TU WIEN

TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

D I S S E R T A T I O N

User-Guided Information Extraction
from Print-Oriented Documents

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften

unter Anleitung von
o. Univ.-Prof. Dipl.-Ing. Dr. techn. Georg Gottlob,
Institut für Informationssysteme (E184)
der Technischen Universität Wien

eingereicht an der Technischen Universität Wien,
Falkultät für Informatik, durch

Tamir Hassan, MEng
Kolingasse 3/18
A-1090 Wien
Matrikelnummer: 0427489
geboren am 21.10.1981 in London

Wien, am 6. Mai 2010

Abstract

In recent years, a number of systems have been developed in the academic and commercial domain for *wrapping*, or user-guided information extraction, from Web sources. An important feature of Web documents is their inherent tree structure, which is used by wrapping systems such as the *Lixto Visual Wrapper* to locate instances of the data to be extracted. Because this tree structure somewhat represents the *logical structure* of the content, such wrapping methods work successfully.

This dissertation is concerned with extending these wrapping techniques to PDF documents. This is a challenging task, as the logical structure of a PDF is (usually) not explicitly available in the file. The use of *document analysis* techniques to rediscover this structure from the *layout conventions* that are used in the document's presentation is therefore a central theme of this thesis.

We present two approaches to wrapping PDF documents: the *conversion approach* and the *graph-based approach*. The conversion approach is based on an automated, structured conversion of PDF documents into HTML, which are then used as input to the *Lixto Visual Wrapper* to extract the desired information. In this approach, we place particular emphasis on detecting tables and representing them in a structured manner in HTML.

The graph-based approach represents a novel method for specifying wrapping programs directly on the physical structure of the document. Using an algorithm based on *subgraph isomorphism*, other instances of the data are found. As this approach is not reliant on the complete and accurate detection of structures in the document, it is more robust and enables a much wider range of documents to be wrapped. Because the physical structure is more intuitive for the user, this approach also enables wrapper programs to be created in a user friendly, interactive way.

Kurzfassung

In den letzten Jahren wurden sowohl im akademischen als auch im kommerziellen Umfeld mehrere Systeme für *Wrapping*, d.h. benutzergeleitete Informationsextraktion, von Webquellen entwickelt. Ein wichtiges Merkmal von Webdokumenten ist ihre inhärente Baumstruktur, welche von Wrappingsystemen wie dem *Lixto Visual Wrapper* benutzt wird, um Instanzen der zu extrahierenden Daten zu lokalisieren. Da diese Baumstruktur der *logischen Struktur* des Inhalts einigermaßen entspricht, können derartige Methoden erfolgreich funktionieren.

Diese Dissertation beschäftigt sich mit der Erweiterung dieser Wrapping-Techniken auf druckorientierte Dokumente im PDF-Format. Diese Aufgabe stellt eine große Herausforderung dar, da die logische Struktur eines PDFs (üblicherweise) nicht explizit in der Datei vorhanden ist. Der Einsatz von Techniken aus den Bereichen *document analysis* und *document understanding* bildet daher ein zentrales Thema dieser Doktorarbeit, um diese Struktur in den in der Darstellung verwendeten Layoutkonventionen wiederzuentdecken.

Zwei Ansätze für Wrapping von PDF-Dokumenten werden vorgestellt: der *Konvertierungsansatz* und der *graphbasierte Ansatz*. Der Konvertierungsansatz beruht auf der automatischen, strukturierten Konvertierung von PDF-Dokumenten in HTML, welche nachfolgend mit dem *Lixto Visual Wrapper* bearbeitet werden, um die gewünschten Daten zu extrahieren. In diesem Ansatz wird der Schwerpunkt auf die Erkennung von Tabellen und deren strukturierte Repräsentation in HTML gesetzt.

Mit dem graphbasierten Ansatz wurde eine ganz neue Methode für die Spezifikation von Wrapper-Programmen direkt auf der physischen Struktur des Dokuments geschaffen, welche mittels eines auf *Subgraphisomorphismus* basierten Verfahrens die gewünschten Daten extrahiert. Da dieser Ansatz nicht von einer vollständigen, genauen Erkennung von Strukturen abhänging ist, wird das Wrapping von einer größeren Auswahl von Dokumenten mit geringerer Fehleranfälligkeit ermöglicht. Zudem ähnelt die Graphstruktur der physischen Struktur des Dokuments, ist somit für den Benutzer intuitiver zu verstehen und erlaubt die benutzerfreundliche, interaktive Erstellung von Wrapper-Programmen.

Contents

Contents

Chapter 1

Introduction

1.1   Background

A large amount of information is published nowadays on the Web not only in HTML format, but also in *poorly structured* formats such as Portable Document Format (PDF). We use the term **poorly structured documents** to refer to documents encoded digitally in a visual form, which do not contain meta-information which explicitly describes their **logical structure** in sufficient detail to allow us to locate individual data items for information extraction purposes.

This thesis is concerned with the development of new methods for extraction of textual information from PDF documents. The PDF format is based on the printing language *PostScript*, and is what we name a **print-oriented** format. Section 1.2 describes PDF in more detail. The widespread availability of PDF documents on the Internet makes this an ideal format to investigate for data extraction purposes.

The methods described in this thesis could also be adapted to other print-oriented formats, in particular PostScript, due to the similarities between the two formats. Bitmap and vector images could also be described as unstructured, but they are less commonly used to publish business-oriented data.

In order to obtain a rough idea about what we mean by *logical structure*, consider the following questions which could be asked about a document:

- Is the document subdivided into sections? Does it contain separate articles, for example?

- How do these sections interact with each other?

- How is the text structured? Is it set in columns? Where do paragraphs begin and end?

- What is the reading order of the document?

- Are other structures, such as figures or tables, present?

- Does the document exhibit a hierarchical structure? (For example, does it contain headings and sub-headings?)

It takes very little effort for a human reader to find answers to these questions for a given document. Normally, just a quick glance at the page is required to recognize its general structure, thanks to our cognitive perception abilities. Using a combination of hereditary and acquired knowledge, we can understand the structure of even complex documents without significant effort.

In order to make such documents amenable to machine processing, this logical structure needs to be made available in machine readