iteratively cluster these connected components into words, lines, *etc.* Superficially, bottom-up algorithms seem to generally be variations of considering a text fragment in context of the immediately surrounding text, and then group together those closer to each other than x, where x is hopefully determined in a smart way.

Top-down algorithms start from the entire image of a page, and iteratively subdivide it into smaller pieces until some condition is met, after which the current state is considered the final segmentation. This approach naturally favours establishing a higher level layout representation of the document layout, and then grouping the content according to that.

With such a high level layout representation, text grouping is easier, because it is done within certain constraints. Also, classification of horizontal whitespace is then reduced to only two options (*i.e.* does this whitespace divide two words or not, it will be known if it separates columns). Note especially in Figure 1.1 how even a complex layout can be described by listing just a few column boundaries.


Årsmelding 2008

Administrasjon og personale

Norsk Polarinstitutt hadde 141 årsverk tidlig på året, mens det ved utgangen av 2008 var 152 ansatte ved instituttet, derav 85 faste og 67 engasjerte. Økningen skyldes i hovedsak en betydelig utvidelse av engasjementer i forbindelse med Polaråret, samt midlertidig stillinger for Antarktisvirksomheten. Det var på dette tidspunkt 13 nasjonaliteter på instituttet. Det ble gjennomført 26 tilsettingssaker og turnover var på 2,4 % av fast ansatte.

Markeringer og hendelser

Bokpris til klimarapport
FNs miljøprogram sin klimarapport «Global Outlook for Ice and Snow» ble i 2008 kåret til beste faktabok av Atmospheric Science Librarians International (ASLI) på området meteorologi, klima og atmosfæreforskning. Rapportboka ble laget til hovedmarkeringa av Verdens Miljøverndag som Norsk Polarinstitutt hadde ansvar for i 2007. Hovedbudskapet i rapporten er at klodens snø- og isdekke er i kraftig tilbakegang, og at dette får store globale, regionale og lokale konsekvenser for mennesker, dyr og planter. Flere enn 70 forskere fra hele verden har bidratt med forskning og tekster i rapporten, deriblant forskere fra Polarinstituttet.

Statsministeren på Troll
Jens Stoltenberg var den første norske statsministeren som besøkte Antarktis da han i januar ankom den norske forskningsstasjonen Troll i Dronning Maud Land. Stoltenberg kom i følge med representanter fra Statsministerens kontor, Utenriksdepartementet, Justisdepartementet, Miljøverndepartementet og Norsk Polarinstitutt, samt pressefolk. Flyet gjestene ankom med var en C-130 Hercules fra 335-skvadronen på Gardermoen. Ved landing ble de mottatt av ekspedisjonslederen for Norwegian Antarctic Resarch Expeditions (NARE) 2007-2008, Ken Pedersen fra Polarinstituttet. Det var travle dager for statsministeren som under oppholdet både døpte fjelltoppen «Trollveikja», åpnet satellittstasjonen Trollsat (eid av Kongsberg Satellite Services), var på omvisning på Norsk institutt for luftforskning (NILU) sin stasjon og fikk en orientering om NARE. Statsministerfølget var med på guidet tur til Jutulsessen med Polarinstituttets forskningsdirektør, Kim Holmén, som guide i «Trollbussen». Det ble også tid til videokonferanse/pressekonferanse med instituttets direktør, Jan-Gunnar Winther, som befant seg på Sørpolen i forbindelse med den norsk-amerikanske forskningsekspedisjonen «Antarktistraversen». Statsministeren roste Polarinstituttet både som vertskap og for innsatsen som blir gjort i Antarktis.

Synlig på Svalbard
Polarinstituttet var også i 2008 godt synlig på Svalbard, både gjennom fysisk tilstedeværelse, egne faglige aktiviteter og deltakelse i nasjonalt og internasjonalt miljøovervåkings- og forskningssamarbeid på øygruppen. Instituttets

Som de andre besøkende måtte også statsminister Jens Stoltenberg sove i telt under oppholdet på Troll i Antarktis. | *Like the other guests, Prime Minister Jens Stoltenberg slept in a tent during his stay at Troll in Antarctica. Foto/Photo: A. C. Markussen*

rolle på Svalbard kom dessuten godt til syne gjennom artikler og kronikker i Svalbardposten.

På Svalbard ivaretar Polarinstituttet sin rolle gjennom sitt nærvær på Sverdrupstasjonen i Ny-Ålesund og i Forskningsparken i Longyearbyen, ved utstrakt forskning, kartlegging, forvaltningsrådgivning og logistikkvirksomhet. Forskningsfartøyet RV Lance opererte i Svalbards territorialfarvann og tilgrensende havområder fra februar til september. I 2008 utførte instituttet til sammen 30 årsverk på Svalbard, herav 3 387 feltdøgn (tilsvarende 15 årsverk), fire ansatte stasjonert i Ny-Ålesund og 10 ansatte i Longyearbyen (derav ca 1,5 årsverk knyttet til Svalbard Science Forum) og ett årsverk innenfor øvrig virksomhet.

Kjempefossil stilles ut
I april åpnet en utstilling på Tromsø Museum som for første gang viser frem fiskeøglefossilet som et team fra Norsk Polarinstitutt var med på å grave ut i 2007. Fossilet er rundt 240 millioner år gammel og et av de største reptiler som noen gang er avdekket på Svalbard. Øglen er seks meter lang og mangler hode, samt en del av overkroppen som følge av erosjon. Det anslås at dyret i levende live hadde en lengde på cirka 10 meter, hvilket er helt eksepsjonelt stort for en fiskeøgle. Forskere mener fiskeøglen er et Ichthyosaurus-eksemplar, og at den mest sannsynlig hører til slekten Merriamosaurus.

Kunstnere og media
Polarinstituttet ga i år både kunstnere og media mulighet til å delta på tokt og feltarbeid. Hensikten med invitasjonen var å få dem til å dokumentere deres opplevelse av forskningen gjennom bilder, film og tekst. Tokt til Framstedet med kystvaktskipet KV Svalbard, tokt med RV Lance i Svalbardområdet og tokt i Arktis i forbindelse med forskningsnettverket ARCTOS (Young Scientist Forum) hadde alle media og/eller kunstnere ombord.

Kåret til beste film
Dokumentaren «Polhavets pionerer» mottok flere priser i løpet av året, deriblant ble den kåret til beste film under en filmfestival i St. Petersburg. Filmen følger forskere fra Norsk Polarinstitutt på tokt med KV Svalbard – i lys av Fridtjof Nansens Framekspedisjon i 1893-1896. I filmen får vi se hvordan dagens forskere bygger videre på de undersøkelser som ble utført under den tre år lange Framekspedisjonen. Forskerne retter i filmen søkelyset mot den globale oppvarmingen. Kurt Salo har produsert og regissert filmen, og Nils P. Johansen har komponert musikken.

Fra åpent hav til åpent skip
Store og små fikk i juni høre ferske resultater om polarforskning og klima da forskere som kom rett fra tokt i Polhavet inviterte til «Åpent skip» med KV Svalbard i Tromsø. Forskere fra både Norges Fiskerihøgskole og Norsk Polarinstitutt fortalte og svarte på spørsmål fra publikum om polarforskning og klima, og ikke minst var det mulighet for besøkende å prøvesitte i helikopter, ta på et isflak, studere ekte isbjørnspor eller holde en sjøstjerne. Ombord og inne i Panoramahallen på Rica Ishavshotell var det omvisning og utstilling. Forskningstoktet var en del av Polarårsprosjektet iAOOS-Norway, og et samarbeid mellom Norges Fiskerihøgskole, Kystvakta og Norsk Polarinstitutt.

Miljøsertifisert
På forsommeren ble Polarinstituttet miljøsertifisert sammen med de øvrige institusjonene som er lokalisert ved Polarmiljøsenteret AS (POMI) i Tromsø. Selve overrekkelsen av miljøfyrtårnsertifikatet var det Tromsø-ordfører Arild Hausberg som stod for. Miljøfyrtårn er en nasjonal sertifiseringsordning rettet mot virksomheter i privat og offentlig sektor, som gjennom et nettverk av godkjente konsulenter hjelper til med å finne miljø- og klimatiltak som er konkrete, målbare og lønnsomme. Bakgrunnen for at POMI ble miljøsertifisert var at styret lenge

5

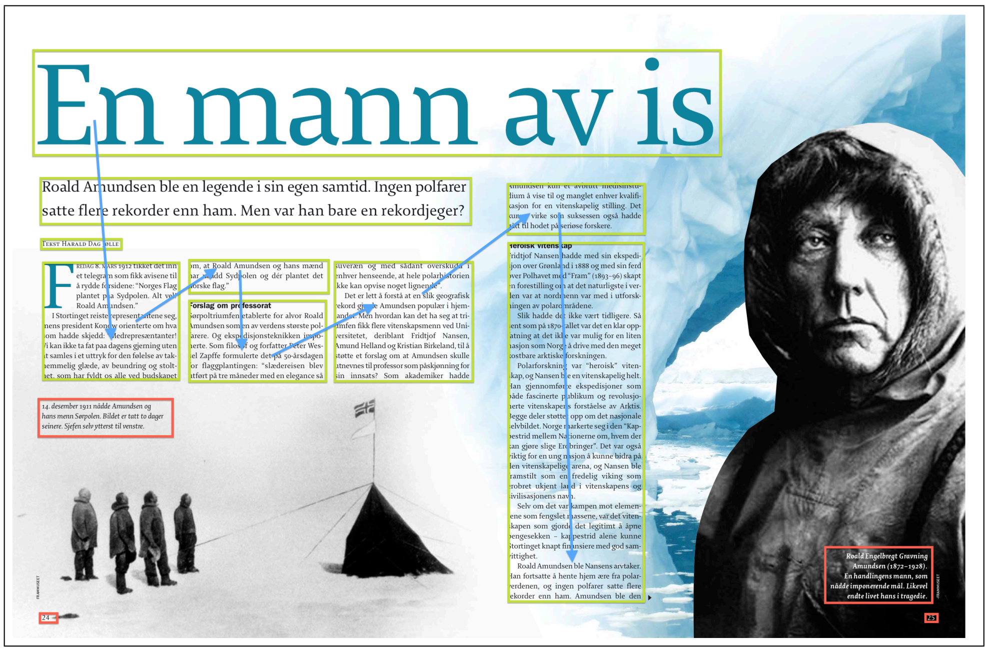
Figure 1.1: Columnal layout described by rectangles

Hybrid algorithms are those which are a mix of the two, or do not really fit in any of the two categories.

For an overview over a selection of algorithms and which category they are deemed to belong to, see Mao, Rosenfeld, and Kanungo (2003). Of the many techniques available, many inherently work only on bitmapped data (for example identifying blocks by smearing images), while others seem more general in that they can lend themselves to this new context of PDF documents instead of OCR.

1.3.2 Ordering of text content

The process of ordering the identified text blocks in its logical order is also crucial in order to successfully extract information, as any failure to do so will result in garbled, wrongly combined or out of sequence fragments of text. The essential problem is to look at all the whole page, and figure out in which order a person would read all the contained blocks. For that it is necessary to consider all the text blocks and do a topological sort of them. Figure 1.2 shows recovered reading order for a page with complex layout. All text on a page is sorted; content like *i.e.* page numbers, which is not part of the body text, is separated at a later stage.



En mann av is

Roald Amundsen ble en legende i sin egen samtid. Ingen polfarer satte flere rekorder enn ham. Men var han bare en rekordjeger?

Tekst Harald Dag Jølle

FREDAG 8. MARS 1912 tikket det inn et telegram som fikk avisene til å rydde forsidene: "Norges Flag plantet paa Sydpolen. Alt vel! Roald Amundsen."

I Stortinget reiste representantene seg, mens president Konow orienterte om hva som hadde skjedd: "Medrepresæntanter! Vi kan ikke ta fat paa dagens gjerning uten at samles i et uttryk for den følelse av taknemmelig glæde, av beundring og stolthet, som har fyldt os alle ved budskapet om, at Roald Amundsen og hans mænd har naadd Sydpolen og dér plantet det norske flag."

Forslag om professorat

Sørpoltriumfen etablerte for alvor Roald Amundsen som en av verdens største polfarere. Og ekspedisjonsteknikken imponerte. Som filosof og forfatter Peter Wessel Zapffe formulerte det på 50-årsdagen for flaggplantingen: "slædereisen blev utført på tre måneder med en elegance så suveræn og med sådant overskudd i enhver henseende, at hele polarhistorien ikke kan opvise noget lignende".

Det er lett å forstå at en slik geografisk rekord gjorde Amundsen populær i hjemlandet. Men hvordan kan det ha seg at triumfen fikk flere vitenskapsmenn ved Universitetet, deriblant Fridtjof Nansen, Amund Helland og Kristian Birkeland, til å støtte et forslag om at Amundsen skulle utnevnes til professor som påskjønning for sin innsats? Som akademiker hadde Amundsen kun et avbrutt medisinstudium å vise til og manglet enhver kvalifikasjon for en vitenskapelig stilling. Det kunne virke som suksessen også hadde gått til hodet på seriøse forskere.

Heroisk vitenskap

Fridtjof Nansen hadde med sin ekspedisjon over Grønland i 1888 og med sin ferd over Polhavet med "Fram" (1893–96) skapt en forestilling om at det naturligste i verden var at nordmenn var med i utforskningen av polarområdene.

Slik hadde det ikke vært tidligere. Så sent som på 1870-tallet var det en klar oppfatning at det ikke var mulig for en liten nasjon som Norge å drive med den meget kostbare arktiske forskningen.

Polarforskning var "heroisk" vitenskap, og Nansen ble en vitenskapelig helt. Han gjennomførte ekspedisjoner som både fascinerte publikum og revolusjonerte vitenskapens forståelse av Arktis. Begge deler støttet opp om det nasjonale selvbildet. Norge markerte seg i den "Kappestrid mellem Nationerne om, hvem der kan gjøre slige Erobringer". Det var også viktig for en ung nasjon å kunne bidra på den vitenskapelige arena, og Nansen ble framstilt som en fredelig viking som erobret ukjent land i vitenskapens og sivilisasjonens navn.

Selv om det var kampen mot elementene som fengslet massene, var det vitenskapen som gjorde det legitimt å åpne pengesekken – kappestrid alene kunne Stortinget knapt finansiere med god samvittighet.

Roald Amundsen ble Nansens arvtaker. Han fortsatte å hente hjem ære fra polarverdenen, og ingen polfarer satte flere rekorder enn ham. Amundsen ble den

14. desember 1911 nådde Amundsen og hans menn Sørpolen. Bildet er tatt to dager seinere. Sjefen selv ytterst til venstre.

Roald Engelbregt Gravning Amundsen (1872–1928). En handlingens mann, som nådde imponerende mål. Likevel endte livet hans i tragedie.

24

25

Figure 1.2: Determining reading order. Expected reading order among the green boxes marked with arrows. Text which might not be qualified as body text in red

1.3.3 Logical layout analysis

While geometric layout analysis leaves us with a complete physical representation of a page in terms of blocks of segmented content, the next step is to somehow use that to derive a logical structure.

The essential idea is to both assign *labels* to, and figure out the logical relationship between these blocks based on an *a priori* model of a general document. These labels are meant to correspond to concepts that humans perceive as meaningful with respect to the content at hand, typical examples would be `title`, `body text`, `table`, *etc.* The relationships will be for example that a section header precedes and introduces the body text of paragraph, or that a section header belongs beneath the main title. Based on this information it is possible to construct a hierarchical tree-like structure which represents the logical information we have found.

An important consideration here is the set of labels the application will use. It is clear that some labels, say for example `title` and `page number` are very general and will apply to large numbers of documents. Likewise it also seems clear that some possible labels will be appropriate for only specific sets of documents. For example `abstract` and `footnote` will make more sense for research papers than for letters, while the converse will be true for `sender`, `recipient` and `signature`.

In other words, this process in particular could be adjusted or extended to differentiate between a bigger number of labels for a specific set of documents, but then at the cost of restricting the domain on which the program will reliably operate. Although several backends for different types of documents could easily be imagined, this project will include one generic set of labels. Since text extraction from scholarly literature is a primary goal for the project, the set of will be sufficient to recognize and represent that, but apart from that everything is kept as general as possible.

In order to perform this logical analysis, two things are essential: First, we need a foundation which enables us to reason about the document layout and content. In this project we naturally depend on the result of our geometric layout analysis, which leaves us with a high quality and high level representation of the content.

Additionally, we also need some kind of model which describes what we can expect a document to look like, and which characteristics a piece of text should exhibit in order to receive a label. Starting thus with the blocks from the physical analysis, we can consider the labelling of text as a classification problem. To illustrate this point a bit, consider the case of deciding whether a block can be classified as a subsection header or not. In that case, positive answers to most of the following questions might very well indicate that it in fact is:

- is the text preceded by some text which was classified as a section header?
- is the text succeeded by what is considered the document body text?
- is the text formatted differently, perhaps with a bold or different font face or a bigger font size, than body text?
- is the text indented compared to the general column start?
- does the text start with a number, letter, dot, star, *etc...*?

This process is relatively simple for specialized and homogenous groups of content, but in the larger and more general picture it is much harder. There is an incredible number of documents produced which exhibit an immense variety in design and style, and so it goes without saying that the scope of this analysis will necessarily need to be narrowed down to the subset which *has* a structure to be discovered, and which manifests a relatively consistent design.

1.3.4 Reparation of publication-related damage

Additionally, several publishing-related artifacts should be cleaned up, this would include decomposing ligatures into the individual characters they consist of, removing page numbers, migrate specific and obsolete character sets to Unicode, remove control characters, remove word-dividing hyphens, and recombine lines. Except for recombination of lines and removal of page numbers, this topic will not be discussed in much detail as code to perform these functions was inherited when the program started, and PDFBox will now do some of it natively.

1.4 Technical terms

In this thesis a variety of different terms originating from as diverse fields as *natural language processing*, *typography* and plain *computer science* will be used. In addition, some terms require a more precise definition in this context. For this reason a short list of concepts is presented below.

baseline in typography, the baseline is the line upon which most characters are placed. To illustrate, an s is fully over the baseline, while a p has a descender which goes crosses and goes underneath it.

Branch and Bound a general algorithm for finding optimal solutions of various optimization problems, especially in discrete and combinatorial optimization. It consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded *en masse*, by using upper and lower estimated bounds of the quantity being optimized (source: Wikipedia).

character set Refers to a numerical representation of a set of characters so that text can be represented numerically. Earlier there were many character sets for different languages in use, today Unicode is the most important one because it is a superset of all the other character sets and its use hence essentially eliminates character sets as something to be considered.

code point a numerical offset in a character set, as opposed to the character or item it represents.

control character a code point in a character set, that does not in itself represent a printable character. Fonts embedded in PDF documents might use custom encodings which, if no corresponding Unicode translation table is supplied, might cause output text to be garbled and often contain control characters which will make the output appear as binary data.

dehyphenation the process of recombining a word which has been split over two lines with a hyphen (–) while typesetting. The problem is to decide whether or not to keep the hyphen when recombining, as some should be kept as they have a semantic meaning.

descender in typography, a descender is the part of a glyph which is placed below the baseline.

glyph a visual representation of a character in a specific font and style.

Information Retrieval the area of study concerned with searching for documents, for information within documents, and for metadata about documents, as well as that of searching relational databases and the World Wide Web (source:Wikipedia).

kerning the process of adjusting the spacing between characters in a proportional font, usually to achieve a visually pleasing result.

ligature In typography, a ligature occurs when two or more characters are joined into a single glyph. An example is when *f* and *i* are joined into *ﬁ*.

Manhattan layout is a page layout with the property that all text and graphics regions can be separated by horizontal and vertical line segments. Pages which do not have this layout are considered more difficult to analyze.

Page segmentation refers to the first part of the geometric layout analysis, namely the segmentation of homogenous content into blocks. In the context of OCR, this is normally dependant on a preceding *skew analysis* to account for skewed content, but that is unnecessary for PDF documents.

PostScript a dynamically typed concatenative programming language created by John Warnock and Charles Geschke in 1982. PostScript is best known for its use as a page description language in the electronic and desktop publishing areas (source:Wikipedia).

river in typography, rivers are gaps which appears to run into a text paragraph because of accidental alignment of whitespace so that it lines up.

TrueType an outline font standard originally developed by Apple Computer in the late 1980s as a competitor to Adobe's Type 1 fonts used in PostScript (source: Wikipedia).

Unicode a computing industry standard for the consistent encoding, representation and handling of text expressed in most of the world's writing systems. If text is encountered in another character set, it is typically desirable to convert to Unicode.

1.5 Outline of thesis

The following chapters will deal with the different aspects of the project mentioned in the introduction.

- Chapter 2 describes the state of two prior projects which were inherited at the start of the project, shows a performance evaluation which was conducted, and lists some common problems while performing text extraction.
- Chapter 3 deals with how a suitable geometrical layout analysis algorithm was found, and how it was adapted and implemented.
- Chapter 4 is in part a continuation of Chapter 3, in that it describes topics which are related to the geometrical layout analysis. These topics, however, are purely technical and would have been issues regardless of which method would have been chosen in Chapter 3, and are hence collected in a chapter of their own.
- Chapter 5 presents the foundations of a logical layout analysis as described above. The geometric layout analysis was the main focus of the project, and this chapter thus above all shows how the result from that is of great help when developing this subsequent analysis.
- Chapter 6 describes the organization and implementation of the application which was developed.
- Chapter 7 presents a performance evaluation performed to assess how well the implemented geometrical analysis algorithm works for a collection of previously unseen documents, and discusses the most common imperfections observed in the output.
- Chapter 8 finally presents the conclusion based on a performance evaluation and briefly mentions possibilities for further work.

1.6 Major results

The main focus of the research performed in this project is the geometric layout analysis. The analysis took as its point of departure an existing theory, but during the project the main algorithm was substantially extended and adapted so that the final algorithm has some novelty. Several other related algorithms developed for the project for column boundary identification and recovery of reading order are only loosely based on the existent theory, and are thus original research. Additionally, new solutions for word segmentation and graphics separation and segmentation were found. Some imperfections in the PDFBox and FontBox libraries were found and patched.

The resulting geometric layout analysis is quite powerful, and represents the most important advance made in this project.

Chapter 2

Related work – Comparison of existing projects

At the start of the project in February 2010, there were already two existing projects, both of which could serve as the base for further work. These two were TextGrabber, developed at the University of Oslo (UiO), and the other was pdf2xml, written at German research centre das Deutsche Forschungszentrum für Künstliche Intelligen (DFKI). The two projects had overlapping goals, and were quite compatible source code-wise, since they used the same underlying library, PDFBox (n.d.).

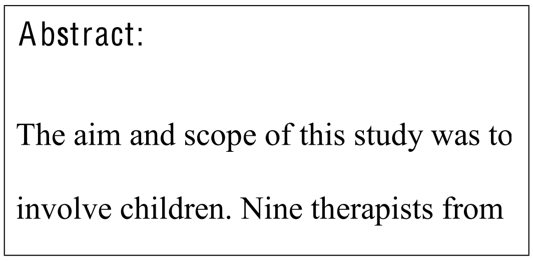
It seemed that in order to see which one were to serve as a starting point, it was logical to conduct a comparative study of the two projects to explore which of them would be the better match. Additionally, it was important to see to which extent they covered the goals which were planned for this project. To this end, a benchmark against which the applications would be tested was created.

Aside from just the performance in the benchmark, the programs would also be judged by the number of features which were present, if these were of the kind which would be wanted for this project. The final quality which would be judged was maintainability. This meant a quick evaluation of code quality, architecture, availability of unit tests, exported XML schema, *etc.*

2.1 Challenges and common problems

Before we present the comparison itself, it's useful to have a look at common problems encountered while processing PDF documents. All of these challenges complicate analysing PDF files in different ways.

- Many documents are laid out with bigger line spacing than normal. This fact complicates code which combines singular lines into blocks, as one must maintain an idea of average line distance. See Figure 2.1 for an example.

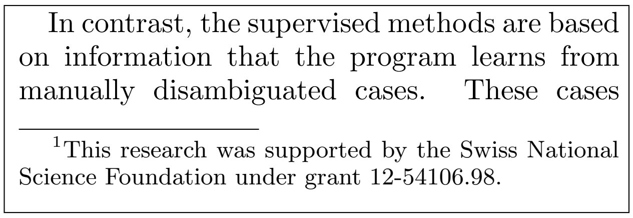

Abstract:

The aim and scope of this study was to

involve children. Nine therapists from

Figure 2.1: Large line spacing

- Another recurring problem is the separation of footnotes from body text. Unless a lot of care is taken, they will frequently end up being part of the running text or, as was the case with these two programs, frequently just dropped from the output altogether. See Figure 2.2 for an example of how a footnote might appear close to body text.

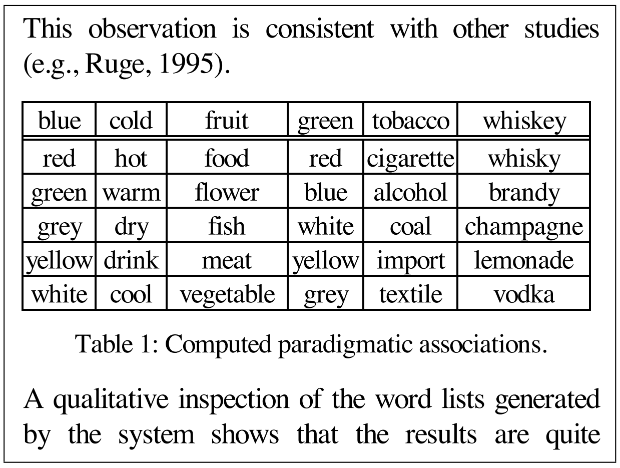

In contrast, the supervised methods are based on information that the program learns from manually disambiguated cases. These cases

[1]This research was supported by the Swiss National Science Foundation under grant 12-54106.98.

Figure 2.2: Footnote below paragraph

- Graphical figures and the textual content contained withing them will, if care is not taken to separate them out along with their contained textual contents, normally appear as text inside a column of body text. See Figure 2.3 for an example where a table appears with more or less the same width as the column it is enclosed in.

- Mathematical formulæ are notoriously difficult to deal with for several reasons. Some of the characters might be drawn with vector operations, the vertical position varies much and is obviously significant. Outside of running text they might conceivably be stripped out, but inlined in the text they have to be dealt with. Since it is very difficult to make any sense of the extracted equations, the most important thing is not to break the segmentation of surrounding text.



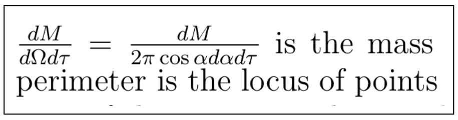
This observation is consistent with other studies (e.g., Ruge, 1995).

blue	cold	fruit	green	tobacco	whiskey
red	hot	food	red	cigarette	whisky
green	warm	flower	blue	alcohol	brandy
grey	dry	fish	white	coal	champagne
yellow	drink	meat	yellow	import	lemonade
white	cool	vegetable	grey	textile	vodka

Table 1: Computed paradigmatic associations.

A qualitative inspection of the word lists generated by the system shows that the results are quite

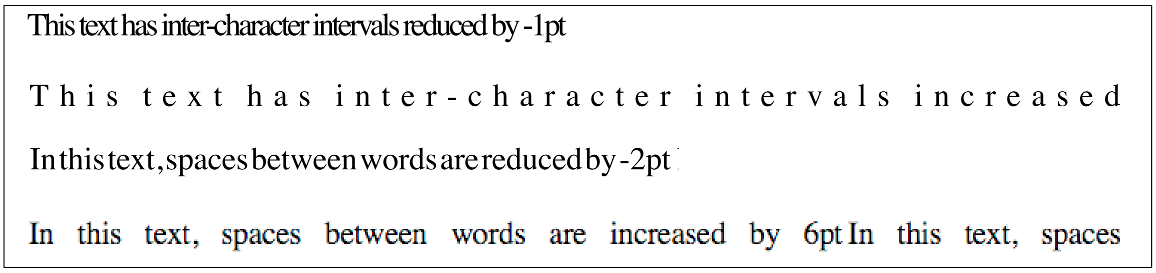
Figure 2.3: Figure in the middle of a column



$\frac{dM}{d\Omega d\tau} = \frac{dM}{2\pi \cos\alpha d\alpha d\tau}$ is the mass perimeter is the locus of points

Figure 2.4: A multiline equation

- Variable word and character spacing can be a big problem because it can fool the word segmentation algorithms. As can be seen in Figure 2.5, this variation can be very noticeable between documents.



This text has inter-character intervals reduced by -1pt

T h i s t e x t h a s i n t e r - c h a r a c t e r i n t e r v a l s i n c r e a s e d

In this text, spaces between words are reduced by -2pt

In this text, spaces between words are increased by 6pt In this text, spaces

Figure 2.5: Examples of variable spacing. This document was obvisouly crafted to highlight the possible variation, but none of the examples are implausible in the wild

2.1.1 Variation encountered in PDF files

Apart from the specific problems mentioned above, a no small part of the problem faced while extracting information from PDF files is the sheer variety of tools used to produce the files, and all the subtly different ways in which they encode them. To illustrate the variation in terms of tools used to produce documents, a script was written which collected some statistics from a collection of around 27000 documents from the Norwegian Open Research Archives (NORA)[1] collection. This is a primarily a collection of research literature, and should be a good representation of documents in the wild. The distribution of tools can be seen in Figure 2.6.

The different tools used will necessarily affect the typographical appearance of the content, the manner and order in which content is written, the amount and size of separational whitespaces, the representation of graphical content, the amount, format and encoding of fonts, and amount of meta-information. All of this will affect and frequently complicate the task at hand, especially because of the arising need to work around missing information and avoid assumptions which some times will turn out to be false.

Number of documents	**Name of tool**
1063	214 other programs with less than 50 documents
62	Windows NT
64	AFPL Ghostscript PDF Writer
64	ESP Ghostscript
65	easyPDF SDK
75	GPL Ghostscript PDF Writer
89	HP Digital Sending Device
106	AFPL Ghostscript
122	PDF PT
123	FrameMaker
150	OmniPage Pro
160	Pscript.dll
163	Acrobat Distiller
180	Writer
211	PrimoPDF
221	PDFCreator
225	Aladdin Ghostscript
228	Canon iR EUR
232	Acrobat Distiller for Windows
392	Adobe InDesign
641	GNU Ghostscript
874	dvips
2199	TeX
2538	Adobe Acrobat
3641	PScript5.dll
6750	Word
248	Broken documents
6222	Documents without metadata

Figure 2.6: Overview of tools used to create PDF files

[1]http://www.ub.uio.no/nora/search.html

2.2 Feature comparison

The two applications must have had largely the same general goals, so it is logical that the both have very similar lists of features. The main features, at least those which could easily be seen were implemented in the code are the following:

- Both projects provide adequate text extraction possibilities for many documents
- Both projects have some code to create geometrical boxes out of the many fragments of texts which is returned by the PDF parser library, and in that way group text
- Removal of page numbers from text
- Removal of control characters from text
- Reparation of publication related damage:
  - Removal of superfluous hyphens
  - Decomposition of ligatures

In addition, the pdf2xml tries to recognise title, author and abstract for academic papers. TextGrabber performs language identification and converts the legacy windows 1251 character set into Unicode.

2.3 Maintainability and architecture

The state of the code of both projects was assessed before the benchmark. Among the main considerations were:

- the code itself
- how maintainable and easily understandable the code appears
- possibilities and limitations
- Structure of exported eXtensible Markup Language (XML) schema
- Recovery of structure in the text

2.3.1 pdf2xml

There are both good and bad things to say about this project. To start with the positive, it has a relatively small amount of code, it uses clever abstractions for text grouping (`TextBox` and `TextGroup`) as well as a graph component of a math library which seems like a good fit, it integrates well with the inheritance model established by PDFBox (and should hence be very forward compatible).

On the other hand the same clever text grouping code (which does after all constitute the biggest part of the code) is also both difficult to understand and very poorly documented, and is hence in need of some refactoring and documentation. Less severe, but still worth mentioning, is the fact that some of the code is written in python, and that the source distribution omits a required bibtex python module to build it. This is however a much smaller component which at worst could even have been migrated to Java easily.

2.3.2 TextGrabber

At first sight the code gives a very good impression, all the code is written in Java and it has a working build script. It seems to be coded in a well planned manner with an extensible pipeline system for processing of the text in the XML tree, and it takes advantage of Java's `java.util.concurrent` framework for parallel processing. Also there is already some code to easily integrate it with Apache's Lucene[2] text indexing and search project.

On the negative side, the project requires a patch to the third party PDFBox library to work properly, something which could make upgrading to a newer version a bit more difficult. By superficial analysis it seems possible to rewrite that part of the code.

The XML schema, although easily extendable, defines very little document structure. The text transformer classes practically operate on raw text.

2.4 Performance overview

2.4.1 Benchmark

In order to perform the performance comparison in a thorough and unbiased way, a benchmark with several real world documents was needed. It was deemed that a set of 15 heterogenous documents chosen in a way as to cover as much variation as possible would cover at least a good subset of common problems. This number should probably ideally have been higher, but in this phase of the project it was also necessary to keep the amount of preparatory work to a minimum.

In order to construct this benchmark set, two collections of documents were considered. The first is the ACL Anthology Reference Corpus (ARC), which is provided by the Association for Computational Linguistics (ACL). As is described in Bird et al. (2008), it is a set of quite homogenous academic documents.

[2] `http://lucene.apache.org/`

The second collection comes from the NORA collection mentioned above. The main difference between the two for this purpose is that the documents in the NORA collection generally exhibit more variation in layout.

To obtain the wanted variation, it was decided to use the metadata embedded in the documents to extract information on what tool had been used to create the document, and which year it was made. A script then, based on that information, choose a subset of 30 documents from the total sets of about 16000 documents in such a way that it maximized variation of tools and periods of time. Then, based on a manual analysis, the set was trimmed to 15 documents to avoid documents which added little to the test, or were outright unsuitable (all text in images, saved with unrecoverable encoding, *etc*).

To perform the benchmark, a list of possible defects was compiled for each document, which in turn was then manually compared against the actual results of the different programs. The focus was both on the quality (missing or excessive whitespace, correct segmentation of text lines and words) and quantity (text fragments which are missing partially or altogether) of the text, and the order in which it was written.

A factor which complicated the comparison was the different feature sets supported by the projects; to which degree should one of the projects be penalized for not successfully accomplishing something which the other project does not even attempt?

The problems found (and some general observations) are presented below.

2.4.2 Results

The DFKI project, pdf2xml, generally performed best for the documents which belonged to the ARC set of documents; as it was developed with exactly that kind of documents in mind. Even though it will sometimes lose some body text, the results were quite good. Some headlines were unnecessary filtered out, and footnotes do very randomly appear in the output. For the documents from NORA documents it fared worse, crashing while processing 7 out of the (original) set of 30 documents. The code for finding abstract, author and title worked to a certain degree for the ARC documents, but was useless for the rest. Overall, the impression is thus kind of mixed, it worked satisfactory for ARC documents, and a bit less so for the ones from NORA.

TextGrabber on the other hand had no crashing problems, and generally provided useful output. Also here many documents had subtle or more serious errors when it came to missing, wrongly ordered, and/or partly garbled text, but the results were satisfactory in that the output contained most of the text and was generally indexable.

2.5 Conclusion

Based on the different evaluations above, it was a not entirely straightforward to choose one of the projects. pdf2xml offered what was probably the strongest text grouping engine, while in terms of codebase, features and performance, TextGrabber was deemed to be better.

The main problem was that neither of them offered any definite solutions for the most of the problems outlined above. There was *e.g.* no code to specifically handle lines with superflous line spacing, no graphics-aware code, not smart enough text grouping code, and too little structure in the exported XML schema.

Because of this, it was decided to try to go one step further, and create an application from scratch which would have as its foundation a stronger geometric text grouping engine, and then see how a logical analysis on top of that would perform. Many of the features in TextGrabber like decomposing ligatures, performing dehyphenation, correcting or eliminating control character, performing language detection, *etc* can then easily be incorporated this more generic framework.

Chapter 3

Geometric layout analysis – Page segmentation

The task of geometric layout analysis, and perhaps in particular Page segmentation from images for OCR purposes, remains a frequent topic in the literature, even after having been investigated for an extended amount of time by many researchers. Though many viable solutions have been proposed, it has been commented for example by Antonacopoulos, Gatos, and Bridson (2007) that they often are "devised with a specific application in mind and are fine-tuned to the test image dataset used by their authors".

The problem Page segmentation tries to solve is to divide a given page into homogenous zones, or perhaps more clearly expressed, to separate what visually seems to belong together into separate groups.

The essential problem encountered while doing this, is always in the end a decision of whether a whitespace between two words divides only words or if it divides columns. The reason why it is hard is that independent of font sizes and other known variables, inter-column distances are frequently smaller than inter-word distances.

There seems to be a somewhat inconsistent view as to what would belong in and what would be the output from such an analysis. The traditional approach seems to have been about enveloping text and image regions in rectangular shapes or polygons, while more recently it has been sought to recover the geometrical structure of the document. In practice, that process involves an approximation of column boundaries which are essential to determining reading order. There now also seems to be consensus that logical and physical layout concepts not be mixed, and that proper segmentation of *e.g.* logical paragraphs be deferred to a posterior logical analysis step.

While the core problem remains the same, the task at hand in *this* project is inherently different from that of segmenting images while preparing for an OCR analysis. Primarily, it is a question of the amount and quality of information available to the algorithms. For the layout analysis of a PDF document, all the operations required to render a page are known *a priori*, as is necessarily the result of said rendering. While controlling the interpretation of a document, it can also be very useful to decide *if* and *how* to render every component. Case in point is that careful separation of the rendering of graphics from that of text could for example potentially enable such an analysis to better deal with situations involving a so-called non-Manhattan layout, which have given OCR software so much trouble. There will also necessarily exist information about fonts used to render text, the sequence in which it was rendered, *etc.*

3.1 Choosing an algorithm

The differences in available rendering information mentioned above will obviously have implications for an algorithm which was originally developed for OCR. When starting to look for an algorithm which could be suitable for this new context, the following list of requirements were compiled:

1. That the algorithm not mainly operate on raw bitmapped data. Since we have much more and more precise data available in this context, it is essential that the algorithm be extendable to incorporate that information
2. That it has received some favourable verdict, preferable in a performance comparison
3. That it seems probable that added information will positively affect its performance
4. That it be general and, in order to avoid user intervention, not require manual parameters

Literature on the subject of OCR was consulted to see what was available in terms of algorithms, and two good starting points were found. A review of the state of the art in the OCR literature can be found in Mao et al. (2003). Since it was both a bit dated and none of the algorithms seemed to be exactly what was searched for, it was mainly used for orientation. A more recent performance review was conducted in Shafait, Keysers, and Breuel (2008), where 6 different algorithms were compared. This looked more interesting, and it was thus used as a starting point.

After eliminating 4 of the 6 algorithms based on requirements 1 and 2, the remaining algorithms were X–Y Cut and Constrained Textline Detection.

3.1.1 X–Y cut

This is a simple algorithm which was originally described in Nagy, Seth, and Viswanathan (1992). It belongs to the top-down algorithms described in Section 1.3.1, and works by creating a tree with a root node which represents the whole page. Subsequent recursive steps create subnodes in the tree by subdividing rectangles vertically or horizontally based on the content. The set of leaf nodes thus represents the final segmentation.

The algorithm itself might be a bit too simple for the use envisioned in this project, but the updated version described in Meunier (2005) where it was extended to allow more flexible layout and enable determination of reading order, would be a good match.

3.1.2 Breuel's Constrained Textline Detection

What is described in Breuel (2002) is actually a framework for document layout analysis with a four step process based on a whitespace analysis. The result of that analysis is used to identify columns, which again are used to segment text lines. The last step in the process is determination of reading order.

1. Whitespace analysis The first step is an analysis of the whitespace which surrounds all the text and graphics in terms of rectangular covers. The idea is in a way the opposite of the traditional process of enveloping groups of text in blocks; here what is sought is to use blocks of whitespace to discover the logical structure of the page.

To a certain extent this approach builds on work by others, especially Baird (1992), which was also mentioned by Shafait et al.. It is claimed (by Breuel himself) that this algorithm is far easier to implement, that it, when contrasted to other similar algorithms, outputs rectangles in decreasing size, and that it is quite efficient. This seemed plausible based on the theory presented in the paper, and especially the promise of easy implementation was considered a big plus by the author.

2. Identification of columns The second step in the process is identification of column boundaries based on the whitespace rectangles found. This is essentially a selection process with some constraints, *e.g.* that a possible column boundary must be adjacent to a certain number of characters, that it must have at least a certain aspect ratio, *etc.*

3. Finding text lines Next is a text line finding algorithm which takes the *obstacles* (as it views the content and the identified whitespace rectangles) and column boundaries into account, called *constrained line finding.*

Since text line segmentation is a hard problem for OCR, it has to take a lot of factors into account, most prominently among which are skew due to careless scanning of a page. This does of course not apply for PDF documents.

Identifying text lines has traditionally also been considered a problem because text lines which are physically close to each other (but non-related) might have different line parameters (inter-character and inter-line distances). By using a high-level layout description based on *gutters* (the identified columns), line segmentation only within the resulting blocks of text becomes relatively easy.

4. Finding reading order The last step is to determine the reading order of the content by a topological sort of the content. This process involves sorting the columns which has been identified, and then the lines contained within them.

3.1.3 Verdict

In Shafait et al. (2008), Breuel's Constrained Textline Detection seemed to receive the best verdict of the two layout analysis algorithms which were considered for this project; they concluded that "In the case of a heterogeneous document collection with different font sizes, styles, and scan resolutions, the constrained text-line finding algorithm appears to be the best choice".

In order to get a better impression of the two algorithms, trial implementations were performed of both.

For the X–Y Cut algorithm, the original, simple version was chosen, as the extended version does not change the characteristics of the algorithm in an important way. The main problem found with this algorithm was that it brought relatively little help in making decisions about what belonged together; while the algorithm does provide a context in which to make these decisions, they were still about making local choices (within the current subrectangle) based on observed text distances and font information. Since this is essentially a variation of the original problem, it was felt that other options should be explored.

As for the other algorithm, the sample implementation proved more interesting in that the identified rectangles provided additional help for text grouping. Additionally, the algorithm has few parameters (only a quality function for deciding which rectangles be preferred, and the identification of gutters/column separators.).

Both algorithms were deemed to benefit from the extra information available in the context of PDF analysis, but Breuel's Constrained Textline Detection was in the end chosen because it seemed to fulfill all the prerequisites set forth above, and it seemed like a very interesting match for the problem at hand.

3.2 Page segmentation – Implementation

The implementation in this project follows the spirit of the general process outlined by Breuel (2002). However, since this project deals with a different (though definitively related) problem, it is natural that it omits, adds or interchanges steps as has been deemed suitable.

3.2.1 Motivation

A short discussion about why one would want information about whitespace might perhaps be in order; it is after all about the only thing we would *not* be interested in for any information retrieval purpose. Furthermore it is already left implicit by the placement of everything else on a page.

The main point is indeed also not about the whitespace itself, but about the difference between what we could call *functional* and *nonfunctional* whitespace, where a functional whitespace can be said to have a separational quality.

The recurrent questions when one wants to determine whether text fragments are related or not, are normally questions about their mutual distance and aesthetic differences. What the information from this algorithm brings to the table is really the possibility to ask another set of questions based on *connectivity*.

Are these text fragments separated by something?

Though a long shot from Artificial Intelligence, this information will hopefully serve as an approximation to one of the patterns a human reader would subconsciously make use of to read a page.

3.2.2 Whitespace covering algorithm

For the original description of the algorithm, the interested reader might want to consult Section 2 of Breuel (2002). Since the core algorithm remains the same in this project, this section will necessarily have to be a paraphrase of what is described there.

The algorithm finds a cover of the background whitespace of a document in terms of maximal empty rectangles. It is a top-down algorithm, which means it uses a whole page as its starting point, and works in a way analogous to quicksort or Branch and

Bound algorithms. Whitespace rectangles are returned in order of decreasing quality (the quality function which determines which rectangles we want is detailed below), which means the output should be globally optimal in that no other combination of covering rectangles should cover the page better.

The iterative steps are illustrated in Figure 3.1, where (a) shows the start of the algorithm. We start with a bound and a set of rectangles, which are called *obstacles*. If the set is empty, it means that the bound is a maximal rectangle according to the other obstacles around which the page was divided. If on the other hand there are obstacles, we need to further subdivide the bound. To this end, we choose a *pivot*, which ideally is centered somewhere around the middle of the bound.

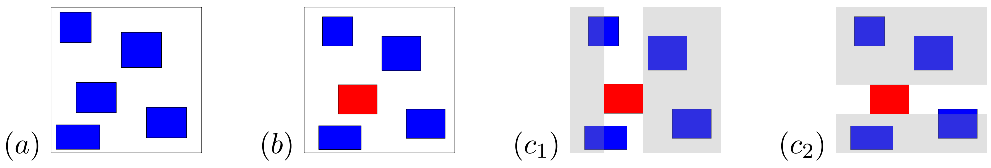


Figure 3.1: Example showing one iteration of the whitespace covering algorithm. In (a) we see some obstacles contained within a bounding rectangle. In (b) one of them is chosen as as pivot (marked in red), and (c_1) and (c_2) show how the original bound is divided into four smaller rectangles (marked in grey) around the pivot.

We know that any maximal rectangle can not contain any obstacles, in particular not the pivot. Based on that there are four possibilities for the solution of the maximal whitespace rectangle problem, one for each side of the pivot. The areas of these four subbounds are computed, a list of intersecting obstacles is computed for each of them, and they are all put back on the queue. The selection of a pivot is seen in (b) in Figure 3.1, while the four resulting rectangles are marked in (c_1) and (c_2). The algorithm can be seen in pseudocode in Figure 3.2.

Data structure The algorithm does not really require a specialized geometrical data structure. The main data structure in use is a queue of `QueueEntry` objects which each represent the bound of a subset of a page, and everything contained within it. The definition can be seen in Figure 3.3. The queue is a priority queue which always will return the `QueueEntry` with the highest priority, as determined by a quality function $Q(r)$.

Choosing a quality function To decide which rectangles we want as output, we use a quality function $Q(r)$ for a rectangle bound r, which in this project is defined as follows

```
Input: Geometrical bound of page pageBound
       A list of obstacles obstacles
Output: The next identified whitespace rectangle

queue ⟵ create a queue which sorts QueueEntries based on their Q()
add a new QueueEntry containing obstacles and bounded by pageBound to queue

while queue is not empty do
    current ⟵ QueueEntry with highest Q() from queue

    if isEmpty (current →obstacles) then
        output current
        return if sufficient rectangles has been found, if not continue with next
        QueueEntry
    end

    pivot ⟵ select the centermost obstacle from current →obstacles

    b ⟵ current→bound
    $r_{right}$ →bound ⟵ (pivot →endX, b →y, b →endX, b →endY)
    $r_{left}$  →bound ⟵ (b →x, b →y, pivot →x, b →endY)
    $r_{above}$ →bound ⟵ (b →x, b →y, b →endX, pivot →y)
    $r_{below}$ →bound ⟵ (b →x, pivot →endY, b →endX, b →endY)
    subrectangles ⟵ $(r_{right}, r_{left}, r_{above}, r_{below})$

    foreach r in subrectangles do
        r →obstacles ⟵ select all of current →obstacles intersecting with r
        →bound
        add r to queue
    end

end
```

Figure 3.2: This is the core algorithm more or less as described in the paper. It is called repeatedly until the requested number of whitespace rectangles has been found. For the next run, the returned whitespace rectangle will be added to the provided list of obstacles

```
class QueueEntry {
    const Rectangle bound
    const float quality ← Q(bound)
    List<Rectangle> obstacles
    int numberOfWhitespaceFound
}
```

Figure 3.3: The pseudocode definition of the `QueueEntry` class. It contains a rectangle bound which corresponds to part of a page, the obstacles contained within it, and a precalculated score as determined by the scoring function $Q(r)$ which is subsequently used to sort a priority queue.

:

$$Q(r) = \text{area}(r) \cdot \frac{\text{height}(r)}{4}$$

As can be seen, tall rectangles are preferred. The trick while choosing this $Q(r)$ was to keep that preference while still allowing wide rectangles to be chosen. After having experimented with quite a few variations, this simple function was considered a good solution.

Finding *n*–best solutions (a greedy version) The algorithm as presented above returns one maximal rectangle; in order to find more it is would be possible to continue to expand nodes from the queue. However, this way of obtaining many rectangles would produce rectangles with substantial overlap. In order to avoid that, a greedy version of the algorithm, which was proposed in the original paper, was used.

After a maximal rectangle has been found, it is added back to the list of obstacles. Whenever a `QueueEntry` is dequeued, its list of obstacles can be recomputed to include newly identified whitespace rectangles. This is the purpose of the field `numberOfWhitespaceFound` in the definition of `QueueEntry`, which on dequeueing is checked against the current count.

3.2.3 Problems and adaptations

Although the algorithm worked like it was supposed to, there were some problems with it for the use envisioned. The most important problem was that the algorithm was not *smart enough*. It would leave connecting gaps between non-related groups of text, and it would separate related text into several groups. Needless to say, tweaking $Q(r)$ and setting constant minimum bounds was of no use in the general case, so a few adaptations to the core algorithm were needed.

Secondly, the algorithm as initially implemented was too slow. While running (single threadedly) on a semi-modern Intel *Core 2 Duo*, worst case performance for analysing *one page* was around 20 seconds, with averages closer to 3-4 seconds.

The following factors were important for the performance of the algorithm:

1. The number of rectangles required to adequately cover the page
2. The cost of computing one rectangle
3. The number of needlessly created rectangles (*i.e.* those which did not end up in the output)
4. Implementation deficiencies and missing optimisations

It was thus clear that most, or preferably all, of these should be reduced to a minimum.

3.2.4 Avoiding unwanted passages by overlapping

The most visible problem was that rectangles would frequently *almost* touch each other, while letting small narrow passages remain inbetween. An example of this problem can be seen to the left in Figure 3.4. This would complicate grouping later on, as it would be necessary to then have some kind of limit of how narrow a connection between two pieces of text could be. As this problem would essentially be just a variation of the original problem faced, it was highly undesirable.

The main culprit here seemed to be uneven shapes of text groups, where a small amount of content standing out could negatively affect the output as described. Although this effect may to a certain degree be implicated by how the algorithm works, some simple adaptations helped remedy the problem to a large degree.

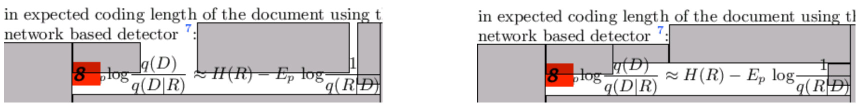
The first and most obvious adaption, which was also suggested in the original article, was to allow a candidate whitespace rectangle to overlap the surrounding obstacles by some percent. After considering a few different ways to enable this functionality, the following function was chosen for deciding whether any two rectangles overlap. WHITESPACE_FUZZINESS is a constant which in the finished code was set to 15%, which was thought to be a reasonable overlap and found to work well.

$$\text{overlaps}(r_1, r_2) = \text{area}(r_1 \cap r_2) > \min_{\text{area}(r_1),\ \text{area}(r_2)} \cdot \text{WHITESPACE\_FUZZINESS}$$

Having defined that, we are left with the decision of whether to consider a potential whitespace rectangle ρ as being empty, given the set of obstacles O in a page.

$$\begin{aligned}\text{isEmptyEnough}(\rho) = \nexists\, o \in O : \text{overlaps}(\rho, o) \\ \wedge \sum_{o \in O} \text{area}(\rho \cap o) < \text{area}(\rho) \cdot \text{ WHITESPACE\_FUZZINESS}\end{aligned}$$

This also helped in that the generation of quite a few unnecessary rectangle candidates was avoided, and it helped ensure a more smooth execution. The effect of this adaption can be seen to the right in Figure 3.4.


in expected coding length of the document using t
network based detector [7]:
8 $_p\log\frac{q(D)}{q(D|R)} \approx H(R) - E_p \log\frac{1}{q(R|D)}$

in expected coding length of the document using tl
network based detector [7]:
8 $_p\log\frac{q(D)}{q(D|R)} \approx H(R) - E_p \log\frac{1}{q(R|D)}$

Figure 3.4: Before (left) and after (right) the fix for unwanted passages.

3.2.5 Avoiding stray whitespace by continuity

Another problem were rectangles which were identified in the middle of text groups. While normally not a big problem, it could erroneously break up said text groups at a later stage. An example of such rectangles can be seen in Figure 3.5.

A check was added which impeded the algorithm from accepting rectangles which were not adjacent to an already accepted rectangle, or to the border of the page. In order to maintain the correctness of the algorithm, rejected rectangles are put in a *hold list*. Each time a new rectangle is identified and accepted, this hold list will be added back to the queue in case any of them will have become valid.

This had the effect that the algorithm started from the borders of the page and worked itself inwards, thus more naturally working its way through groups of text. It also avoided stray rectangles located inside paragraphs.

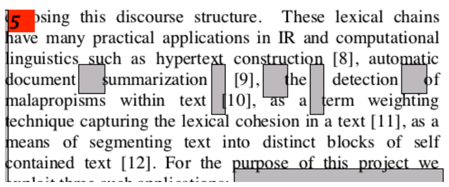

5 osing this discourse structure. These lexical chains have many practical applications in IR and computational linguistics such as hypertext construction [8], automatic document summarization [9], the detection of malapropisms within text [10], as a term weighting technique capturing the lexical cohesion in a text [11], as a means of segmenting text into distinct blocks of self contained text [12]. For the purpose of this project we

Figure 3.5: Whitespace rectangles were frequently misplaced inbetween groups of text

3.2.6 Avoiding intraparagraph lines by local minimum bounds

With a $Q(r)$ which favoured long or tall rectangles for their separational qualities, it would frequently happen that lines in some paragraphs would have whitespace rectangles inbetween all their lines. While this would have been possible to remedy later, it was still also undesirable.

This was solved by introducing lower bounds for the height and width of rectangles. As an optimization, conservative page-wide bounds were precalculated based on average font sizes (for vertical bounding) and average vertical distance between characters (for horizontal bounding). For every identified rectangle, a more precise estimate of minimum bounds is calculated based on the immediate surrounding text (if any), and the rectangle will be accepted based on that. The effect of this fix can be seen in Figure 3.6.

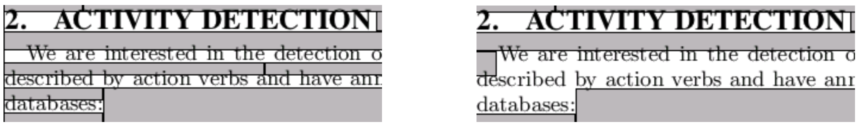

2. ACTIVITY DETECTION
We are interested in the detection o
described by action verbs and have anr
databases:
2. ACTIVITY DETECTION
We are interested in the detection o
described by action verbs and have anr
databases:

Figure 3.6: Before (left) and after (right) the fix.

3.2.7 Avoiding two word separators

Rectangles would frequently sneak in into groups of text via rivers or extra spacing. Although not a big problem, this had the effect of frequently breaking up short paragraphs of 1-4 lines. An example of this can be seen in Figure 3.7

This was solved by imposing a rule stating that any rectangle located between one text element both to the left and to the right will not be accepted.

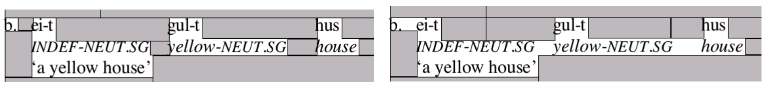

b. ei-t gul-t hus
INDEF-NEUT.SG yellow-NEUT.SG house
'a yellow house'
b. ei-t gul-t hus
INDEF-NEUT.SG yellow-NEUT.SG house
'a yellow house'

Figure 3.7: Before (left) and after (right) the fix. Also take note how big enough spaces are still filled, by looking at the spacing separating the words in "Eit gul-t hus"

3.2.8 Misc adaptions

Quite a few optimizations had to be done in order to get the algorithm working fast enough.

- The bounds within which the algorithm works are the minimum bound which envelops all the content on the page, not the dimensions of the page. This decreases the amount of rectangles needed to cover the page
- A maximum bound for queue size was applied. It is sufficiently large that enough data will have been found once it is met
- The Java profiler JProfiler[1] was used to identify local bottlenecks. The code was thus optimized to perform a minimum of calculations. As importantly, the originally object-oriented design of the relevant classes had to be changed a bit to enable static invocation and hence inlining of functions; although that may seem overly technical, it effectively halfed execution time

3.2.9 Discussion

It was mentioned initially that the performance of the algorithm depends on a variety of variables: page layout, whitespace rectangles needed, *etc.*

While the amount of whitespace rectangles needed is an important consideration, it is also crucial to note a few things about *what kind* of rectangles are wanted as well. In Breuel (2002), 200 rectangles are extracted from each page, with a pairwise overlap of up to 80%, and no overlap with textual content. Here, in comparison, a whitespace rectangle can overlap both other whitespace rectangles and text by the rules presented in Section 3.2.4 (up to 15% pairwise and combined overlap).

The algorithm changes performance characteristics a bit by the changes presented above. The overlapping rule and the page-wide bounds mentioned in Section 3.2.6 together drastically cut down on the number of generated potential rectangles. On the other hand, the rest of the adaptions can be seen as filtering, and thus potentially throw away many rectangles before one is accepted. This increases the cost again, but adds a lot of value by outputting more useful rectangles.

More useful rectangles do here translate directly into a drastically reduced number needed when compared to the original algorithm. The net result is that to completely cover a page with advanced layout, 30 rectangles are almost always enough. A maximum of 40 rectangles was eventually chosen to cover the odd case where it would be beneficial with more.

The advantage of the added information provided by a PDF as opposed to a document scan became apparent while implementing this algorithm. Permitting whitespace rectangles to partly overlap with text had a very good effect. That would have been very

[1]http://www.ej-technologies.com/products/jprofiler

difficult to do in OCR because it would have relied on information which is typically not yet available. Likewise for the filtering process, which is in part powered by having access to mostly exactly positioned content and font sizes for approximate boundaries.

In terms of CPU time, the algorithm is as mentioned expensive, though much less so than the initial implementation. Once Java's *Just in Time* compiler has warmed up, a normal page is fully analysed in 100 ms on the same computer. The upper bound seems to be around 1 second for finding these 40 rectangles. A graph which shows run times for computing 0–50 rectangles for three pages of varying complexity is shown in Figure 3.8. To illustrate the progression of the algorithm on a real page, see Figure 3.9.

Since this algorithm ended up being at the core of the project, and also was the most researched part of the project, it was decided to include the implementation code in Appendix B.

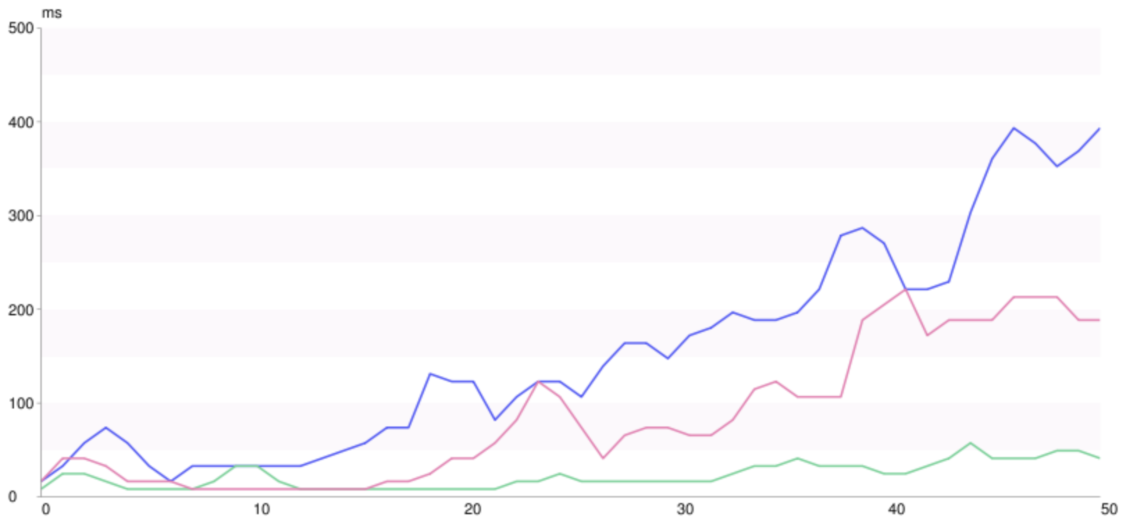


Figure 3.8: Number of whitespace rectangles identified plotted against how long it took for three different pages, one simple and two complicated (The one with the longest runtime can be seen in Figure 3.9) Even for 50 rectangles 400 ms is enough, while simple pages will in any case be analysed in well under 100 ms. These numbers come from effectively running the program 50 times for each page with one smoothing pass for the runtimes.

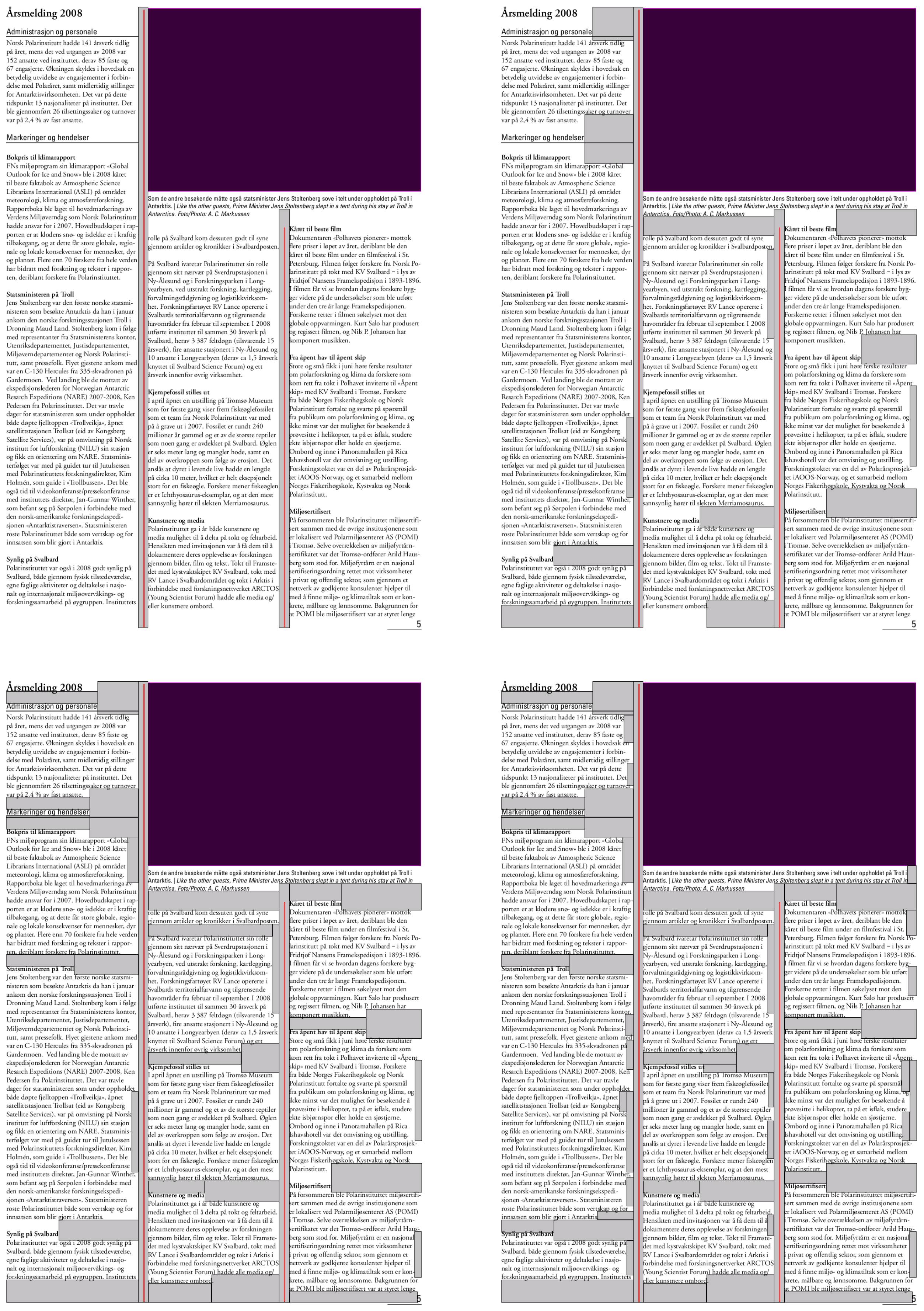


Figure 3.9: This demonstrates the progression after 2, 15, 30 and 50 identified rectangles. We see that enough information to recover columnal structure is available after only two rectangles (as there are two column boundaries). If text grouping is to rely on this information alone, 30 rectangles would not sufficient for this page, as can be seen by *e.g.* the gaps left behind above section headers

3.3 Determining page layout

The next step according to Breuel (2002) is to use the identified whitespace rectangles to determine the page layout in terms of columns. As mentioned in Section 3.2.1, the whitespace analysis is expected to provide functional whitespace, which can be used for the analysis; this is its first application.

3.3.1 Challenges

It should be noted that while the whitespace rectangles enable this functionality, there are no guarantees that there has been identified *one* rectangle which is equivalent to a whole column boundary. This is a natural consequence of how the algorithm works, at least with a $Q(r)$ which admits reasonably sized horizontal rectangles. Such a horizontal rectangle might be scored higher than a potential vertical rectangle which would have covered a column perfectly, and thus block it.

In addition there can also be imperfections like gaps in the identified whitespace if the page in question contains some content which breaks with the general layout. An example of this can be seen to the left in Figure 3.10. It can also happen that the whitespace cover algorithm erroneously outputs a set of rectangles which will leave unrelated pieces of text connected. This can be seen at the right in the same figure.

In the original design by Breuel there was an assumption that such a guarantee was indeed given, and that all that was needed was to select whitespace rectangles as column separators based on a list of criteria. Because of the 80% permitted overlap among the whitespace rectangles, column spacing would seldom be blocked, but the imperfections would in any case remain, still making column adjustments necessary.

Taking these complications into account, our revised process for determining column boundaries consists of three consecutive steps:

1. Extract a set of candidate column boundaries from the set of identified whitespace rectangles
2. Vertically expand column boundary candidates
3. Combine equivalent and filter out unwanted boundaries.

3.3.2 Extract column boundary candidates

The objective here is to select whitespace rectangles which are likely to correspond to at least part of a real column boundary. In order to do this, there must necessarily exist some criteria which a potential candidate must meet. Because we will introduce

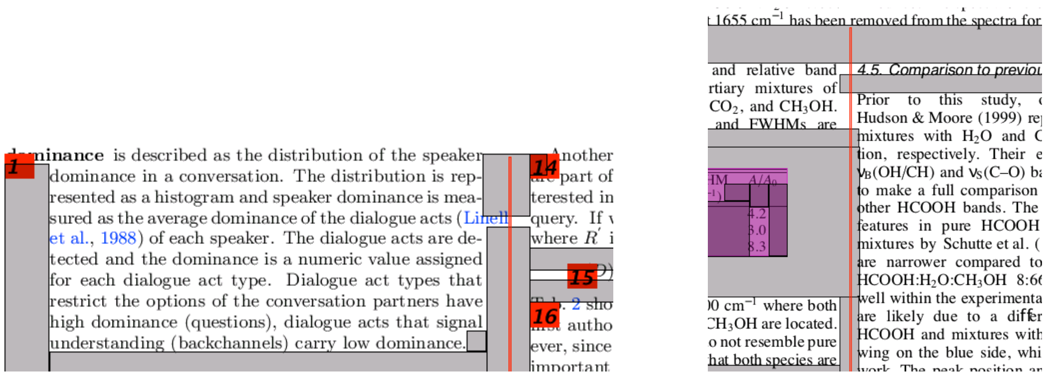


Figure 3.10: Left: The name “Linell” breaks out from the layout. In order to obtain the column boundary (marked in red), it is necessary to combine the two whitespaces located above and below.
Right: The column boundary has to be interpolated based on the whitespace rectangle at the bottom (starting roughly between “both” and “well”)

a combination and filtering phase at a later stage, it is appropriate to have quite loose requirements here. These are summarised in the following paragraphs.

Aspect ratio Most importantly, the whitespace rectangle must be taller than it is wide. In the original paper the minimum aspect ratio was 1 : 3. It was observed that in some situations that would allow some columns to escape undiscovered, so for this project this demand was relaxed to 2 : 3.

Surrounding content Additionally, there must be some bound on how much text is required to surround the whitespace rectangle. This candidate selection essentially just checks that there is indeed content on both the right and left sides. Even one side with no content can be tolerated if there is a lot of content on the other; this solves a very specific problem where small floaters located between page boundaries and a column of text would not be separated from the column.

The pseudocode in Figure 3.11 illustrates how the selection is done.

3.3.3 Height adjustment of columns

The real column boundaries on a page can, as seen in Figure 3.10, extend beyond individual whitespaces. To approximate these, it is necessary to try to adjust them. This is done by finding a vertical range extending from that of the boundary column candidate

```
Input: Set of identified whitespaces $W$
Output: Set of column boundary candidates $B$
foreach Rectangle $w$ in $W$ do
    Only accept with a certain aspect ratio;
    if $w.height/w.width < 1.5$ then
        reject $w$;
    end
    leftCount $\leftarrow$ locate count of text fragments which ends just left of $w$;
    rcCount $\leftarrow$ locate count of text fragments which starts just right of $w$;
    Rule out whitespace with no content on either side, unless there is a lot on the
    other;
    if leftCount $= 0$ and rcCount $< 8$ then
        reject $w$;
    end
    if rcCount $= 0$ and leftCount $< 8$ then
        reject $w$;
    end
    Finally, There must be 4 or more lines of content on either side;
    if leftCount $\geq 4$ or rcCount $\geq 4$ then
        add $w$ to $B$;
    end
end
```

Figure 3.11: Algorithm for extracting column boundary candidates. The constants which appear in the code to signify *a lot* are a bit random, but were found to work. This was essentially a makeshift solution which there was never time to replace with something more clever.

while assuring that there is content on the right and that the range is not blocked by any contents. The algorithm appears in Figure 3.12.

```
Input: Set of column boundary candidates $B$, a geometric data structure with
       the content of the current page $P$
Output: Set of adjusted column boundary candidates $A$
foreach Rectangle $b$ in $B$ do
    Rectangle[] splits = split $b$ into three thin vertical columns, each centered
    around the left, middle and right $X$-coordinates;
    foreach Rectangle $s$ in splits do
        Iterate through the range of $Y$ values and determine the real length of $s$:;
        $x \leftarrow s.minX$ ;
        for $y \leftarrow P.minY$ to $P.maxY$ do
            contentToRight $\leftarrow$ exists( content right of $(x, y)$);
            blocked $\leftarrow$ exists( content intersecting $(x, y)$) or exists(content
            adjacent to $(x, y)$ on both sides);
        end
        startY $\leftarrow$ the first $y$–coordinate right of which there is text and below
        which there is a nonblocked path to $b$;
        endY $\leftarrow$ the last $y$–coordinate right of which there was text and above
        which there was a nonblocked path to $b$.;
        $s \leftarrow$ updated position with vertical coordinates from startY to endY;
    end
    add maxHeight(splits) to $A$
end
```

Figure 3.12: Algorithm for adjusting column boundary candidates

3.3.4 Combination and filtering of column boundaries

Frequently the two introductory phases will leave us with several column boundary candidates which effectively represent the same real boundary. While this is not critical, it is easy to combine them. This is done by sorting the column boundary candidates on their X–coordinate, and then combining pairs of them when there is no content inbetween them. There is also a lower bound on column height, both because there tended to be many falsely identified columns of short length, and because very short columns are insignificant layout-wise since they are generally correctly grouped and ordered anyway.

The resulting set of column boundaries together form the high level layout which will subsequently be used for text grouping.

3.4 Text grouping and segmentation

3.4.1 Page division – Column segmentation

Dividing a page into subparts when given the identified column boundaries is conceptually very simple. In practice this is done in this project by starting from the right hand side of the page, and then iteratively working leftwards boundary by boundary while extracting the content remaining right of each boundary. All text deemed to fall within a graphical figure is also extracted into special graphical elements.

Internally this is supported by a hierarchical, geometrical data structure in which blocks of extracted content are placed under the content from which it was separated. This is mainly done to facilitate finding the correct order of text later.

3.4.2 Block segmentation

Traditionally, many tools have enveloped content in rectangular blocks, as it is both conceptually and computationally the simplest solution. One of the advantages we get for free by using the whitespace covering algorithm is that we end up with implicit blocks formed after the shape of the content contained within the "holes" left behind by the whitespace rectangles.

Calculating blocks of variable shapes bounded by a set of rectangles was not entirely straightforward, but was solved with a flooding, recursive function which for every unassigned content locates the nearest connected (*i.e.* with a path not blocked by any whitespace rectangle) neighbour in all directions, marks those as assigned to the same block, and then restarts the procedure from each neighbour.

There is an exception for handling equations, where if what probably equals a line of text is deemed (by its content and/or font) to be an equation, all whitespace separating content horizontally is ignored. This works around a problem where equations are erroneously split up.

3.4.3 Line segmentation

Line segmentation is done blockwise. All content within a block is sorted on Y–coordinates, and the algorithm starts from the top and works itself downwards, marking content as belonging to the current line as it goes. For every Y–coordinate, an estimate of how many pixels are covered by content is made, and a new line is created for local minima beyond a certain distance from the last newline.

3.5 Recovering reading order

The step for determining reading order is described in Breuel (2002) as “4. Determine reading order using both geometric and linguistic information”. In this project there are no means to use linguistic information to determine this, so we are stuck with geometric information. In Breuel (2003), the same author outlines another method of ordering text. In section 3 we are shown how topological sorting of text lines (not larger text blocks) seems to do the job reasonably well. In section 4 a probabilistic extension is suggested, which would be consulted when a document is ambigous at the layout level.

The probabilistic extension seemed both untested and time-consuming to implement, so it was decided to go with the topological sort, and see how it would work and whether it could be extended if it was not good enough.

The comparison function in use receives two blocks o_1 and o_2, and then decides which of them should be sorted first. See Figure 3.13 for details.

```java
/**
  * Sorts two blocks, o1 and o2. if -1 is returned, that means o1 sorts before o2.
  */
public int compare(@NotNull final HasPosition o1, @NotNull final HasPosition o2) {

    /** If one block is located entirely above another, it goes before. */
    if (o1.getPos().endY < o2.getPos().y) {
        return -1;
    }

    if (o1.getPos().y > o2.getPos().endY) {
        return 1;
    }

    /** If one block is entirely to the left, it goes before. */
    if (o1.getPos().endX < o2.getPos().x) {
        return -1;
    }

    if (o1.getPos().x > o2.getPos().endX) {
        return 1;
    }

    /** if the two blocks are located at more or less the same Y-coordinate, the one to the
        left goes before. If not, else the one which starts higher up is sorted first */
    if (!MathUtils.isWithinPercent(o1.getPos().y, o2.getPos().y, 4)) {
        return Float.compare(o1.getPos().y, o2.getPos().y);
    }

    return Float.compare(o1.getPos().x, o2.getPos().x);
}
```

Figure 3.13: Topological sorting function: two and two blocks are sorted relative to each other.

The sorting function used in Breuel (2003) was not really described, but this implementation should follow the general gist of it. As can be seen from the sorting function, it is not meant to correctly sort a flat page full of blocks of content, but depends on an internal data structure briefly mentioned in Section 3.4.1. The trick here is that columns will be divided into *regions*, which then permits nesting of further columns into subregions. To see how this works in practice look at Figure 3.14. There are four top level regions here (labelled in blue), namely the ones containing paragraphs numbered 1–7, 8–19, 20–22 and 23–26. The regions with numbers 15–19 and 25–27 are nested within 8–19 and 23–27, respectively.

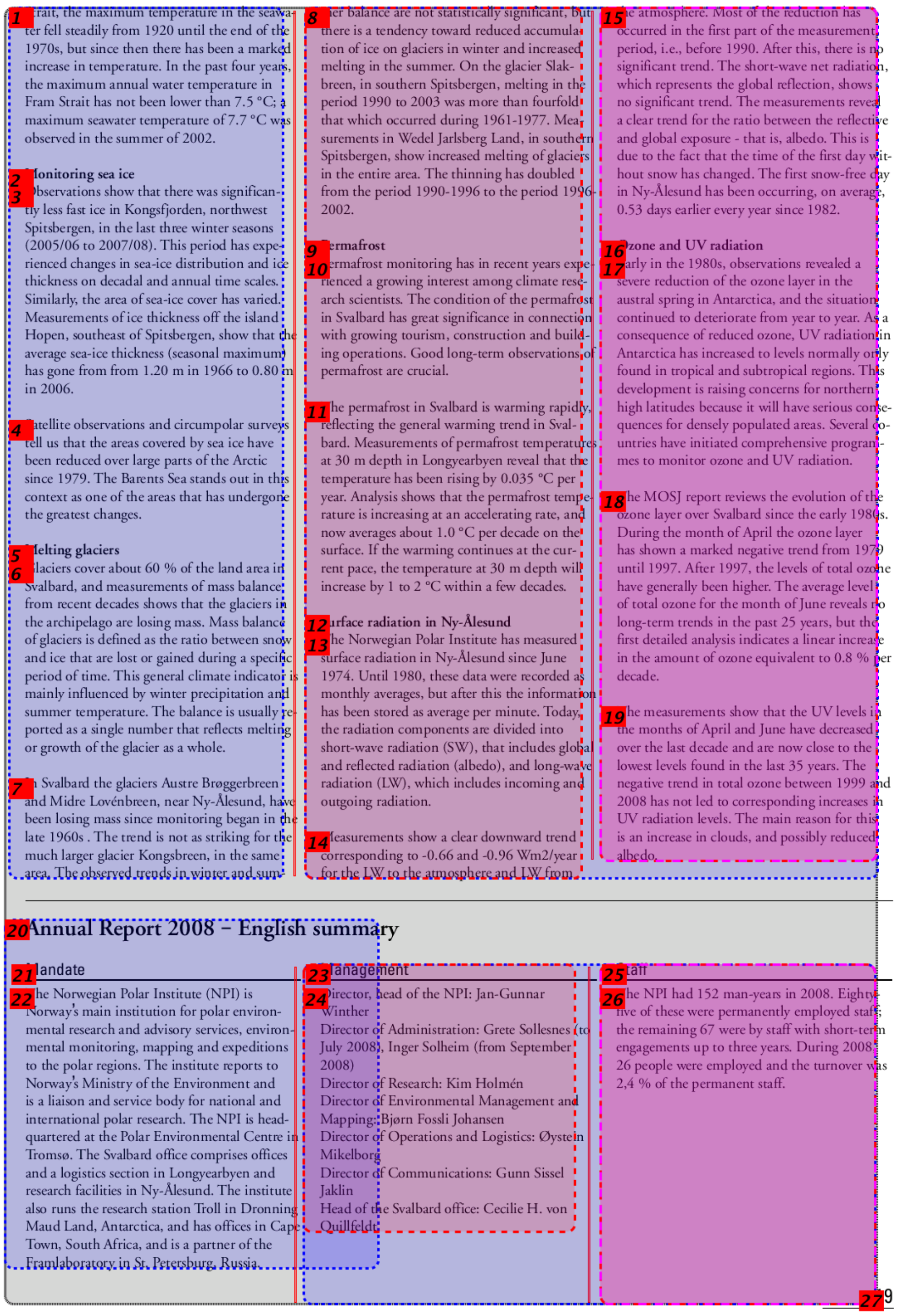
While there is no doubt that there will be some layouts for which this does not work perfectly, it seems very promising and has successfully sorted content from a large range of layouts.

3.6 State after Geometric Layout Analysis

To conclude the chapter on geometric layout analysis, it can be useful to have a look at all the information we are left with after the geometric layout analysis. In Figure 3.15 we can see a lot of things featured. The timeline at the bottom and the box with the heading "TULIPMANIA" are both categorized as graphics. The text inside is extracted along with the graphics itself, and hence receive no paragraph numbers. Inside the tulipmania box we can see how the whitespace covering algorithm is recursively applied, in case there would be columns inside the box.

To the left, marked with number 2–4 are floating sidetexts which has been separated out as a column of itself. There is top text present as well, which has been segmented together into two blocks, one for each page. Separating these blocks of text out from the body text will be performed as a part of the logical layout analysis.

We also see nested regions in play, as well as correctly ordered text blocks.



rait, the maximum temperature in the seawater fell steadily from 1920 until the end of the 1970s, but since then there has been a marked increase in temperature. In the past four years, the maximum annual water temperature in Fram Strait has not been lower than 7.5 °C; a maximum seawater temperature of 7.7 °C was observed in the summer of 2002.

lonitoring sea ice

bservations show that there was significantly less fast ice in Kongsfjorden, northwest Spitsbergen, in the last three winter seasons (2005/06 to 2007/08). This period has experienced changes in sea-ice distribution and ice thickness on decadal and annual time scales. Similarly, the area of sea-ice cover has varied. Measurements of ice thickness off the island Hopen, southeast of Spitsbergen, show that the average sea-ice thickness (seasonal maximum) has gone from from 1.20 m in 1966 to 0.80 m in 2006.

atellite observations and circumpolar surveys tell us that the areas covered by sea ice have been reduced over large parts of the Arctic since 1979. The Barents Sea stands out in this context as one of the areas that has undergone the greatest changes.

lelting glaciers

laciers cover about 60 % of the land area in Svalbard, and measurements of mass balance from recent decades shows that the glaciers in the archipelago are losing mass. Mass balance of glaciers is defined as the ratio between snow and ice that are lost or gained during a specific period of time. This general climate indicator is mainly influenced by winter precipitation and summer temperature. The balance is usually reported as a single number that reflects melting or growth of the glacier as a whole.

n Svalbard the glaciers Austre Brøggerbreen and Midre Lovénbreen, near Ny-Ålesund, have been losing mass since monitoring began in the late 1960s . The trend is not as striking for the much larger glacier Kongsbreen, in the same area. The observed trends in winter and sum-

ner balance are not statistically significant, but there is a tendency toward reduced accumulation of ice on glaciers in winter and increased melting in the summer. On the glacier Slakbreen, in southern Spitsbergen, melting in the period 1990 to 2003 was more than fourfold that which occurred during 1961-1977. Measurements in Wedel Jarlsberg Land, in southern Spitsbergen, show increased melting of glaciers in the entire area. The thinning has doubled from the period 1990-1996 to the period 1996-2002.

ermafrost

ermafrost monitoring has in recent years experienced a growing interest among climate research scientists. The condition of the permafrost in Svalbard has great significance in connection with growing tourism, construction and building operations. Good long-term observations of permafrost are crucial.

he permafrost in Svalbard is warming rapidly, reflecting the general warming trend in Svalbard. Measurements of permafrost temperatures at 30 m depth in Longyearbyen reveal that the temperature has been rising by 0.035 °C per year. Analysis shows that the permafrost temperature is increasing at an accelerating rate, and now averages about 1.0 °C per decade on the surface. If the warming continues at the current pace, the temperature at 30 m depth will increase by 1 to 2 °C within a few decades.

urface radiation in Ny-Ålesund

he Norwegian Polar Institute has measured surface radiation in Ny-Ålesund since June 1974. Until 1980, these data were recorded as monthly averages, but after this the information has been stored as average per minute. Today, the radiation components are divided into short-wave radiation (SW), that includes global and reflected radiation (albedo), and long-wave radiation (LW), which includes incoming and outgoing radiation.

leasurements show a clear downward trend corresponding to -0.66 and -0.96 Wm2/year for the LW to the atmosphere and LW from

he atmosphere. Most of the reduction has occurred in the first part of the measurement period, i.e., before 1990. After this, there is no significant trend. The short-wave net radiation, which represents the global reflection, shows no significant trend. The measurements reveal a clear trend for the ratio between the reflective and global exposure - that is, albedo. This is due to the fact that the time of the first day without snow has changed. The first snow-free day in Ny-Ålesund has been occurring, on average, 0.53 days earlier every year since 1982.

zone and UV radiation

arly in the 1980s, observations revealed a severe reduction of the ozone layer in the austral spring in Antarctica, and the situation continued to deteriorate from year to year. As a consequence of reduced ozone, UV radiation in Antarctica has increased to levels normally only found in tropical and subtropical regions. This development is raising concerns for northern high latitudes because it will have serious consequences for densely populated areas. Several countries have initiated comprehensive programmes to monitor ozone and UV radiation.

he MOSJ report reviews the evolution of the ozone layer over Svalbard since the early 1980s. During the month of April the ozone layer has shown a marked negative trend from 1979 until 1997. After 1997, the levels of total ozone have generally been higher. The average level of total ozone for the month of June reveals no long-term trends in the past 25 years, but the first detailed analysis indicates a linear increase in the amount of ozone equivalent to 0.8 % per decade.

he measurements show that the UV levels in the months of April and June have decreased over the last decade and are now close to the lowest levels found in the last 35 years. The negative trend in total ozone between 1999 and 2008 has not led to corresponding increases in UV radiation levels. The main reason for this is an increase in clouds, and possibly reduced albedo.

Annual Report 2008 – English summary

landate

he Norwegian Polar Institute (NPI) is Norway's main institution for polar environmental research and advisory services, environmental monitoring, mapping and expeditions to the polar regions. The institute reports to Norway's Ministry of the Environment and is a liaison and service body for national and international polar research. The NPI is headquartered at the Polar Environmental Centre in Tromsø. The Svalbard office comprises offices and a logistics section in Longyearbyen and research facilities in Ny-Ålesund. The institute also runs the research station Troll in Dronning Maud Land, Antarctica, and has offices in Cape Town, South Africa, and is a partner of the Framlaboratory in St. Petersburg, Russia.

lanagement

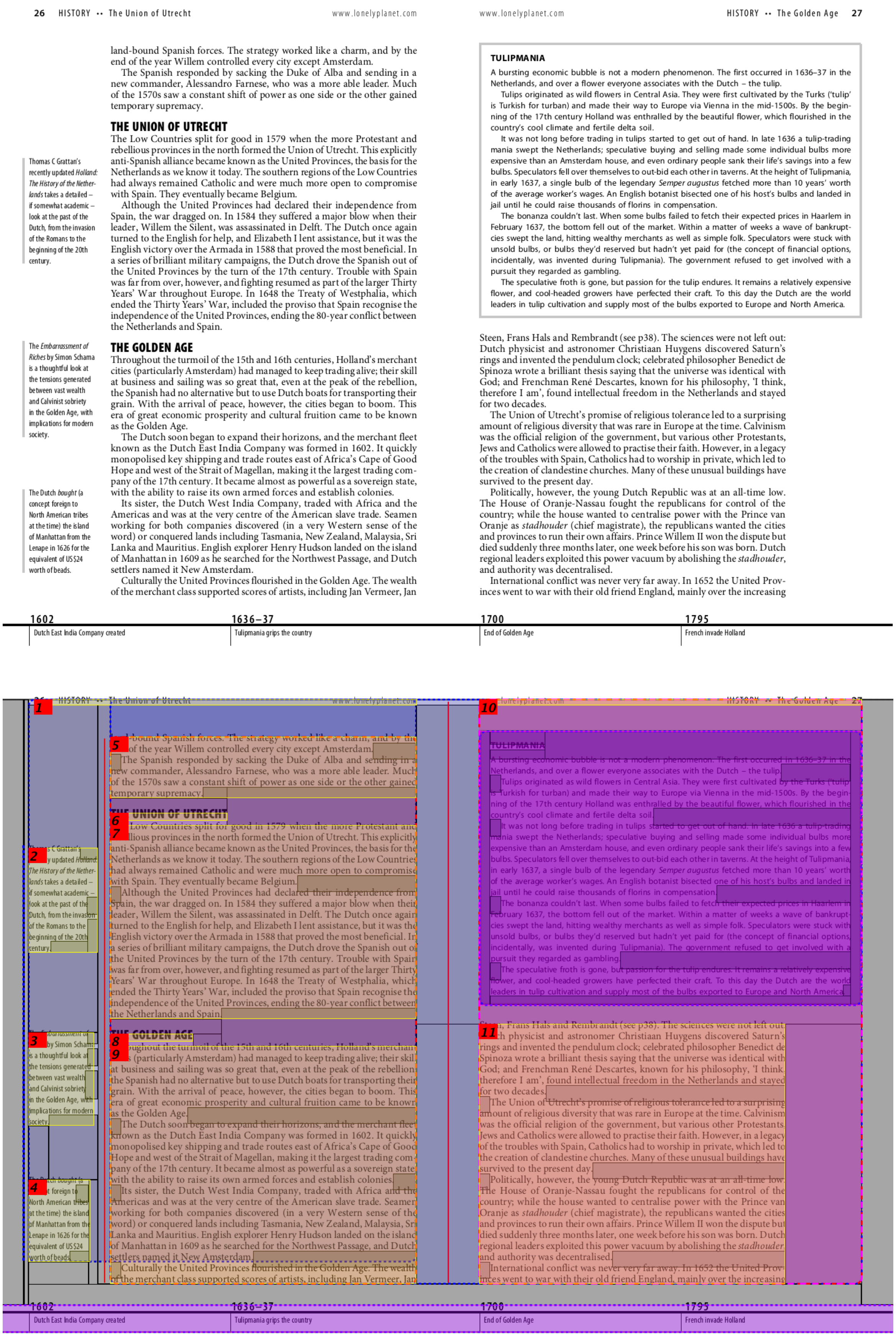
irector, head of the NPI: Jan-Gunnar Winther
Director of Administration: Grete Sollesnes (to July 2008), Inger Solheim (from September 2008)
Director of Research: Kim Holmén
Director of Environmental Management and Mapping: Bjørn Fossli Johansen
Director of Operations and Logistics: Øystein Mikelborg
Director of Communications: Gunn Sissel Jaklin
Head of the Svalbard office: Cecilie H. von Quillfeldt

taff

he NPI had 152 man-years in 2008. Eighty-five of these were permanently employed staff; the remaining 67 were by staff with short-term engagements up to three years. During 2008, 26 people were employed and the turnover was 2,4 % of the permanent staff.

9

Figure 3.14: Reading order: nested page regions.


26 HISTORY ·· The Union of Utrecht www.lonelyplanet.com

land-bound Spanish forces. The strategy worked like a charm, and by the end of the year Willem controlled every city except Amsterdam.

The Spanish responded by sacking the Duke of Alba and sending in a new commander, Alessandro Farnese, who was a more able leader. Much of the 1570s saw a constant shift of power as one side or the other gained temporary supremacy.

THE UNION OF UTRECHT

The Low Countries split for good in 1579 when the more Protestant and rebellious provinces in the north formed the Union of Utrecht. This explicitly anti-Spanish alliance became known as the United Provinces, the basis for the Netherlands as we know it today. The southern regions of the Low Countries had always remained Catholic and were much more open to compromise with Spain. They eventually became Belgium.

Although the United Provinces had declared their independence from Spain, the war dragged on. In 1584 they suffered a major blow when their leader, Willem the Silent, was assassinated in Delft. The Dutch once again turned to the English for help, and Elizabeth I lent assistance, but it was the English victory over the Armada in 1588 that proved the most beneficial. In a series of brilliant military campaigns, the Dutch drove the Spanish out of the United Provinces by the turn of the 17th century. Trouble with Spain was far from over, however, and fighting resumed as part of the larger Thirty Years' War throughout Europe. In 1648 the Treaty of Westphalia, which ended the Thirty Years' War, included the proviso that Spain recognise the independence of the United Provinces, ending the 80-year conflict between the Netherlands and Spain.

Thomas C Grattan's recently updated *Holland: The History of the Netherlands* takes a detailed – if somewhat academic – look at the past of the Dutch, from the invasion of the Romans to the beginning of the 20th century.

THE GOLDEN AGE

Throughout the turmoil of the 15th and 16th centuries, Holland's merchant cities (particularly Amsterdam) had managed to keep trading alive; their skill at business and sailing was so great that, even at the peak of the rebellion, the Spanish had no alternative but to use Dutch boats for transporting their grain. With the arrival of peace, however, the cities began to boom. This era of great economic prosperity and cultural fruition came to be known as the Golden Age.

The Dutch soon began to expand their horizons, and the merchant fleet known as the Dutch East India Company was formed in 1602. It quickly monopolised key shipping and trade routes east of Africa's Cape of Good Hope and west of the Strait of Magellan, making it the largest trading company of the 17th century. It became almost as powerful as a sovereign state, with the ability to raise its own armed forces and establish colonies.

Its sister, the Dutch West India Company, traded with Africa and the Americas and was at the very centre of the American slave trade. Seamen working for both companies discovered (in a very Western sense of the word) or conquered lands including Tasmania, New Zealand, Malaysia, Sri Lanka and Mauritius. English explorer Henry Hudson landed on the island of Manhattan in 1609 as he searched for the Northwest Passage, and Dutch settlers named it New Amsterdam.

Culturally the United Provinces flourished in the Golden Age. The wealth of the merchant class supported scores of artists, including Jan Vermeer, Jan

The *Embarrassment of Riches* by Simon Schama is a thoughtful look at the tensions generated between vast wealth and Calvinist sobriety in the Golden Age, with implications for modern society.

The Dutch *bought* (a concept foreign to North American tribes at the time) the island of Manhattan from the Lenape in 1626 for the equivalent of US$24 worth of beads.

www.lonelyplanet.com HISTORY ·· The Golden Age 27

TULIPMANIA

A bursting economic bubble is not a modern phenomenon. The first occurred in 1636–37 in the Netherlands, and over a flower everyone associates with the Dutch – the tulip.

Tulips originated as wild flowers in Central Asia. They were first cultivated by the Turks ('tulip' is Turkish for turban) and made their way to Europe via Vienna in the mid-1500s. By the beginning of the 17th century Holland was enthralled by the beautiful flower, which flourished in the country's cool climate and fertile delta soil.

It was not long before trading in tulips started to get out of hand. In late 1636 a tulip-trading mania swept the Netherlands; speculative buying and selling made some individual bulbs more expensive than an Amsterdam house, and even ordinary people sank their life's savings into a few bulbs. Speculators fell over themselves to out-bid each other in taverns. At the height of Tulipmania, in early 1637, a single bulb of the legendary *Semper augustus* fetched more than 10 years' worth of the average worker's wages. An English botanist bisected one of his host's bulbs and landed in jail until he could raise thousands of florins in compensation.

The bonanza couldn't last. When some bulbs failed to fetch their expected prices in Haarlem in February 1637, the bottom fell out of the market. Within a matter of weeks a wave of bankruptcies swept the land, hitting wealthy merchants as well as simple folk. Speculators were stuck with unsold bulbs, or bulbs they'd reserved but hadn't yet paid for (the concept of financial options, incidentally, was invented during Tulipmania). The government refused to get involved with a pursuit they regarded as gambling.

The speculative froth is gone, but passion for the tulip endures. It remains a relatively expensive flower, and cool-headed growers have perfected their craft. To this day the Dutch are the world leaders in tulip cultivation and supply most of the bulbs exported to Europe and North America.

Steen, Frans Hals and Rembrandt (see p38). The sciences were not left out: Dutch physicist and astronomer Christiaan Huygens discovered Saturn's rings and invented the pendulum clock; celebrated philosopher Benedict de Spinoza wrote a brilliant thesis saying that the universe was identical with God; and Frenchman René Descartes, known for his philosophy, 'I think, therefore I am', found intellectual freedom in the Netherlands and stayed for two decades.

The Union of Utrecht's promise of religious tolerance led to a surprising amount of religious diversity that was rare in Europe at the time. Calvinism was the official religion of the government, but various other Protestants, Jews and Catholics were allowed to practise their faith. However, in a legacy of the troubles with Spain, Catholics had to worship in private, which led to the creation of clandestine churches. Many of these unusual buildings have survived to the present day.

Politically, however, the young Dutch Republic was at an all-time low. The House of Oranje-Nassau fought the republicans for control of the country; while the house wanted to centralise power with the Prince van Oranje as *stadhouder* (chief magistrate), the republicans wanted the cities and provinces to run their own affairs. Prince Willem II won the dispute but died suddenly three months later, one week before his son was born. Dutch regional leaders exploited this power vacuum by abolishing the *stadhouder*, and authority was decentralised.

International conflict was never very far away. In 1652 the United Provinces went to war with their old friend England, mainly over the increasing

1602	1636–37	1700	1795
Dutch East India Company created	Tulipmania grips the country	End of Golden Age	French invade Holland

Figure 3.15: This figure displays state after page division, text grouping, graphics extraction and text ordering. See text for details

Chapter 4

Geometric layout analysis – Technical challenges

4.1 Exact glyph positioning

4.1.1 Motivation

From the outset, exact glyph positioning was not thought of to be an issue, and hence not given much thought. However it soon became clear that specific characters could be severely misplaced by up to several lines. The immediate consequence was that the existence of even a single equation could negatively affect text grouping of all surrounding text, and hence it would be necessary to look into. Although the problems encountered manifest themselves for many fonts, it is much more noticeable in those which contain variably sized glyphs, the prime example of which would be fonts used for math.

4.1.2 Problem description

The reason why this was a tough challenge to handle is composite, and ranges from bugs in PDFBox, to missing information about the fonts in use, to the sheer variety of font types supported in PDF documents. It was also clear that the text extraction component of PDFBox was probably not written with this functionality as a specific goal, so it was apparent that it would have to be patched in order to approximate the desired functionality.

Recovering this geometrical information has very much been an iterative process. This resulted in several consecutive steps which are best described by a series of illustrations. The document used for that was chosen because it exhibits several traits which will be described below.

The original information as received from PDFBox is shown to the left in Figure 4.1. While at a casual glance these misplacements might not seem like much, they severely complicate things down the line because:

1. Characters which are next to each other might appear to be completely above or below one another (in this instance have a look at for example the small *a*s in the 3rd line)
2. Characters might be placed well into another line
3. Characters will overlap graphic details, complicating their use for layout recognition purposes

4.1.3 Offsetting Y-coordinates

The first and most obvious step, independently of which font type is used, is to offset the original Y-coordinate by each characters height; this leaves us with the information drawn to the right in the same figure. From what we can see, this appears to fix most characters except for some which seemed about correct in the first step. No amount of hacky solutions emerged which could correctly predict which characters exhibited this behavior, so a sturdier solution was needed. Although not terribly important, we can also see that no glyphs with descenders are placed under the baseline.

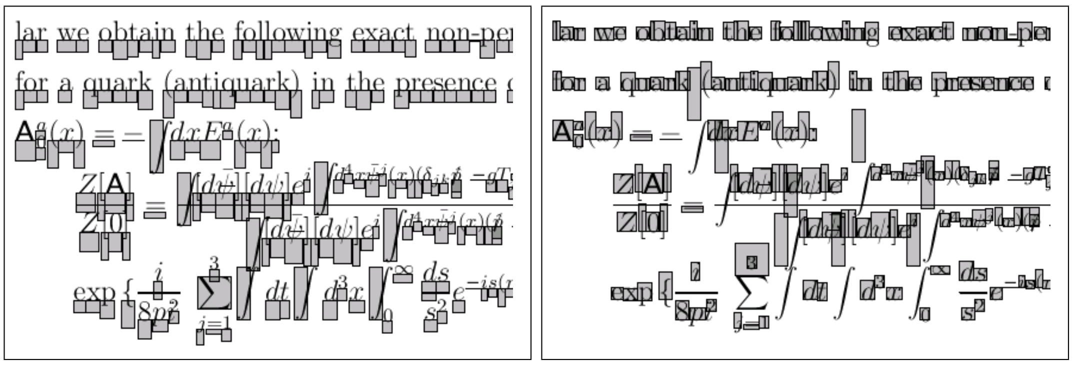


Figure 4.1: Left: Original information. Right: After naïve Y-coordinate offsetting

4.1.4 Discovering bounding boxes

These misplaced positions were correctly calculated according to the information available, but the error seemed to stem from the fact that a preceding PDF operator moved the current text position back up on the page before drawing. Strange as it seemed, it warranted some more investigation into the inner workings of PDFBox and its font library, FontBox.

It was discovered that most fonts has information available about boundary boxes for each glyph and for the whole font. A glyph boundary box is the smallest rectangle which will completely surround the given glyph, and also position it relative to the baseline (by letting, say, a lower case g span Y–values -200 through 600 for a common 1000-based font where 0 is the baseline). The font bounding box is the geometrical union of all the glyph bounding boxes, thus it can contain any glyph.

As it turned out, fonts which exhibited placement problems did indeed have at least one very tall symbol which skewed the vertical size of the font boundary box by a large amount. The jump back up was to make room for these huge font boxes, and the font rendering routines would place each glyph relative to that.

Since these boxes were not a part of the placement calculation for the text extraction code, characters were simply misplaced at the top end of the space which was made available.

Fixing this involved both a patch to PDFBox to let it export glyph bounding box information, and also some heavily specialized code which involves no less than three unit types – for text, glyph and display–, and Y–coordinates in two different coordinate systems (Java places $(0, 0)$ in the upper left, while PDF places it in the lower left).

4.1.5 Glyph mapping

Boundary boxes solved a lot of problems, but not all. Careful inspection of the equations would have shown (if one were to patch PDFBox' rendering routines) that some characters have two corresponding glyphs. In this example there is supposed to be both a big and a small integral sign (corresponding to TEX' internal `\integral` and `\textintegral` characters, respectively), with the small ones located above the fraction bar. This would again not be a problem if we wouldn't have had big integral signs crossing over several lines in the middle of running text, which destroyed the text grouping algorithms.

In short, both glyphs were mapped to the same Unicode representation (there is only one integral sign in Unicode, ditto with other symbols like square roots for which TEX has at least 5 glyphs of different sizes), and that was in turn used to look up character information, making it essentially random which of those sizes would be picked.

Fixing this involved patching PDFBox to use the glyph's unique code point within the font instead to avoid ambiguity. The image to the left in Figure 4.2 shows the state after these two fixes.

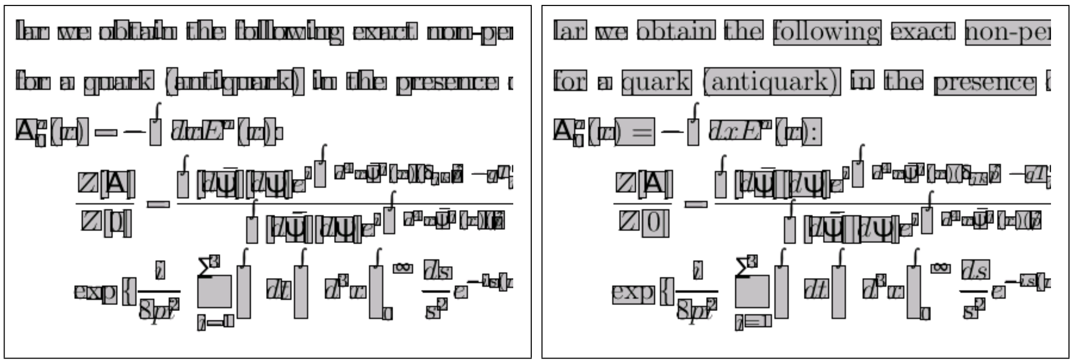

lar we obtain the following exact non-pe
for a quark (antiquark) in the presence
exp

Figure 4.2: Left: Correctly placed characters. Right: After combining characters into words

4.1.6 Missing information

Of course, life wouldn't be interesting if that was all there were to it. Except for the Y-position offsetting, the solutions above hold primarily for Adobe's various types of *Type 1* fonts, but there are of course several other types of fonts:

- *TrueType* fonts are also very common in modern PDF documents. They are handled a bit differently, and does not exhibit any of the problems mentioned above. Similar information about exact positioning is typically embedded in the font, but is not used by PDFBox (except indirectly for rendering).
- *Type 0* fonts, or *composite* fonts, are relatively common in documents typeset in non-latin character sets. They are called composite because they reference multiple descendent fonts. Support in PDFBox seems unfinished, but as they haven't been causing problems it has not been investigated further.
- *Type 3* is what must originally have given text extraction from PDF documents a bad reputation. All glyphs are rendered using the full set of PDF graphical operators, and can thus be used for diverse purposes such as including legacy or bitmap fonts, or drawing logos. Since this type of fonts can be generated by a PDF production tool (not to be confused with embedding font subsets into a document), these fonts typically and frequently contain little or no meta-information such as glyph sizes or Unicode mapping. Placement is usually not too far off, but PDFBox consistently guessed overly large dimensions for all glyphs. This problem was mostly alleviated by disabling a font cache in PDFBox. For the odd case where it still came back wrong, some code was added to drastically shrink those estimates, the effect of which can be seen in Figure 4.3. It should be stressed that this

adjustment does not seek to increase exactness, but rather to avoid problems.

- There are other, more obscure, font types available for use. These may or may not inherit the fixes outlined above, and may or may not work well.

It goes for all fonts that glyph sizes would have been computable by evaluating the drawing operations for each glyph, but that was considered out of scope for this project.

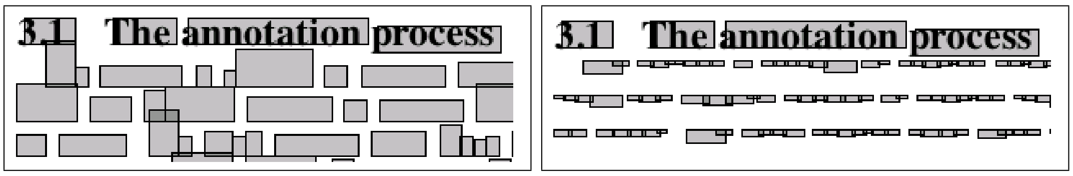


Figure 4.3: *Type 3* font before (left) and after adjustment. Characters are not rendered because of a PDFBox bug, but their contents remains available for text extraction.

4.2 Physical word segmentation

4.2.1 Motivation

Because of how whitespace is (not) represented in PDF files, It became abundantly clear early on that the in order to succeed, the project would need a very well performing and powerful word segmentation algorithm.

Let it be clear from the start that no stones has been left unturned in the search for an algorithm which would meet these standards. Literature has duly been consulted, but current research seems more focused on word segmentation as it applies to hand written manuscripts and Asian scripts. The source code of the free software OCR project located at Tesseract-OCR (n.d.) was also consulted, without getting any real help since it had the option of delaying a decision until a later stage where it could consult databases as to find the most probable combination of words. The algorithm eventually reached through this process is thus very much my own creation.

4.2.2 Technical background

PDF supports two ways of representing whitespace in text, they can be explicitly written in the data stream, or they can be left implicit by spacing. To fully appreciate the complications this second kind of whitespace introduces, it is helpful to see how this is done in PDF documents on a technical level.

While drawing text, a compliant PDF interpreter must keep track of a *text state* which the data stream in the PDF files manipulates with a range of operators. Some relevant ones for text rendering are the following[1]:

- `Tc` sets character spacing
- `Tf` sets font and size
- `Tw` sets word spacing
- `T*` jumps to start of next line, like a normal line break
- `TJ` shows text string. It supports horizontal adjustments for every glyph it draws in order to efficiently support typographical demands like kerning.
- `TD` offsets the text cursor on the page, by (x,y) coordinates it receives as arguments.

4.2.3 Explicit space representation

If a document were ever typeset using only this kind of whitespace representation, there would be no need to write this section, because text extraction would be straightforward. In Figure 4.4 we see operators from a PDF file writing "Scandinavian, which is " in the straightforward way, with two cases of moving glyphs closer to each other.

```
COSArray{[COSString{Scandinavian, which i}, COSFloat{17.6}, COSString{s}, COSFloat{17.6}, COSString{ }]}
PDFOperator{TJ}
```

Figure 4.4: Well behaved northerners

4.2.4 Implicit space representation

For comparison, a case of implicit space representation is shown in the log extract in Figure 4.5.

On line 2, `TJ` parses the prefixed arguments, and renders "Cani". Next it processes the number, which moves the current text position 677.5 glyph units right, then renders "d" and "a" with an added 19 glyph units between them. Line 6 offsets the current text position with 4 text units, and line 9 adjusts normal spacing between characters in the middle of the line. The last line renders more text.

The most interesting thing to note is that this line, which is arguably badly typeset, has very variable inter-word spacing. The final distances we are left with for this line can be seen in Figure 4.6.

[1] For a complete list, the reader is encouraged to consult chapter 9 in ISO 32000-1:2008 (2008)

```
COSArray{[COSString{Cani}, COSFloat{-677.5}, COSString{d}, COSFloat{-19.1}, COSString{a}]}
PDFOperator{TJ}

COSFloat{4.0714}
COSInt{0}
PDFOperator{TD}

COSFloat{-0.0122}
PDFOperator{Tc}

COSArray{[COSString{guardia}, COSInt{-298}, COSString{d}, COSFloat{51.4}, COSString{i},
        COSFloat{-940.8}, COSString{grosse}]}
PDFOperator{TJ}
```

Figure 4.5: Know your enemy: This sequence of operators renders the text *"Cani da guardia di grosse"*

$\mathtt{C}_{0.51}\mathtt{a}_{-0.23}\mathtt{n}_{-0.05}\mathtt{i}_{9.73}$ $\mathtt{d}_{0.43}\mathtt{a}_{10.41}$ $\mathtt{g}_{0.38}\mathtt{u}_{0.64}\mathtt{a}_{-0.07}\mathtt{r}_{0.18}\mathtt{d}_{0.18}\mathtt{i}_{0.70}\mathtt{a}_{4.41}$ $\mathtt{d}_{-0.54}\mathtt{i}_{13.74}$ $\mathtt{g}$...

Figure 4.6: Final character distances

4.2.5 Making sense of it

The problem of determining whether the distance between two arbitrary glyphs represents a real whitespace or not, might sound simple. In most realistic examples like the one presented here, there would be a clear difference of scale between the distances separating characters and words. It is indeed very simple to come up with a solution which fits any given document at hand; The difficulties will not really enter the picture until you need to generalize for *all* documents, because of the enormous variation encountered.

In the search for a working algorithm, a word boundary space has at different points in time been defined to be:

- A space broader than $x\%$ of font size
- A space broader than $x\%$ average width of the y preceding, succeeding or all characters
- A space broader than $x\%$ of any relation between the narrowest and broadest spaces in the line

Numerous algorithms determining these x in more or less smart ways have been proposed, but every inherent assumption in every one of them always caused them to fail in subtle or not so subtle ways. The final algorithm which is presented below is *in no way perfect*, but it certainly works for a very large percentage of the cases.

4.2.6 Implementation

The algorithm works by approximating the character text spacing (as could be set by the `Tc` operator) by averaging a number of the lower character distances within a line. This charspacing is subsequently used to *normalize* the distances, that is, subtract the found charspacing from every distance to ideally place character distances around 0, while word distances will remain larger (they will also be relatively much larger than before in comparison), thus facilitating identification. The identification of word boundaries itself ended up being perhaps overly simple, it just compares the normalized distance to a percentage of font size. This is actually the imperfect part of the equation, but as indicated above it was frustratingly difficult to find any assumptions which generally would hold.

Estimating charspacing

The main idea is that font kerning and other local adjustments will contribute relatively little to the observed distance between characters, whereas the more general applied character spacing will make up by far the biggest amount of the space. These local adjustments will contribute in both directions, so in many cases we will be able to get a somewhat good approximation if we average out a semi-random number of the smallest distances.

To put some numbers to this, say we have character distances varying from 3.5 to 9pt. If we iterate through the first n distances, with distances ranging from 3.5 to 4.5pt (the rest being skipped for being too big), the approximation of the character spacing would thus end up around 4.

The Java implementation of the algorithm can be seen in Figure 4.7.

```java
static float approximateCharSpacing(@NotNull List<PhysicalText> line) {
    /** the real lower bound where this algorithm applies might be higher, but
     *  at least for 0 or 1 distances it would be non-functional*/
    if (line.size() <= 1) {
        return 0.0f;
    }

    final float[] distances = calculateDistancesBetweenCharacters(line);
    Arrays.sort(distances);

    /**
     * This value deserves a special notice. When it was written semi-random above,
     *  this is essentially what was meant. We start out with the smallest distance,
     *  and will keep iterating until the numbers start to be bigger. The underlying
     *  assumption here is that word spacing will never be only 2 times the smallest
     *  space occurring between two characters.
     *
     * The 0.6 is a quite random number, it's purpose is to avoid breaking the
     *  algorithm for negative character distances (which are common and useful),
     *  and its only properties are that it is too small to ever separate a word, and
     *  that it is a positive number :)
     */
    final float maxBoundary = Math.max(0.6f, distances[0] * 2.0f);

    int counted = 0;
    float sum = 0.0f;

    for (float sortedDistance : distances) {
        if (sortedDistance > maxBoundary) {
            break;
        }
        sum += sortedDistance;
        counted++;
    }

    return sum / (float) counted;
}
```

Figure 4.7: Determining character spacing – Java implementation

Word segmentation algorithm

The algorithm iterates through the characters in a line from left to right. While segmenting words, it considers both ways of implementing word spacing. If there is already whitespace in the text, the words are segmented in the obvious way. If not, it, it calculates distances between all the characters and subtracts the approximated charspacing as described above. Then word boundaries are then identified where these normalized character distances are larger than a fraction of the font size.

```
@NotNull
Collection<PhysicalText> createWordsInLine(@NotNull final List<PhysicalText> line) {

    /* keep the characters sorted at all times. note that unfinished words are put back into
     *   this queue, and will this be picked as currentWord below
     */
    final Queue<PhysicalText> queue = new PriorityQueue<PhysicalText>(line.size(), sortByLowerX);
    queue.addAll(line);

    /* this list of words will be returned */
    final Collection<PhysicalText> segmentedWords = new ArrayList<PhysicalText>();

    /* if we already have whitespace information */
    final boolean containsSpaces = containsWhiteSpace(line);

    /* an approximate average charspacing distance */
    final float charSpacing = approximateCharSpacing(line);

    /* all font sizes will be the same. if it is missing (Type3 fonts >: ) just guess 10 */
    final float fontSize;
    if (line.get(0).getStyle().xSize != 0) {
        fontSize = (float) line.get(0).getStyle().xSize;
    } else {
        fontSize = 10.0f;
    }

    /**
```

```java
     * iterate through all texts from left to right, and combine into words as we go
     */
    while (!queue.isEmpty()) {
        final PhysicalText currentWord = queue.remove();
        final PhysicalText nextChar = queue.peek();

        /* we have no need for these spaces after establishing word boundaries, so skip */
        if (" ".equals(currentWord.getText())) {
            continue;
        }
        /* if it is the last in line */
        if (nextChar == null) {
            segmentedWords.add(currentWord);
            break;
        }

        /**
         * determine if we found a word boundary or not
         */
        final boolean isWordBoundary;
        if (containsSpaces) {
            isWordBoundary = " ".equals(nextChar.getText());
        } else {
            final float distance = currentWord.getPos().distance(nextChar.getPos());
            isWordBoundary = distance - charSpacing > (fontSize / 8.0f) + charSpacing*0.5f;
        }

        /**
         *  combine characters if necessary
         */
        if (isWordBoundary) {
            /* save this word and continue with next */
            segmentedWords.add(currentWord);
        } else {
            /* combine the two fragments */
            PhysicalText combinedWord = currentWord.combineWith(nextChar);
            queue.remove(nextChar);
            queue.add(combinedWord);
        }
    }

    return segmentedWords;
}
```

Figure 4.8: Word segmentation algorithm – Java implementation

4.2.7 Performance

Because it was necessary to have a benchmark against which one could compare the output, a compilation of 5502 lines of text was compiled. Depending on parameters the code would have around 40 lines with at least one error. Of those several would be related to punctuation (which in any case could be fixed automatically), some lines were a bit ambiguous as to what would be the correct segmentation, and some were picked because they were notoriously difficult to get right. Since this was a very quickly crafted, manually corrected set of text, undoubtedly there will also be some lines flagged as correct which ideally would have been segmented differently. In any case, for this set of text, the failure rate counted in lines would be less than 0.01%. In the wild, the success ratio will probably be lower, because of the variation mentioned.

4.3 Extracting graphical information

4.3.1 Motivation

It was decided that it would be very beneficial to have information about all graphical elements on a page for the logical layout recognition. In particular it would make it easier to separate out floating boxes with text, text which belonged inside of or next to an illustration, find graphical text separators, *etc.*

4.3.2 Vector graphics

As PDF is at its core a resolution independent data format where everything is drawn in terms of operators which manipulate and draw shapes, it should thus not come as a surprise that most embedded graphics are vector based.

All drawing operators were originally disabled for the PDFBox' text extraction code, so the first step towards this information was to reenable them. Thanks to PDFBox' well-thought-out architecture, this was as easy as swapping a Java `properties`-file with another one.

Then, i needed to somehow come up with a capturing surface on which it would be possible to draw. The existing rendering code was again cleverly written, so eventually the methods which had to be reimplemented was narrowed down to the interface seen in Figure 4.9:

The two vector methods receive a `java.awt.geom.GeneralPath` object which describes the shape of the object which is to be drawn. Unfortunately, because frequently several non-overlapping objects are described by the same `java.awt.geom.GeneralPath` object, computing a simple bound is not sufficient. It turned out that one of the operators available in that class is `closePath()`, so it was straightforward to write a method to split paths around those.

4.3.3 Bitmap graphics

Bitmap graphics were a bit easier to get access to. Physically the bitmap data is stored in separate content streams in the PDF, and PDFBox conveniently provides access to it as `java.awt.Image` objects. Both vector and bitmap graphics can be subject to transformations, but while PDFBox took care of transforming vector data, it is up to the rendering

```java
/**
 * This represents a surface on which it is possible to draw. <p/> For graphic segmentation purposes
 * no real drawing will occur, but a list of graphic placements will be created
 */
public interface DrawingSurface {

// ------------------------- PUBLIC METHODS -------------------------
    void clearSurface();

    void drawImage(Image image, AffineTransform at, Shape clippingPath);

    void fill(GeneralPath originalPath, Color color, Shape clippingPath);

    @NotNull
    List<GraphicContent> getGraphicContents();

    void strokePath(GeneralPath originalPath, Color color, Shape clippingPath);
}
```

Figure 4.9: Drawing surface interface – Java definition

layer to perform the transformations on bitmap data and ultimately compute the final position of the image. The provided clipping path is in practice PDF's implementation of clipping images, so it had to be considered as well.

4.3.4 Missing in action

There seems to be a bug in PDFBox which affects handling of (at least) graphics in the Tagged Image File Format (TIFF) format, so some graphics will be missing. This would probably have been possible to fix as it seemed to stem from incorrect assumptions about endianness in the PDFBox code, but there was no time for that.

4.3.5 Putting the pieces together

It frequently occurs that one single vectorized image can be composed of up to several thousand vector operations, so for practical use it is important that these parts be combined into the real present graphics. Also, perhaps more surprisingly, bitmapped images are subject to a relatively high degree of fragmentation as well, and must thus be recombined in the same way.

It should be noted that erroneous combination of graphical components will mean that we could be left with information about a too big graphic which could potentially cover a lot of text. Since graphical information is primarily used to separate out text related to images, such information would be rendered useless. It was thus decided to enforce a rather strict boundary for how far apart these graphical components can be and still be combined. Also, many documents render invisible boxes around body text which would frequently overlap with real graphics. The colour passed to our "rendering" component was thus used to filter out most graphic with the background colour before combining even starts.

Chapter 5

Logical layout analysis

The line which separates what belongs in a logical layout analysis and what belongs in a geometric layout analysis is not a firm one. There has been considerable variation as to what goes into which throughout the history of the research done in this field. As is made clear by its name, geometric layout analysis is very much a process which relies on geometrical and spatial information, and in this project it includes most processes which are based on manipulation of coordinates. The processes which constitute the logical layout analysis are principally concerned with blocks of text and their relationships to each other.

It should be made clear from the start that the main focus of this thesis has been on the geometric layout analysis, and that has been where most of our effort has been spent. What was deemed to be the one of most intriguing things about the geometric layout analysis was that as long as it could be made to perform well, it would go a long way towards later enabling a powerful logical layout analysis without too much work. This means, unfortunately, that the theory behind what is presented here is not as developed as what has been presented in Chapter 3 and also partly Chapter 4. It also means that while the implementation of the different parts of the geometric layout analysis has had time to mature, what is presented here can in many ways be considered more of a proof of concept.

The results from the processes which make up the logical layout analysis are inherently a bit more difficult to render and show graphically than those from the geometric layout analysis. To give an impression of how it works, there are excerpts of the results from running an example document through the application, provided in Appendix A.

5.1 Text labelling

The heart and soul of the logical layout analysis as described in Section 1.3.3 is the text labelling process. This is in many ways, together with proper text ordering and separation of floating text, the reason why we wanted to go to all the trouble and provide such a thorough geometric layout analysis. Although more plentiful and more varied labels would have been desirable, the ones which are implemented in this project are listed below in the order they are identified:

`Body text` Counts how many characters are formatted with each *style*, a combination of font and font size, which is represented in the document, and chooses the one with the most.

`Title` The heuristic for finding the title is quite easy: Choose the heading with the biggest font size on the first page. After identifying it, it is extracted from the main text.

`Abstract` If one of the first pages has a single-line block with a style which is bigger/bolder than body text, and contains the world abstract, it is choosen as an abstract header. All the body text following it until the next header-like block is encountered, is taken to be the abstract text.

`Footnote` Footnote identification is admittedly also very simple. It looks for blocks on the lower part of the page with smaller or slightly different style than body text, and checks that they start with something like a number. When found, they are extracted from the main text into floating elements. Identification of other types of floaters like top text and text located besides the main text should be very similar to implement.

Three levels of `div` – headers After having labelled title and abstract, the code then tries to identify up to three levels of section headers. In order to do that, it starts by assembling a list of potential header styles. To decide if a certain style might be a header, it is compared to the body text style, which has already been identified. Essentially the style should preferably have bigger font size, or alternatively at least the same size but with a bolded, slanted or different font face.

Having constructed the list of potential header styles, the code iterates through all the content to try to recognize the different levels of section headers. Looking at the style itself proved to not be sufficient, so the code also takes a look at the content of potential section headers. When that content starts with something which could correspond to a numbered section, *i.e.* something like *1.2* or *(1a)*, it is very probable that it is, effectively, a section header.

The last bit of information which is used in determining which lines might be which kind of headers, is the order in which they appear. It is assumed that a `div1`–header will always appear earlier in the document than a `div2`–header, and so on.

After having identified the different headers in the document, these headers are subsequently used to create a logical hierarchical representation where document title, abstract, floaters and figures are separated from the main text. The main text itself is put into a tree of `div`–elements, where lower level headers will be placed underneath their higher level headers, and all content will be placed together with its introducing header.

5.2 Dehyphenation

When documents are typeset, long words are frequently divided over two lines with a hyphen to ensure that every line contains a suitable amount of text. In order to recover the original text, it is neccessary to recombine these word fragments. The reason why this is difficult is that it sometimes the hyphen should not be removed because it has a semantic meaning.

This is a problem which has been solved elsewhere, and in this project an existing solution was used which was developed as a part of the inherited TextGrabber project, as mentioned in Chapter 2.

5.3 Logical paragraph segmentation

We already segmented text on several levels in the geometric layout analysis, what remains here is to convert physical blocks of text into logical paragraphs. The difference between the two is simple, a block is just a collection of lines, while a logical paragraph is equivalent to the normal text subdivisional entity we normally think about. Naturally, a paragraph might correspond to part of a block, or it might span several, for example in the case when it is divided over two pages.

In order to form these logical paragraphs, the first thing which needs to be done is to combine consecutive text blocks with equal style. Then the second step is to redivide the resulting text based on indentation, exploiting the fact that new paragraphs are normally indented. Additionally, this process should probably be able to split paragraphs if there is more vertical space than normal between them, but this is not implemented yet.

5.4 Output format

The output XML schema to be used was for a long time a design consideration of the program. It was desireable to find one which would natively and easily support all the different elements which the logical layout analysis recognizes, and which preferrably was widely used.

This process was difficult, because the different standards each favour different kinds of information. The focus could be on metadata, on physical page layout, on archivation of and uniquely identifying scanned images, *etc.* The many foci meant that at least initially it all seemed like a bewildering array of partly overlapping and related standards. Eventually an overview which breaks down different schemas into categories was found in Day (2010). Suggested in the category of *Text encoding* was the Text Encoding Initiative (TEI) Guidelines[1].

It was chosen for this project because it supports everything we want to represent, it seemed to have some momentum and had been in use for some time, and finally because there were other projects in our department which already used it.

TEI is an international consortium which is dedicated to maintaining these guidelines as a recommended standard for textual markup. The guidelines, officially called *Guidelines for Electronic Text Encoding and Interchange*, are described in Wittern, Ciula, and Tuohy (2009).

They provide recommendations for markup for a a wide range of textual, physical, literary, and linguistic features. Groups of features exist for describing *e.g.* document structure, punctuation, quotations, semantics, links and references, citations, verse, *etc.*

Although this application really just needed a very small subset of the comprehensive set of elements available in the standard, the structured way in which the standard is defined practically begged for an automatic generation of a Java model. This was accomplished with a Maven plugin for The Java™ Architecture for XML Binding (JAXB). JAXB is described in Joseph Fialli (2003). This was not entirely straightforward because some elements were duplicated at several levels, something which is allowed in the XML schemas used, but not in JAXB. This was solved by creating a binding file which mapped these problematic tags to other names in the Java model. XPath, which is described in Clark and DeRose (1999), was used to do the replacements.

[1]http://www.tei-c.org/Guidelines/P5/

Chapter 6

The application – PDFExtract

6.1 Technical introduction

The application is a free-standing Java 6 SE application with a multi-moduled Maven 2[1] build system, altogether consisting of around 8500 lines of code. The different modules are written to be as independent as possible, so it makes most sense to describe them individually. The source code for the application can be found at `http://github.com/elacin/PDFExtract`.

The dependency structure of the different modules in the project can be seen in Figure 6.1.

6.1.1 Model

The `model` contains the entities referenced throughout the application and is split into various packages.

geom Since a large part of the task was about manipulating geometrical data, it was necessary to write some supporting classes to facilitate this.

1. `Rectange` and `Point` are specialized classes which represent the obvious things
2. `RectangleCollection` enables grouping of rectangles, queries about their locations, and caching of frequent queries
3. `Sorting` is a utility class with frequent ways to sort geometrical data
4. `HasPosition` is an interface which is implemented by every class with an associated location

[1] `http://maven.apache.org`

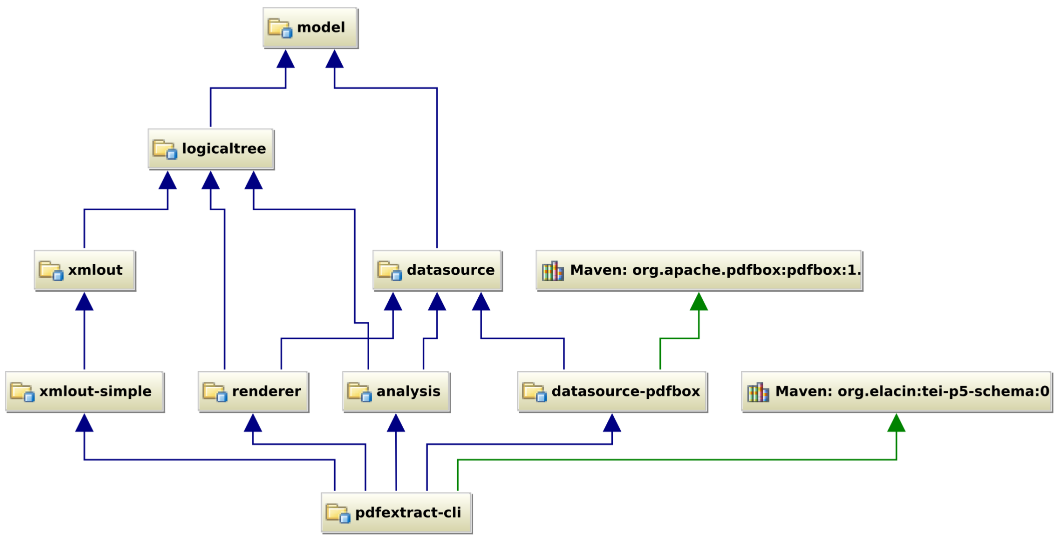


Figure 6.1: The architecture of PDFExtract. The lines represent dependency so that everything ultimately depends on `model`, and `pdfextract-cli` depends on everything. A subset of external dependencies are shown only, with green arrows. `org.elacin.tei-p5-schema` depends on `xmlout-simple`, but it was not picked up by the UML-tool

5. `PositionCache` is an abstract class which provides an organized way to update a cache of the current position. This is useful because various classes contain a changing collection of rectangles.

style A `Style` is a combination of a specific font face and text size. This package provides this entity along with various ways to compare them.

content This package defines a hierarchy of classes which represents the content on a page, and is used to physically segment the page. The classes and their relationships can be seen in Figure 6.2

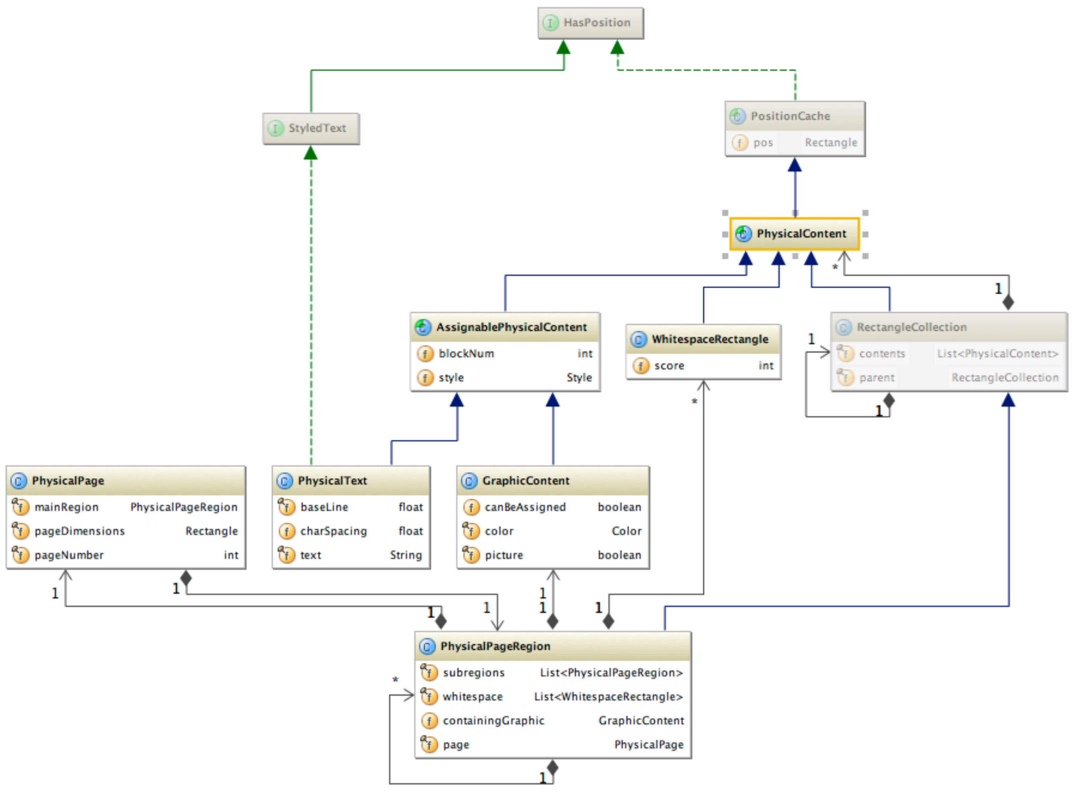


Figure 6.2: Inheritance is marked by upwards pointing dark blue lines, implementation of an interface by dotted lines, and class relationship with multiplicity by black lines. Shaded classes are in other packages. Note especially how `PhysicalPageRegion` supports being split up in subregions

6.1.2 Datasource

This module contains an interface which specifies a `PDFSource`. This is a way to decouple the application logic from the library used to read the PDF file, so the entities contain all the content from the file in a lightweight format, the structure of which can be seen in Figure 6.3. The application naturally ships with an implementation of this interface for PDFBox.

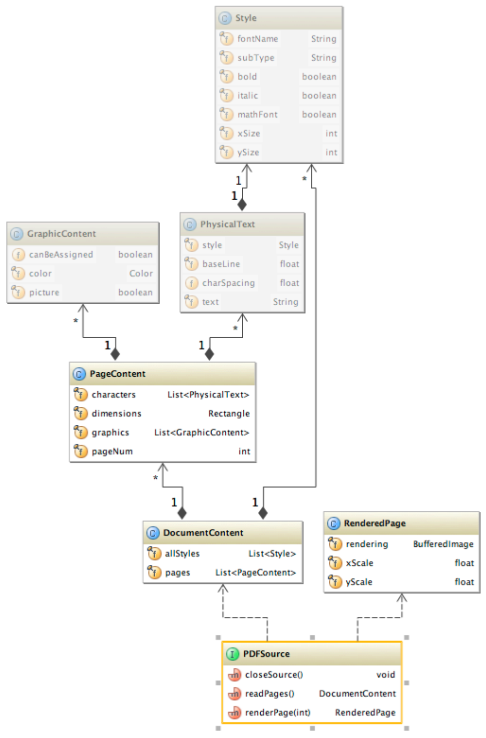


Figure 6.3: The PDF library interface. `PhysicalText`, `GraphicalContent` and `Style` are from the `content` package seen in Figure 6.2

.

6.1.3 Datasource – PDFBox

Although PDFBox is a sophisticated and well developed library, it was felt during development that it may have left a thing or two to be desired.

It was especially the pain experienced with exact positioning of characters and missing graphical elements which is described in Section 4.3.4, which caused this feeling. For this reason it was decided to implement an interface – `PDFSource` – through which one could communicate with the library. Also the implementation details of PDFBox was abstracted away. The rough details of this interface can be seen in Figure 6.3

Somewhat interestingly, another obvious problem is the performance penalty implied by having the library implemented in Java compared to for example interfacing with a C library. Interpretation of PDF files is relatively quick, but font handling is abysmal.

The interface obviously supports reading documents, and textual and graphical contents is returned page by page. It also abstracts out rendering of individual pages so that no other parts of the program need to depend on PDFBox.

6.1.4 Logical tree

It was deemed appropriate to, in addition to the physical representation of the tree outlined above, also create a logical representation. This tree structure is the output of the physical page segmentation. For logical segmentation, all the nodes can be assigned `Roles` according to which function they are deemed to have. The hierarchy can be seen in Figure 6.4.

6.1.5 analysis

The `analysis` package is the heart and soul of the application, it contains all the classes which together perform the geometrical and logical analysis described in Chapters 3 and 5.

6.1.6 tei-output and tei-p5-schema

As mentioned in Section 5.4, it was decided to use version 5 of the TEI Guidelines as the preferred output from the application. Since the definition was so structured, it was decided to generate a Java model which makes it easier to use. This model has then then packaged as a separate project, which could be reusable by anyone. For exactly that reason it is placed in a separate github project[2]. In Figure 6.1 this is seen as an external dependency called “`Maven:org.elacin:tei-schema:0.1`”.

6.1.7 xml-output

The second of the two modules for writing the results of the analysis to XML is a lot simpler. It essentially dumps the identified physical data structure into a homegrown XML dialect. This was mostly used for debugging.

[2]`https://github.com/elacin/TEI-P5-Java-model`

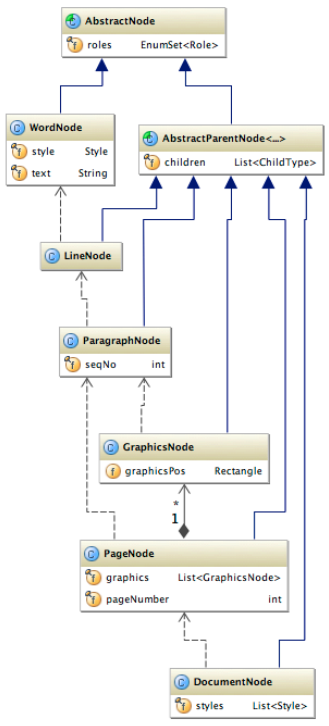


Figure 6.4: The document tree which is the output of the page segmentation algorithm. Note that the dotted relationships between the nodes should be 1→∗, but since each node class' type of subnode is determined by generics, my UML-tool did not understand that.

6.1.8 renderer

There is also a rendering component bundled, which main use has also been for debugging. It delegates the rendering of the actual document data to the `datasource`, and renders requested portions of the logical tree and/or physical content on top of it. It has been used to create a sizeable share of the figures in this thesis.

6.2 Example output

In order to showcase what the application can do, it was decided to include three pages from an example document, and the output generated from the `xml-output` and `tei-output` modules. The logical output from the `tei-output` module is located in Appendix A.2, while an excerpt from the physical output from `xml-output` can be found in Appendix A.3.

It should be mentioned that this exact document was chosen because the logical output was considered good, and should be considered more of a showcase of what a well-written logical layout analysis can obtain.

Chapter 7

Performance evaluation

7.1 Benchmark

In order to get an idea of how well PDFExtract performs on real life documents, it was necessary to test it on previously unseen data. In keeping with the introductory comparison, it was logical to again test on both a set of the ACL Anthology Reference Corpus (ARC), and a set of Norwegian Open Research Archives (NORA) documents. Two sets of documents were chosen, and the output for every document was then manually inspected to see how well the program had performed. To keep the amount of work at a reasonable level, it was decided to consider 5 random pages from each document. Also, since the logical layout analysis is still in early stages of implementation, it was decided that it made most sense to focus the evaluation on the results of the geometric layout analysis.

NORA A sample of 54 NORA documents from 2007 were chosen, amounting to 266 pages altogether (4 documents had only 4 pages in total). The results can be seen to the left in in Figure 7.1.

ARC For the ARC, a sample of 39 documents, a third of the `C04` section, was chosen. From these 39 documents, a total of 195 pages were examined. The results can be seen to the right in the same figure.

NORA

Total number of pages	**266**
Graphics problems	3
Needlessly separated text	15
Incorrectly detected columns	4
Incorrect ordering of text	4
Correctly segmented pages	**247**

ACL

Total number of pages	**195**
Graphics problems	12
Needlessly separated text	25
Incorrectly detected columns	6
Incorrect ordering of text	16
Correctly segmented pages	**136**

Figure 7.1: Results of performance evaluation. The numbers will not add up, because some pages exhibited several problems

7.2 Behind the numbers

These numbers deserve some discussion, because a page-level success rate of just over 80 % may seem relatively low. Note that here we apply a maximally stringent definition of 'success', *i.e.* a perfect output for all content elements. To start off, let us have a look at the most common errors encountered and what causes them.

7.2.1 Needlessly separated text

This is definitely the most frequent category in terms of errors found. It is dominated by two subgroups:

1. Formulæ, which might lose fragments located well above or below the baseline, for example arguments to a summation operator, or their associated number in the left or right margin)
2. Numbered section headers, which are some times separated from their number

These errors highlight a problem with relying too much on the functionality of the whitespace rectangles, as this content is, in contrast to normal body text, purposely laid out with extra spacing. There is already a special strategy in place for segmenting formulae, but it is apparently not quite clever enough yet. Another discussion is whether whitespace should be used for horizontal division of text at all after the high-level column layout has been established. These errors are not necessarily grave in that they have limited consequence for the surrounding text, but errors they remain.

7.2.2 Graphics problems

The second biggest category. In practise this means that fragments of graphical content are not properly recognized as belonging together. Since graphical elements are used for text grouping, this may produce some odd output in that parts of text which belong inside a graphic will fail to be marked and separated as such. Also a lot of these errors are caused by missing graphical information (as mentioned in Section 4.3.4) which in turn is caused by either PDFBox or the PDF files themselves, and would hence be difficult to fix. But in any case, graphics segmentation is a weak point.

7.2.3 Incorrectly detected columns

This is the category of errors with the biggest consequence for the output, and occurs when a page region is separated around a nonexistent column boundary. A frequent culprit happens to be tables, which is natural since they are inherently column based. This would be fine, if it were not for the fact that there is not yet any table recognition code; in other words, a table which is not contained in a graphical frame will break up output in unfortunate ways. There were also a handful of real false positives, frequently coinciding with blocks of formulae with aligned spacing.

The conclusion here is that the column boundary filtering code qualifies as another weak point.

There was exactly one occurrence of a column boundary which was not completely recognized, and that was because there was text crossing over the whole boundary, something the algorithm is not prepared to deal with. Apart from that it worked very satisfyingly.

7.2.4 Incorrect ordering of text

The vast majority of this category was caused by the above-mentioned incorrectly detected columns. The remaining few were local errors where two paragraphs placed more or less side by side would come out in the wrong order. This is a consequence of what must be a bug in the text ordering code.

7.3 Evaluation

All in all, it was surprising that the performance was much better in the documents from the NORA collection than from the ARC, as the NORA collection contains a much more heterogenous collection of documents. The main reason why this is so, is that the set of documents which was used from ARC, happened to consist of many formulæ and graphics heavy papers which exposed imperfections in the application to a bigger degree. The layout variation found in the NORA documents is satisfyingly handled.

Chapter 8

Conclusion

Let me start out by affirming that the problem of complete page segmentation has decisively proved itself again to be a hard one. In the context of text extraction from PDF files, this project has sought to restate a problem of making sense of a data stream into a principally geometrical problem. By drawing from the considerable amount of research on OCR and augmenting it with the extra information available, it was hoped to make some progress towards the ambitious goal of solving this problem. Ambitious enough that (the above-mentioned results notwithstanding) one would perhaps be wise not to expect to fully solve it in the general case without a human brain at one's disposal. The loss of structural information in PDF files is so massive that its wide use for archival purposes is nothing less than troubling.

Though, progress has indeed been made:

- The main ingredient in the analysis which has been developed, the whitespace rectangle covering algorithm, has been extended to make use of the additional information available in PDF compared to OCR to produce more useful rectangles
- These rectangles are successfully used to recover column boundaries and to group text
- The text ordering algorithm successfully recovers reading order from pages with complex layout

Comparing PDFExtract to other state-of-the-art PDF text extractors is quite an apples-to-oranges comparison, given that it tries to do more. Grouping content in this way is really about taking a risk by ignoring the inherent structure in the content stream from the PDF file, while trying to find a correct logical structure. If a mistake is made in that process, the output might be of worse quality than what one would obtain by simply performing a naïve geometrical sorting of the contents.

That being said, the output of PDFExtract is overall very good. Most of the mistakes found above are related to formulæ handling and segmenting content together with graphical content. These are in themselves challenges which, as far as the author is aware of, are not addressed in full elsewhere. A failure to perfectly perform these two functions will most of the time mean that the output contains isolated text fragments with a piece of a formula or an illustration. This is not necessarily very grave for the quality of the output.

The real number of the examined pages with seriously damaged content lies around 15 (10 from *incorrectly detected columns*, and a handful of *graphics problems*). This leaves us with 3-4% of pages with serious errors, most of which could be remedied by more intelligent column filtering code.

To conclude, PDFExtract might need more work in order to work even better. The theoretic base on which it is built seems to be both sound and useful. But also in its current state it is competitive with, if not superior to, other alternatives. The clean, modular design of the application combined with a permissive open source license makes reuse of individual modules possible.

8.1 Further work

As seen in the performance evaluation, possibilities for further work are plentiful. The overall usefulness of the geometric layout analysis method presented in this project will depend on a few factors:

- A solution needs to be found for dealing with tables, which are inherently column-based.
- For this project the column finding algorithm was deliberately made more liberal in what it accepts. For that to be a good idea, there needs to be smarter filtering process in place.
- Graphics segmentation needs to be further developed

Also, opportunities abound for developing a more in-depth logical layout analysis, something which could easily have merited a project on its own.

Bibliography

Antonacopoulos, A., Gatos, B., & Bridson, D. (2007, sept). Page Segmentation Competition. In *Document Analysis and Recognition, 2007. ICDAR 2007. Ninth International Conference on* (Vol. 2, p. 1279 -1283). Available from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4377121

Baird, H. S. (1992). Background structure in document images. In *In advances in structural and syntactic pattern recognition* (pp. 17–34). World Scientific. Available from http://cvit.iiit.ac.in/papers/DocStructure.pdf

Bird, S., Dale, R., Dorr, B. J., Gibson, B., Joseph, M., Kan, M.-Y., et al. (2008, Jan). The ACL Anthology Reference Corpus: A reference dataset for bibliographic research in computational linguistics. In *Lrec.* European Language Resources Association. Available from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.139.2351&rep=rep1&type=pdf

Breuel, T. (2002, Jan). Two geometric algorithms for layout analysis. In D. Lopresti, J. Hu, & R. Kashi (Eds.), *Document analysis systems v* (Vol. 2423, p. 687-692). Springer Berlin / Heidelberg. Available from http://www.springerlink.com/index/w0fn0bnjecvqtlte.pdf

Breuel, T. (2003). Layout analysis based on text line segment hypotheses. *DLIA'03*. Available from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.124.3478&rep=rep1&type=pdf

Cattoni, R., Coianiz, T., & Messelodi, S. (1998, Jan). Geometric layout analysis techniques for document image understanding: a review. *ITC-irst Technical Report TR#9703-09*. Available from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.37.353&rep=rep1&type=pdf

Clark, J., & DeRose, S. (1999, Jan). XML path language (XPath) version 1.0. *faa.gov*. Available from http://www.faa.gov/about/office_org/headquarters_offices/ato/service_units/techops/atc_comms_services/swim/documentation/media/compliancy/Xpathv1.0.pdf

Day, M. (2010, Jan). IMPACT best practice guide: Metadata for text digitisation and OCR. *opus.bath.ac.uk*. Available from http://opus.bath.ac.uk/23311/1/IMPACT-metadata-bpg-pilot-1.pdf

ISO 32000-1:2008. (2008). *Document management, Portable Document Format, PDF 1.7*. ISO, Geneva, Switzerland. Available from http://www.adobe.com/devnet/pdf/pdf_reference.html

Joseph Fialli, S. V. (2003, Jan). The Java™ Architecture for XML Binding (JAXB). *JSR Specification*. Available from http://svn-mirror.glassfish.org/hj3/tags/0.3/etc/jaxb-1_0-fr-spec.pdf

Mao, S., Rosenfeld, A., & Kanungo, T. (2003, Jan). Document structure analysis algorithms: a literature survey. *Proc. SPIE Electronic Imaging*. Available from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1.6218&rep=rep1&type=pdf

Meunier, J.-L. (2005). Optimized XY-Cut for Determining a Page Reading Order. In *ICDAR '05: Proceedings of the Eighth International Conference on Document Analysis and Recognition* (pp. 347–351). Washington, DC, USA: IEEE Computer Society. Available from http://dx.doi.org/10.1109/ICDAR.2005.182

Nagy, G., Seth, S., & Viswanathan, M. (1992, July). A prototype document image analysis system for technical journals. *Computer*, *25*(7), 10 -22. Available from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=144436

PDFBox, A. (n.d.). PDFBox project [Computer software]. Website. (http://pdfbox.apache.org)

Shafait, F., Keysers, D., & Breuel, T. (2008, June). Performance Evaluation and Benchmarking of Six-Page Segmentation Algorithms. *IEEE Trans. Pattern Anal. Mach. Intell.*, *30*, 941–954. Available from http://portal.acm.org/citation.cfm?id=1399104.1399432

Tesseract-OCR, G. (n.d.). Tesseract-OCR [Computer software]. (http://code.google.com/p/tesseract-ocr)

Wittern, C., Ciula, A., & Tuohy, C. (2009, Jan). The making of TEI P5. *Literary and Linguistic Computing*. Available from http://llc.oxfordjournals.org/cgi/content/abstract/fqp017

List of Figures

Acronyms

ACL the Association for Computational Linguistics. 24

ARC ACL Anthology Reference Corpus. 24, 25, 77, 79, 89

DFKI das Deutsche Forschungszentrum für Künstliche Intelligen. 19, 25

JAXB The Java™ Architecture for XML Binding. 68

NORA Norwegian Open Research Archives. 21, 24, 25, 77, 79

OCR Optical Character Recognition. 9, 11, 15, 27, 28, 30, 39, 55, 81

PDF Portable Document Format. 7

PDF/UA PDF/Universal Accessibility, or “Tagged PDF”. 7

TEI Text Encoding Initiative. 68, 73

TIFF Tagged Image File Format. 62

UiO University of Oslo. 19

UML Unified Modeling Language. 69, 73

XML eXtensible Markup Language. 23, 24, 26, 68, 73

Appendix A

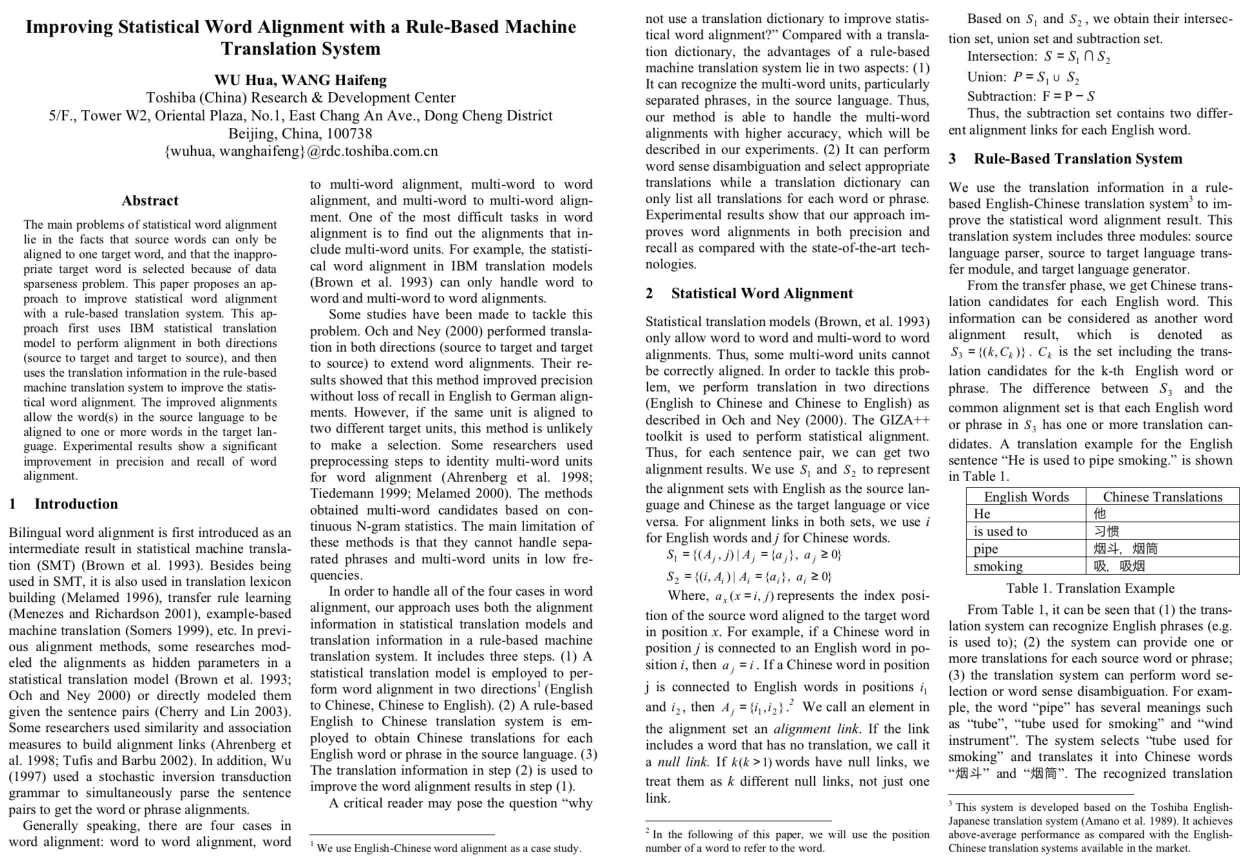
Example document and result of analysis

To give an idea of what the application is able to do, three pages of an example document from the ARC is provided, along with the output from the complete document layout analysis, and then an excerpt from the output of the geometric layout analysis alone. The output from the complete analysis displays several features which are described in Chapter 5 such as title identification, building a hierarchy of sections and subsections, extraction of abstract, identification of footnotes, and separation of text contained within graphical figures. We also see logical paragraph segmentation and removal of publication-introduced hyphens.

A few shortcomings are also present. There is no code to extract author information, so it appears in the text before the first heading. Also, there is no functionality yet to extract figure captions from the text, so they remain in the body text. Individual characters in equations are frequently not combined into words, this is because it has both not been given much attention and it is frequently not clear what the correct output of such a word segmentation would be. Doing this in a "right" way would require interpretation which is entirely out of scope for this project.

There are some characters missing from the output, namely some chinese characters and math characters like $\cap$ and $\cup$; this is the fault of the LaTeX software used to typeset the text, they survive into the Unicode–encoded output from the application.

A.1 Example document



Improving Statistical Word Alignment with a Rule-Based Machine Translation System

WU Hua, WANG Haifeng
Toshiba (China) Research & Development Center
5/F., Tower W2, Oriental Plaza, No.1, East Chang An Ave., Dong Cheng District
Beijing, China, 100738
{wuhua, wanghaifeng}@rdc.toshiba.com.cn

Abstract

The main problems of statistical word alignment lie in the facts that source words can only be aligned to one target word, and that the inappropriate target word is selected because of data sparseness problem. This paper proposes an approach to improve statistical word alignment with a rule-based translation system. This approach first uses IBM statistical translation model to perform alignment in both directions (source to target and target to source), and then uses the translation information in the rule-based machine translation system to improve the statistical word alignment. The improved alignments allow the word(s) in the source language to be aligned to one or more words in the target language. Experimental results show a significant improvement in precision and recall of word alignment.

1 Introduction

Bilingual word alignment is first introduced as an intermediate result in statistical machine translation (SMT) (Brown et al. 1993). Besides being used in SMT, it is also used in translation lexicon building (Melamed 1996), transfer rule learning (Menezes and Richardson 2001), example-based machine translation (Somers 1999), etc. In previous alignment methods, some researches modeled the alignments as hidden parameters in a statistical translation model (Brown et al. 1993; Och and Ney 2000) or directly modeled them given the sentence pairs (Cherry and Lin 2003). Some researchers used similarity and association measures to build alignment links (Ahrenberg et al. 1998; Tufis and Barbu 2002). In addition, Wu (1997) used a stochastic inversion transduction grammar to simultaneously parse the sentence pairs to get the word or phrase alignments.

Generally speaking, there are four cases in word alignment: word to word alignment, word to multi-word alignment, multi-word to word alignment, and multi-word to multi-word alignment. One of the most difficult tasks in word alignment is to find out the alignments that include multi-word units. For example, the statistical word alignment in IBM translation models (Brown et al. 1993) can only handle word to word and multi-word to word alignments.

Some studies have been made to tackle this problem. Och and Ney (2000) performed translation in both directions (source to target and target to source) to extend word alignments. Their results showed that this method improved precision without loss of recall in English to German alignments. However, if the same unit is aligned to two different target units, this method is unlikely to make a selection. Some researchers used preprocessing steps to identity multi-word units for word alignment (Ahrenberg et al. 1998; Tiedemann 1999; Melamed 2000). The methods obtained multi-word candidates based on continuous N-gram statistics. The main limitation of these methods is that they cannot handle separated phrases and multi-word units in low frequencies.

In order to handle all of the four cases in word alignment, our approach uses both the alignment information in statistical translation models and translation information in a rule-based machine translation system. It includes three steps. (1) A statistical translation model is employed to perform word alignment in two directions[1] (English to Chinese, Chinese to English). (2) A rule-based English to Chinese translation system is employed to obtain Chinese translations for each English word or phrase in the source language. (3) The translation information in step (2) is used to improve the word alignment results in step (1).

A critical reader may pose the question "why not use a translation dictionary to improve statistical word alignment?" Compared with a translation dictionary, the advantages of a rule-based machine translation system lie in two aspects: (1) It can recognize the multi-word units, particularly separated phrases, in the source language. Thus, our method is able to handle the multi-word alignments with higher accuracy, which will be described in our experiments. (2) It can perform word sense disambiguation and select appropriate translations while a translation dictionary can only list all translations for each word or phrase. Experimental results show that our approach improves word alignments in both precision and recall as compared with the state-of-the-art technologies.

2 Statistical Word Alignment

Statistical translation models (Brown, et al. 1993) only allow word to word and multi-word to word alignments. Thus, some multi-word units cannot be correctly aligned. In order to tackle this problem, we perform translation in two directions (English to Chinese and Chinese to English) as described in Och and Ney (2000). The GIZA++ toolkit is used to perform statistical alignment. Thus, for each sentence pair, we can get two alignment results. We use S_1 and S_2 to represent the alignment sets with English as the source language and Chinese as the target language or vice versa. For alignment links in both sets, we use *i* for English words and *j* for Chinese words.

$$S_1 = \{(A_j, j) \mid A_j = \{a_j\},\ a_j \geq 0\}$$

$$S_2 = \{(i, A_i) \mid A_i = \{a_i\},\ a_i \geq 0\}$$

Where, $a_x (x = i, j)$ represents the index position of the source word aligned to the target word in position *x*. For example, if a Chinese word in position *j* is connected to an English word in position *i*, then $a_j = i$. If a Chinese word in position j is connected to English words in positions i_1 and i_2, then $A_j = \{i_1, i_2\}$.[2] We call an element in the alignment set an *alignment link*. If the link includes a word that has no translation, we call it a *null link*. If $k (k > 1)$ words have null links, we treat them as *k* different null links, not just one link.

Based on S_1 and S_2, we obtain their intersection set, union set and subtraction set.

Intersection: $S = S_1 \cap S_2$

Union: $P = S_1 \cup S_2$

Subtraction: $\mathrm{F} = \mathrm{P} - S$

Thus, the subtraction set contains two different alignment links for each English word.

3 Rule-Based Translation System

We use the translation information in a rule-based English-Chinese translation system[3] to improve the statistical word alignment result. This translation system includes three modules: source language parser, source to target language transfer module, and target language generator.

From the transfer phase, we get Chinese translation candidates for each English word. This information can be considered as another word alignment result, which is denoted as $S_3 = \{(k, C_k)\}$. C_k is the set including the translation candidates for the k-th English word or phrase. The difference between S_3 and the common alignment set is that each English word or phrase in S_3 has one or more translation candidates. A translation example for the English sentence "He is used to pipe smoking." is shown in Table 1.

English Words	Chinese Translations
He	他
is used to	习惯
pipe	烟斗，烟筒
smoking	吸，吸烟

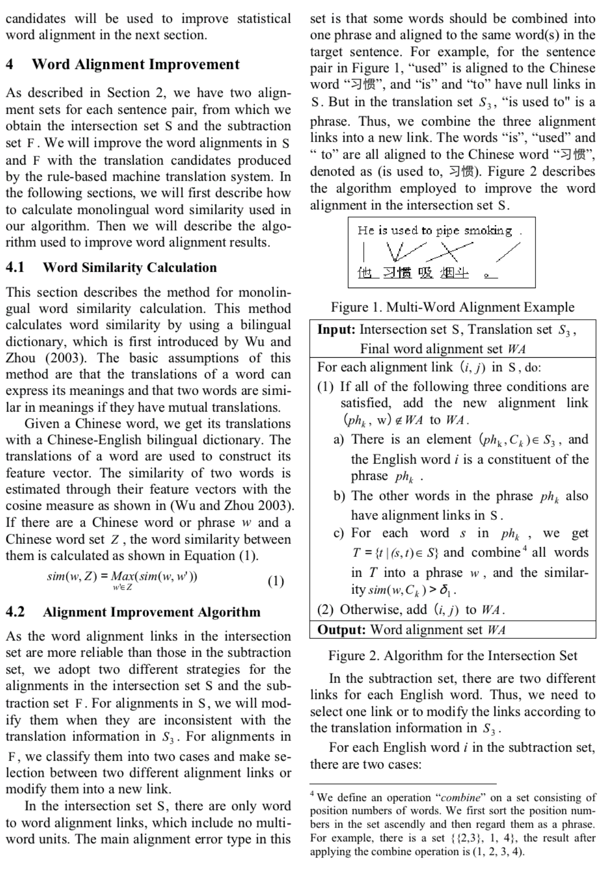
Table 1. Translation Example

From Table 1, it can be seen that (1) the translation system can recognize English phrases (e.g. is used to); (2) the system can provide one or more translations for each source word or phrase; (3) the translation system can perform word selection or word sense disambiguation. For example, the word "pipe" has several meanings such as "tube", "tube used for smoking" and "wind instrument". The system selects "tube used for smoking" and translates it into Chinese words "烟斗" and "烟筒". The recognized translation

[1] We use English-Chinese word alignment as a case study.

[2] In the following of this paper, we will use the position number of a word to refer to the word.

[3] This system is developed based on the Toshiba English-Japanese translation system (Amano et al. 1989). It achieves above-average performance as compared with the English-Chinese translation systems available in the market.



candidates will be used to improve statistical word alignment in the next section.

4 Word Alignment Improvement

As described in Section 2, we have two alignment sets for each sentence pair, from which we obtain the intersection set S and the subtraction set F. We will improve the word alignments in S and F with the translation candidates produced by the rule-based machine translation system. In the following sections, we will first describe how to calculate monolingual word similarity used in our algorithm. Then we will describe the algorithm used to improve word alignment results.

4.1 Word Similarity Calculation

This section describes the method for monolingual word similarity calculation. This method calculates word similarity by using a bilingual dictionary, which is first introduced by Wu and Zhou (2003). The basic assumptions of this method are that the translations of a word can express its meanings and that two words are similar in meanings if they have mutual translations.

Given a Chinese word, we get its translations with a Chinese-English bilingual dictionary. The translations of a word are used to construct its feature vector. The similarity of two words is estimated through their feature vectors with the cosine measure as shown in (Wu and Zhou 2003). If there are a Chinese word or phrase *w* and a Chinese word set *Z*, the word similarity between them is calculated as shown in Equation (1).

$$sim(w, Z) = \underset{w' \in Z}{Max}(sim(w, w')) \qquad (1)$$

4.2 Alignment Improvement Algorithm

As the word alignment links in the intersection set are more reliable than those in the subtraction set, we adopt two different strategies for the alignments in the intersection set S and the subtraction set F. For alignments in S, we will modify them when they are inconsistent with the translation information in S_3. For alignments in F, we classify them into two cases and make selection between two different alignment links or modify them into a new link.

In the intersection set S, there are only word to word alignment links, which include no multi-word units. The main alignment error type in this set is that some words should be combined into one phrase and aligned to the same word(s) in the target sentence. For example, for the sentence pair in Figure 1, "used" is aligned to the Chinese word "习惯", and "is" and "to" have null links in S. But in the translation set S_3, "is used to" is a phrase. Thus, we combine the three alignment links into a new link. The words "is", "used" and " to" are all aligned to the Chinese word "习惯", denoted as (is used to, 习惯). Figure 2 describes the algorithm employed to improve the word alignment in the intersection set S.

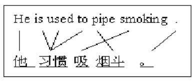


Figure 1. Multi-Word Alignment Example

```
Input: Intersection set S, Translation set $S_3$,
       Final word alignment set $WA$
For each alignment link $(i, j)$ in S, do:
(1) If all of the following three conditions are satisfied, add the new alignment link
    $(ph_k, w) \notin WA$ to $WA$.
    a) There is an element $(ph_k, C_k) \in S_3$, and the English word $i$ is a constituent of the
       phrase $ph_k$.
    b) The other words in the phrase $ph_k$ also have alignment links in S.
    c) For each word $s$ in $ph_k$, we get $T = \{t \mid (s,t) \in S\}$ and combine[4] all words
       in $T$ into a phrase $w$, and the similarity $sim(w, C_k) > \delta_1$.
(2) Otherwise, add $(i, j)$ to $WA$.
Output: Word alignment set $WA$
```

Figure 2. Algorithm for the Intersection Set

In the subtraction set, there are two different links for each English word. Thus, we need to select one link or to modify the links according to the translation information in S_3.

For each English word *i* in the subtraction set, there are two cases:

[4] We define an operation "*combine*" on a set consisting of position numbers of words. We first sort the position numbers in the set ascendly and then regard them as a phrase. For example, there is a set {{2,3}, 1, 4}, the result after applying the combine operation is (1, 2, 3, 4).

A.2 Logical output

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<TEI xmlns="http://www.tei-c.org/ns/1.0" xmlns:ns2="http://www.tei-c.org/ns/Examples">
    <teiHeader>
        <fileDesc>
            <titleStmt>
                <title>Improving Statistical Word Alignment with a Rule-Based Machine Translation
                    System
                </title>
            </titleStmt>
        </fileDesc>
    </teiHeader>
    <text>
        <front>
            <div type="abs">
                <head>ABSTRACT</head>
                <p>The main problems of statistical word alignment lie in the facts that source
                    words can only be aligned to one target word, and that the inappropriate target
                    word is selected because of data sparseness problem. This paper proposes an
                    approach to improve statistical word alignment with a rule-based translation
                    system. This approach first uses IBM statistical translation model to perform
                    alignment in both directions (source to target and target to source), and then
                    uses the translation information in the rule-based machine translation system to
                    improve the statistical word alignment. The improved alignments allow the
                    word(s) in the source language to be aligned to one or more words in the target
                    language. Experimental results show a significant improvement in precision and
                    recall of word alignment.
                </p>
            </div>
        </front>
        <body>
            <div>
                <p>WU Hua, WANG Haifeng</p>
                <p>Toshiba (China) Research &amp; Development Center 5/F., Tower W2, Oriental Plaza,
                    No.1, East Chang An Ave., Dong Cheng District
                </p>
                <p>Beijing, China, 100738</p>
                <p>{wuhua, wanghaifeng}@rdc.toshiba.com.cn</p>
            </div>
            <div1>
                <head xml:id="sec1">Introduction</head>
                <p>Bilingual word alignment is first introduced as an intermediate result in
                    statistical machine translation (SMT) (Brown et al. 1993). Besides being used in
                    SMT, it is also used in translation lexicon building (Melamed 1996), transfer
                    rule learning (Menezes and Richardson 2001), example-based machine translation
                    (Somers 1999), etc. In previous alignment methods, some researches modeled the
                    alignments as hidden parameters in a statistical translation model (Brown et al.
                    1993; Och and Ney 2000) or directly modeled them given the sentence pairs
                    (Cherry and Lin 2003). Some researchers used similarity and association measures
                    to build alignment links (Ahrenberg et al. 1998; Tufis and Barbu 2002). In
                    addition, Wu (1997) used a stochastic inversion transduction grammar to
                    simultaneously parse the sentence pairs to get the word or phrase alignments.
                </p>
                <p>Generally speaking, there are four cases in word alignment: word to word
                    alignment, word to multi-word alignment, multi-word to word alignment, and
                    multi-word to multi-word alignment. One of the most difficult tasks in word
                    alignment is to find out the alignments that include multi-word units. For
                    example, the statistical word alignment in IBM translation models (Brown et al.
                    1993) can only handle word to word and multi-word to word alignments.
                </p>
                <p>Some studies have been made to tackle this problem. Och and Ney (2000) performed
                    translation in both directions (source to target and target to source) to extend
                    word alignments. Their results showed that this method improved precision
                    without loss of recall in English to German alignments. However, if the same
                    unit is aligned to two different target units, this method is unlikely to make a
                    selection. Some researchers used preprocessing steps to identity multi-word
                    units for word alignment (Ahrenberg et al. 1998; Tiedemann 1999; Melamed 2000).
                    The methods obtained multi-word candidates based on continuous N-gram
                    statistics. The main limitation of these methods is that they cannot handle
                    separated phrases and multi-word units in low frequencies.
                </p>
                <p>In order to handle all of the four cases in word alignment, our approach uses
                    both the alignment information in statistical translation models and translation
                    information in a rule-based machine translation system. It includes three steps.
                    (1) A statistical translation model is employed to perform word alignment in two
                    directions 1 (English to Chinese, Chinese to English). (2) A rule-based English
                    to Chinese translation system is employed to obtain Chinese translations for
                    each English word or phrase in the source language. (3) The translation
                    information in step (2) is used to improve the word alignment results in step
                    (1).
                </p>
                <p>A critical reader may pose the question "why not use a translation dictionary to
                improve statistical word alignment"? Compared with a translation dictionary, the
                advantages of a rule-based machine translation system lie in two aspects: (1) It
                can recognize the multi-word units, particularly separated phrases, in the
                source language. Thus, our method is able to handle the multi-word alignments
                with higher accuracy, which will be described in our experiments. (2) It can
                perform word sense disambiguation and select appropriate translations while a
                translation dictionary can only list all translations for each word or phrase.
                Experimental results show that our approach improves word alignments in both
                precision and recall as compared with the state-of-the-art technologies.
            </p>
        </div1>
        <floatingText xml:id="footnote1" type="footnote">
            <body>
                <div>
                    <p>We use English-Chinese word alignment as a case study.</p>
                </div>
            </body>
        </floatingText>
        <div1>
            <head xml:id="sec2">Statistical Word Alignment</head>
            <p>Statistical translation models (Brown, et al. 1993) only allow word to word and
                multi-word to word alignments. Thus, some multi-word units cannot be correctly
                aligned. In order to tackle this problem, we perform translation in two
                directions (English to Chinese and Chinese to English) as described in Och and
                Ney (2000). The GIZA++ toolkit is used to perform statistical alignment. Thus,
                for each sentence pair, we can get two alignment results. We use S 1 and S 2 to
                represent the alignment sets with English as the source language and Chinese as
                the target language or vice versa. For alignment links in both sets, we use i
                for English words and j for Chinese words.
            </p>
            <p>S 1 = {( A j , j ) | A j = { a j }, a j  0 }</p>
            <p>S 2 = {( i , A i ) | A i = { a i }, a i  0 }</p>
            <p>Where, a x ( x = i , j ) represents the index position of the source word aligned
                to the target word in position x . For example, if a Chinese word in position j
                is connected to an English word in position i , then a j = i . If a Chinese word
                in position j is connected to English words in positions i 1 and i 2 , then A j
                = { i , i } . 2 1 2 We call an element in the alignment set an alignment link .
                If the link includes a word that has no translation, we call it a null link. If
                k ( k &gt; 1 ) words have null links, we treat them as k different null links,
                not just one link.
            </p>
            <p>Based on S 1 and S 2 , we obtain their intersection set, union set and
                subtraction set.
            </p>
            <p>Intersection: S = S 1  S 2</p>
            <p>Union: P = S 1  S 2</p>
            <p>Subtraction: F = P - S</p>
            <p>Thus, the subtraction set contains two different alignment links for each English
                word.
            </p>
        </div1>
        <floatingText xml:id="footnote2" type="footnote">
            <body>
                <div>
                    <p>In the following of this paper, we will use the position number of a word
                        to refer to the word.
                    </p>
                </div>
            </body>
        </floatingText>
        <div1>
            <head xml:id="sec3">Rule-Based Translation System</head>
            <p>We use the translation information in a rulebased English-Chinese translation
                system 3 to improve the statistical word alignment result. This translation
                system includes three modules: source language parser, source to target language
                transfer module, and target language generator.
            </p>
            <p>From the transfer phase, we get Chinese translation candidates for each English
                word. This information can be considered as another word alignment result, which
                is denoted as S 3 = {( k , C k )} . C k is the set including the translation
                candidates for the k-th English word or phrase. The difference between S 3 and
                the common alignment set is that each English word or phrase in S 3 has one or
                more translation candidates. A translation example for the English sentence "He
                is used to pipe smoking". is shown in Table 1. Table 1. Translation Example
            </p>
            <p>From Table 1, it can be seen that (1) the translation system can recognize
                English phrases (e.g. is used to); (2) the system can provide one or more
                translations for each source word or phrase; (3) the translation system can
                perform word selection or word sense disambiguation. For example, the word""
                pipe has several meanings such as ""tube, "tube used for "smoking and "wind"
                instrument. The system selects "tube used for "smoking and translates it into
                Chinese words " "   and " "  . The recognized translation candidates will be
                used to improve statistical word alignment in the next section.
            </p>
        </div1>
```

```
<floatingText xml:id="footnote3" type="footnote">
    <body>
        <div>
            <p>This system is developed based on the Toshiba EnglishJapanese translation
                system (Amano et al. 1989). It achieves above-average performance as
                compared with the EnglishChinese translation systems available in the
                market.
            </p>
        </div>
    </body>
</floatingText>
<div1>
    <head xml:id="sec4">Word Alignment Improvement</head>
    <p>As described in Section 2, we have two alignment sets for each sentence pair,
        from which we obtain the intersection set S and the subtraction set F . We will
        improve the word alignments in S and F with the translation candidates produced
        by the rule-based machine translation system. In the following sections, we will
        first describe how to calculate monolingual word similarity used in our
        algorithm. Then we will describe the algorithm used to improve word alignment
        results.
    </p>
    <div2>
        <head xml:id="sec4.1">Word Similarity Calculation</head>
        <p>This section describes the method for monolingual word similarity
            calculation. This method calculates word similarity by using a bilingual
            dictionary, which is first introduced by Wu and Zhou (2003). The basic
            assumptions of this method are that the translations of a word can express
            its meanings and that two words are similar in meanings if they have mutual
            translations.
        </p>
        <p>Given a Chinese word, we get its translations with a Chinese-English
            bilingual dictionary. The translations of a word are used to construct its
            feature vector. The similarity of two words is estimated through their
            feature vectors with the cosine measure as shown in (Wu and Zhou 2003). If
            there are a Chinese word or phrase w and a Chinese word set Z , the word
            similarity between them is calculated as shown in Equation (1). sim ( w , Z
            ) = Max ( sim ( w , w ' )) w '   Z (1)
        </p>
    </div2>
    <div2>
        <head xml:id="sec4.2">Alignment Improvement Algorithm</head>
        <p>As the word alignment links in the intersection set are more reliable than
            those in the subtraction set, we adopt two different strategies for the
            alignments in the intersection set S and the subtraction set F . For
            alignments in S , we will modify them when they are inconsistent with the
            translation information in S 3 . For alignments in F , we classify them into
            two cases and make selection between two different alignment links or modify
            them into a new link.
        </p>
        <p>In the intersection set S , there are only word to word alignment links,
            which include no multiword units. The main alignment error type in this set
            is that some words should be combined into one phrase and aligned to the
            same word(s) in the target sentence. For example, for the sentence pair in
            Figure 1, ""used is aligned to the Chinese word " "  , and ""is and ""to
            have null links in S . But in the translation set S 3 , "is used to&quot; is
            a phrase. Thus, we combine the three alignment links into a new link. The
            words ""is, ""used and "" to are all aligned to the Chinese word " "  ,
            denoted as (is used to,   ). Figure 2 describes the algorithm employed to
            improve the word alignment in the intersection set S . Figure 1. Multi-Word
            Alignment Example Figure 2. Algorithm for the Intersection Set
        </p>
        <p>In the subtraction set, there are two different links for each English word.
            Thus, we need to select one link or to modify the links according to the
            translation information in S 3 .
        </p>
        <p>For each English word i in the subtraction set, there are two cases:</p>
    </div2>
</div1>
<floatingText xml:id="footnote4" type="footnote">
    <body>
        <div>
            <p>We define an operation " combine " on a set consisting of position
                numbers of words. We first sort the position numbers in the set ascendly
                and then regard them as a phrase. For example, there is a set {{2,3}, 1,
                4}, the result after applying the combine operation is (1, 2, 3, 4).
            </p>
        </div>
    </body>
</floatingText>
<figure>
    <p>English Words Chinese Translations He   is used to   pipe    smoking   </p>
</figure>
<figure>
    <p/>
</figure>
```

```
                <figure>
                    <p>Input: Intersection set S , Translation set S 3 , Final word alignment set WA For
                        each alignment link  i , j ) in S , do: (1) If all of the following three
                        conditions are satisfied, add the new alignment link  ph k , w    WA to WA .
                        a) There is an element  ph k , C k )  S 3 , and the English word i is a
                        constituent of the phrase ph k . b) The other words in the phrase ph k also have
                        alignment links in S . c) For each word s in ph k , we get T = { t | (s , t )
                        S } and combine 4 all words in T into a phrase w , and the similarity sim ( w ,
                        C k ) &gt;  1 . (2) Otherwise, add  i , j ) to WA . Output: Word alignment set
                        WA
                    </p>
                </figure>
            </body>
            <back/>
        </text>
</TEI>
```

A.3 Physical output

```
<document>
<styles>
<style id="NSimSun--9" font="NSimSun" size="9"/>
<style id="SimSun--10" font="SimSun" size="10"/>
<style id="SimSun--9" font="SimSun" size="9"/>
<style id="SymbolMT--11" font="SymbolMT" size="11"/>
<style id="SymbolMT--6" font="SymbolMT" size="6"/>
<style id="SymbolMT--9" font="SymbolMT" size="9"/>
<style id="TimesNewRoman--10" font="TimesNewRoman" size="10"/>
<style id="TimesNewRoman--10B" font="TimesNewRoman" size="10" bold="true"/>
<style id="TimesNewRoman--10I" font="TimesNewRoman" size="10" italic="true"/>
<style id="TimesNewRoman--11" font="TimesNewRoman" size="11"/>
<style id="TimesNewRoman--11I" font="TimesNewRoman" size="11" italic="true"/>
<style id="TimesNewRoman--12" font="TimesNewRoman" size="12"/>
<style id="TimesNewRoman--12B" font="TimesNewRoman" size="12" bold="true"/>
<style id="TimesNewRoman--15B" font="TimesNewRoman" size="15" bold="true"/>
<style id="TimesNewRoman--6" font="TimesNewRoman" size="6"/>
<style id="TimesNewRoman--6I" font="TimesNewRoman" size="6" italic="true"/>
<style id="TimesNewRoman--7" font="TimesNewRoman" size="7"/>
<style id="TimesNewRoman--7I" font="TimesNewRoman" size="7" italic="true"/>
</styles>
<page num="1">
    <paragraph x="83.76" y="94.0077" w="427.8874" h="27.059975" seqno="2001">
        <line styleRef="TimesNewRoman--15B">Improving Statistical Word Alignment with a Rule-Based Machine</line>
        <line styleRef="TimesNewRoman--15B">Translation System</line>
    </paragraph>
    <paragraph x="230.64" y="136.26208" w="134.02309" h="7.871994" seqno="4001">
        <line styleRef="TimesNewRoman--12B">WU Hua, WANG Haifeng</line>
    </paragraph>
    <paragraph x="102.0" y="149.88202" w="391.3822" h="49.271774" seqno="4002">
        <line styleRef="TimesNewRoman--12">Toshiba (China) Research & Development Center</line>
        <line styleRef="TimesNewRoman--12">5/F., Tower W2, Oriental Plaza, No.1, East Chang An Ave., Dong Cheng ...
        <line styleRef="TimesNewRoman--12">Beijing, China, 100738</line>
        <line styleRef="TimesNewRoman--12">{wuhua, wanghaifeng}@rdc.toshiba.com.cn</line>
    </paragraph>
    <paragraph x="158.34" y="229.80206" w="44.693985" h="7.871994" seqno="5001">
        <line styleRef="TimesNewRoman--12B">Abstract</line>
    </paragraph>
    <paragraph x="82.259995" y="248.66118" w="197.01521" h="201.87206" seqno="5002">
        <line styleRef="TimesNewRoman--10">The main problems of statistical word alignment</line>
        <line styleRef="TimesNewRoman--10">lie in the facts that source words can only be</line>
        <line styleRef="TimesNewRoman--10">aligned to one target word, and that the inappro-</line>
        <line styleRef="TimesNewRoman--10">priate target word is selected because of data</line>
        <line styleRef="TimesNewRoman--10">sparseness problem. This paper proposes an ap-</line>
        <line styleRef="TimesNewRoman--10">proach to improve statistical word alignment</line>
        <line styleRef="TimesNewRoman--10">with a rule-based translation system. This ap-</line>
        <line styleRef="TimesNewRoman--10">proach first uses IBM statistical translation</line>
        <line styleRef="TimesNewRoman--10">model to perform alignment in both directions</line>
        <line styleRef="TimesNewRoman--10">(source to target and target to source), and then</line>
        <line styleRef="TimesNewRoman--10">uses the translation information in the rule-based</line>
        <line styleRef="TimesNewRoman--10">machine translation system to improve the statis-</line>
        <line styleRef="TimesNewRoman--10">tical word alignment. The improved alignments</line>
        <line styleRef="TimesNewRoman--10">allow the word(s) in the source language to be</line>
    </paragraph>
   ...
</page>
...
</document>
```

Appendix B

Whitespace covering algorithm – Java implementation

```java
/*
 * Copyright 2010-2011 Øyvind Berg (elacin@gmail.com)
 *
 *    Licensed under the Apache License, Version 2.0 (the "License");
 *    you may not use this file except in compliance with the License.
 *    You may obtain a copy of the License at
 *
 *        http://www.apache.org/licenses/LICENSE-2.0
 *
 *    Unless required by applicable law or agreed to in writing, software
 *    distributed under the License is distributed on an "AS IS" BASIS,
 *    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *    See the License for the specific language governing permissions and
 *    limitations under the License.
 */
package org.elacin.pdfextract.physical.column;

import org.apache.log4j.Logger;

import org.elacin.pdfextract.Constants;
import org.elacin.pdfextract.content.PhysicalContent;
import org.elacin.pdfextract.content.PhysicalPageRegion;
import org.elacin.pdfextract.content.WhitespaceRectangle;
import org.elacin.pdfextract.geom.FloatPoint;
import org.elacin.pdfextract.geom.HasPosition;
import org.elacin.pdfextract.geom.Rectangle;
import org.elacin.pdfextract.geom.RectangleCollection;

import java.util.ArrayList;
import java.util.List;
import java.util.PriorityQueue;

import static org.elacin.pdfextract.Constants.*;
import static org.elacin.pdfextract.geom.RectangleCollection.Direction.E;
import static org.elacin.pdfextract.geom.RectangleCollection.Direction.W;

/**
 * Created by IntelliJ IDEA. User: elacin Date: Jun 23, 2010 Time: 13:05:06
 */
public final class WhitespaceFinder {

// ----------------------------- FIELDS -----------------------------
    private static final Logger log = Logger.getLogger(WhitespaceFinder.class);

    /* min[Height|Width] are the thinnest rectangles we will accept */
    private final float minHeight, minWidth;

    /* all the obstacles in the algorithm are found here, and are initially all
        the words on the page */
    protected final RectangleCollection region;

    /**
     * State while working follows below
     */

    /* a queue which will give us the biggest/best rectangles first */
    private final PriorityQueue<QueueEntry> queue;

    /* this holds a list of all queue entries which are not yet accepted. Upon finding a new
     * whitespace rectangle, these are added back to the queue. */
    private final List<QueueEntry> holdList = new ArrayList<QueueEntry>();

    /* this holds all the whitespace rectangles we have found */
    private final WhitespaceRectangle[] foundWhitespace;
    private int                         foundWhitespaceCount = 0;

    /* the number of whitespace we want to find */
    private final int wantedWhitespaces;

// --------------------------- CONSTRUCTORS ---------------------------
    WhitespaceFinder(RectangleCollection region, final int numWantedWhitespaces, final float minWidth,
                     final float minHeight) {

        this.region       = region;
        wantedWhitespaces = numWantedWhitespaces;
        foundWhitespace   = new WhitespaceRectangle[numWantedWhitespaces];
        queue             = new PriorityQueue<QueueEntry>(WHITESPACE_MAX_QUEUE_SIZE);
        this.minWidth     = minWidth;
        this.minHeight    = minHeight;
    }

// -------------------------- PUBLIC STATIC METHODS --------------------------
    public static List<WhitespaceRectangle> findWhitespace(final PhysicalPageRegion region) {

        final long t0             = System.currentTimeMillis();
        final int  numWhitespaces = WHITESPACE_NUMBER_WANTED;
        WhitespaceFinder finder   = new WhitespaceFinder(region, numWhitespaces,
                                        region.getMinimumColumnSpacing(), region.getMinimumRowSpacing());
        final List<WhitespaceRectangle> ret  = finder.findWhitespace();
        final long                      time = System.currentTimeMillis() - t0;

        log.info(String.format("LOG00380:%d of %d whitespaces for %s in %d ms", ret.size(),
                               numWhitespaces, region, time));

        return ret;
    }

// -------------------------- STATIC METHODS --------------------------

    /**
     * Finds the obstacle which is closest to the centre of the rectangle bound
     */
    static HasPosition choosePivot(QueueEntry entry) {

        final FloatPoint centrePoint     = entry.bound.centre();
        float            minDistance     = Float.MAX_VALUE;
        HasPosition      closestToCentre = entry.obstacles[0];

        for (int i = 0; i < entry.numObstacles; i++) {
            HasPosition obstacle = entry.obstacles[i];
            final float distance = obstacle.getPos().distance(centrePoint) * 100.0f
                                   / obstacle.getPos().height;

            if (distance < minDistance) {
                minDistance     = distance;
                closestToCentre = obstacle;
            }
        }

        return closestToCentre;
    }

    /**
     * Checks whether the rectangle represented by whitespaceCandidate is empty enough to be
     *  considered a whitespace rectangle
     */
    static boolean isEmptyEnough(QueueEntry whitespaceCandidate) {

        if (Constants.WHITESPACE_FUZZY_EMPTY_CHECK && (whitespaceCandidate.numObstacles != 0)) {

            /* accept a small intersection */
            float       intersectSum   = 0.0f,
                        whitespaceArea = whitespaceCandidate.bound.area();
            final float intersectLimit = whitespaceArea * WHITESPACE_FUZZINESS;

            for (int i = 0; i < whitespaceCandidate.numObstacles; i++) {
                final Rectangle obstaclePos   = whitespaceCandidate.obstacles[i].getPos();
                final float intersectSize     = whitespaceCandidate.bound.intersection(
                                                    obstaclePos).area();
                final float     smallestArea = Math.min(obstaclePos.area(), whitespaceArea);

                if (intersectSize > smallestArea * WHITESPACE_FUZZINESS) {
                    return false;
                }

                intersectSum += intersectSize;
            }

            return intersectSum < intersectLimit;
        }

        return whitespaceCandidate.numObstacles == 0;
    }

    /**
     * This is the quality function by which we sort rectangles to choose the 'best' one first. The
     * current function bases itself on the area of the rectangle, and then prefers high ones
     */
    static float rectangleQuality(Rectangle r) {
        return r.area() * (1 + r.height * 0.25f);
    }





// ------------------------- OTHER METHODS -------------------------

    /**
     *  The main algorithm. Finds the next whitespace rectangle
     * @return A new identified whitespace rectangle
     */
    WhitespaceRectangle findNextWhitespace() {

        queue.addAll(holdList);
        holdList.clear();

        while (!queue.isEmpty()) {

            /** Place an upper bound. If we reach this queue size we should already have enough data */
            if (WHITESPACE_MAX_QUEUE_SIZE - 4 <= queue.size()) {
                log.warn("Queue too long");

                return null;
            }

            /** this will always choose the rectangle with the highest priority */
            final QueueEntry current = queue.remove();

            /**
             * If we have accepted a whitespace rectangle since this was added to the queue, we need
             *  to recalculate the obstacles it references to make sure it doesnt overlap
             */
            if (current.numberOfWhitespaceFound != foundWhitespaceCount) {
                updateObstacleListForQueueEntry(current);
            }

            /**
             * if this contains no obstacles (or just barely touches on some) we have found a
             *  new whitespace rectangle
             */
            if (isEmptyEnough(current)) {
                final WhitespaceRectangle newWhitespace = new WhitespaceRectangle(current.bound);

                /** check if we accept the whitespace rectangle or not */

                /* check whether the whitespace is connected to either an edge or an existing
                 * whitespace. if it is not, leave it in the holdList list for now */
                if (WHITESPACE_CHECK_CONNECTED_FROM_EDGE &&!isNextToWhitespaceOrEdge(newWhitespace)) {
                    holdList.add(current);

                    continue;
                }

                /* find all the surrounding content. make sure this rectangle is not too small.
                 * This is an expensive check, which is why it is done here. i think it is still
                 * correct. */
                if (WHITESPACE_CHECK_LOCAL_HEIGHT) {
                    if (isWhitespaceTooShortForSurroundingText(newWhitespace)) {
                        continue;
                    }
                }

                /* we do not want to accept whitespace rectangles which has only one or two words
                 * on each side (0 is fine), as these doesn't affect layout and tend to break up
                 * small paragraphs of text unnecessarily */
                if (WHITESPACE_CHECK_TEXT_BOTH_SIDES) {
```

```java
                    if (isWhitespaceNeedlesslySeparatingText(newWhitespace)) {
                        continue;
                    }
                }

                return newWhitespace;
            }

            /** choose an obstacle near the middle of the current rectangle */
            final HasPosition pivot = choosePivot(current);

            /**
             * Create four subrectangles, one on each side of the pivot, and determine the obstacles
             *  located inside it. Then add each subrectangle to the queue (as long as it is not too
             *  thin)
             */
            final QueueEntry[] subrectangles = splitSearchAreaAround(current, pivot);

            for (QueueEntry sub : subrectangles) {
                if (sub == null) {
                    continue;
                }

                queue.add(sub);
            }
        }

        /* if we ran out of rectangles in the queue, return null to signal that. */
        return null;
    }

    /**
     * This method provides a personal touch to the algorithm described in the paper which is
     * referenced. Here we will just accept rectangles which are adjacent to either another one
     * which we have already identified, or which are adjacent to the edge of the page.
     * <p/>
     * By assuring that the we thus form continous chains of rectangles, the results seem to be much
     * better.
     */
    final boolean isNextToWhitespaceOrEdge(final WhitespaceRectangle newWhitespace) {

        /* accept this rectangle if it is adjacent to the edge of the page */
        final float     l    = WHITESPACE_OBSTACLE_OVERLAP;
        final Rectangle wPos = newWhitespace.getPos(),
                        rPos = region.getPos();

        if ((wPos.x <= rPos.x + l) || (wPos.y <= rPos.y + l) || (wPos.endX >= rPos.endX - l)
                || (wPos.endY >= rPos.endY - l)) {
            return true;
        }

        /* also accept if it borders one of the already identified whitespaces */
        for (int i = 0; i < foundWhitespaceCount; i++) {
            final WhitespaceRectangle existing = foundWhitespace[i];

            if (wPos.distance(existing.getPos()) <= WHITESPACE_OBSTACLE_OVERLAP) {
                return true;
            }
        }

        return false;
    }

    /**
     * Finds up to the requested amount of whitespace rectangles based on the contents on the page
     * which has been provided.
     *
     * @return whitespace rectangles
     */
    List<WhitespaceRectangle> findWhitespace() {

        if (foundWhitespaceCount == 0) {

            /* first add the whole page (all its contents as obstacle)s to the priority queue */
            int           obstacleCount = region.getContents().size();
            HasPosition[] obstacles     = region.getContents().toArray(new HasPosition[obstacleCount]);

            queue.add(new QueueEntry(region.getPos(), obstacles, obstacleCount, 0));

            /* continue looking for whitespace until we have the wanted number or we run out */
            while (foundWhitespaceCount < wantedWhitespaces) {
                final WhitespaceRectangle newRectangle = findNextWhitespace();

                /* if no further rectangles exist, stop looking */
                if (newRectangle == null) {
                    break;
                }

                foundWhitespace[foundWhitespaceCount++] = newRectangle;
            }
        }

        ArrayList<WhitespaceRectangle> ret = new ArrayList<WhitespaceRectangle>(foundWhitespaceCount);

        for (int i = 0; i < foundWhitespaceCount; i++) {
            ret.add(foundWhitespace[i]);
        }

        return ret;
    }

    /**
     * Check if the whitespace rectangle is made useless by the way it separates text. see thesis
     *  text for details.
     */
    boolean isWhitespaceNeedlesslySeparatingText(final WhitespaceRectangle newWhitespace) {

        if (newWhitespace.getPos().width > 30) {
            return false;
        }

        /* decrease the size a tiny bit, so we don't include what blocked the rectangle, especially
         *   above and below */
        Rectangle                   search     = newWhitespace.getPos().getAdjustedBy(-1.0f);
        final float                 range      = 8.0f;
        final List<PhysicalContent> right      = region.searchInDirectionFromOrigin(E, search, range);
        int                         rightCount = 0;

        for (PhysicalContent content : right) {
            if (content.isText()) {
                rightCount++;
            }
        }

        if ((rightCount == 1) || (rightCount == 2)) {
            final List<PhysicalContent> left      = region.searchInDirectionFromOrigin(W, search, range);
            int                         leftCount = 0;

            for (PhysicalContent content : left) {
                if (content.isText()) {
                    leftCount++;
                }
            }

            if ((leftCount == 1) || (leftCount == 2)) {
                return true;
            }
        }

        return false;
    }

    /**
     * Check if newWhitespace is too small considering the surrounding content
     */
    boolean isWhitespaceTooShortForSurroundingText(final WhitespaceRectangle newWhitespace) {

        final List<PhysicalContent> surroundings = region.findSurrounding(newWhitespace, 8);

        if (!surroundings.isEmpty()) {
            float averageHeight = 0.0f;
            int   counted       = 0;

            for (PhysicalContent surrounding : surroundings) {
                if (surrounding.isText()) {
                    averageHeight += surrounding.getPos().height;
                    counted++;
                }
            }

            if (counted != 0) {
                averageHeight /= (float) counted;

                float u = Math.max(((PhysicalPageRegion) region).getMinimumRowSpacing(), averageHeight);

                if (u > newWhitespace.getPos().height) {
                    return true;
                }
            }
        }
```

```java

        return false;
    }

    /**
     * Creates four rectangles with the remaining space left after splitting the current rectangle
     * around the pivot. Also divides the obstacles among the newly created rectangles
     */
    QueueEntry[] splitSearchAreaAround(final QueueEntry current, final HasPosition pivot) {

        /* Everything inside here was the definitely most expensive parts of the implementation,
         *   so this is quite optimized to avoid too many float point comparisons and needless
         *   object creations. This cut execution time by a _lot_ :) */
        final int       missingRectangles = wantedWhitespaces - foundWhitespaceCount;
        final float     splitX            = pivot.getPos().x,
                        splitEndX         = pivot.getPos().endX,
                        splitY            = pivot.getPos().y,
                        splitEndY         = pivot.getPos().endY;
        final Rectangle bound             = current.bound;

        /* check which of the four possible subrectangles we want to create, and their dimensions */
        Rectangle     left      = null;
        HasPosition[] leftObs   = null;
        final float   leftWidth = splitX - bound.x;

        if ((splitX > bound.x) && (leftWidth > minWidth)) {
            left    = new Rectangle(bound.x, bound.y, leftWidth, bound.height);
            leftObs = new HasPosition[current.numObstacles + missingRectangles];
        }

        Rectangle     above       = null;
        HasPosition[] aboveObs    = null;
        final float   aboveHeight = splitY - bound.y;

        if ((splitY > bound.y) && (aboveHeight > minHeight)) {
            above    = new Rectangle(bound.x, bound.y, bound.width, aboveHeight);
            aboveObs = new HasPosition[current.numObstacles + missingRectangles];
        }

        Rectangle     right      = null;
        HasPosition[] rightObs   = null;
        final float   rightWidth = bound.endX - splitEndX;

        if ((splitEndX < bound.endX) && (rightWidth > minWidth)) {
            right    = new Rectangle(splitEndX, bound.y, rightWidth, bound.height);
            rightObs = new HasPosition[current.numObstacles + missingRectangles];
        }

        Rectangle     below       = null;
        HasPosition[] belowObs    = null;
        final float   belowHeight = bound.endY - splitEndY;

        if ((splitEndY < bound.endY) && (belowHeight > minHeight)) {
            below    = new Rectangle(bound.x, splitEndY, bound.width, belowHeight);
            belowObs = new HasPosition[current.numObstacles + missingRectangles];
        }

        /**
         * All the obstacles in current already fit within current.bound, so we can do just a quick
         *  check to see where they belong here. this way of doing it is primarily an optimization
         */
        int         leftIndex         = 0,
                    aboveIndex        = 0,
                    rightIndex        = 0,
                    belowIndex        = 0;
        final float adjustedSplitX    = splitX - WHITESPACE_OBSTACLE_OVERLAP,
                    adjustedSplitY    = splitY - WHITESPACE_OBSTACLE_OVERLAP,
                    adjustedSplitEndX = splitEndX + WHITESPACE_OBSTACLE_OVERLAP,
                    adjustedSplitEndY = splitEndY + WHITESPACE_OBSTACLE_OVERLAP;

        for (int i = 0; i < current.numObstacles; i++) {
            HasPosition     obstacle    = current.obstacles[i];
            final Rectangle obstaclePos = obstacle.getPos();

            /* including the pivot will break the algorithm */
            if (obstacle == pivot) {
                continue;
            }

            if ((left != null) && (obstaclePos.x < adjustedSplitX)) {
                leftObs[leftIndex++] = obstacle;
            }

            if ((right != null) && (obstaclePos.endX > adjustedSplitEndX)) {
                rightObs[rightIndex++] = obstacle;
            }

            if ((above != null) && (obstaclePos.y < adjustedSplitY)) {
                aboveObs[aboveIndex++] = obstacle;
            }

            if ((below != null) && (obstaclePos.endY > adjustedSplitEndY)) {
                belowObs[belowIndex++] = obstacle;
            }
        }

        final int n = foundWhitespaceCount;

        return new QueueEntry[] { (left == null) ? null : new QueueEntry(left, leftObs, leftIndex, n),
                                  (right == null)
                                  ? null : new QueueEntry(right, rightObs, rightIndex, n),
                                  (above == null)
                                  ? null : new QueueEntry(above, aboveObs, aboveIndex, n),
                                  (below == null)
                                  ? null : new QueueEntry(below, belowObs, belowIndex, n) };
    }



    /**
     * Checks if some of the newly added whitespace rectangles, that is those discovered after this
     * queue entry was added to the queue, overlaps with the area of this queue entry, and if so
     * adds them to this list of obstacles .
     */
    void updateObstacleListForQueueEntry(final QueueEntry entry) {

        int numNewestObstaclesToCheck = foundWhitespaceCount - entry.numberOfWhitespaceFound;

        for (int i = 0; i < numNewestObstaclesToCheck; i++) {
            final HasPosition obstacle = foundWhitespace[foundWhitespaceCount - 1 - i];

            if (entry.bound.intersectsAdmittingOverlap(obstacle.getPos(), WHITESPACE_OBSTACLE_OVERLAP)) {
                entry.addObstacle(obstacle);
            }

            entry.numberOfWhitespaceFound = foundWhitespaceCount;
        }
    }

// -------------------------- INNER CLASSES --------------------------
    static class QueueEntry implements Comparable<QueueEntry> {

        final Rectangle     bound;
        int                 numberOfWhitespaceFound, numObstacles;
        final HasPosition[] obstacles;
        final float         quality;

        private QueueEntry(final Rectangle bound, final HasPosition[] obstacles, int numObstacles,
                           int numFound) {

            this.bound              = bound;
            this.obstacles          = obstacles;
            this.numObstacles       = numObstacles;
            numberOfWhitespaceFound = numFound;
            quality                 = rectangleQuality(bound);
        }

        public final int compareTo(final QueueEntry other) {
            return Float.compare(other.quality, quality);
        }

        public void addObstacle(HasPosition obstacle) {
            obstacles[numObstacles++] = obstacle;
        }
    }
}
```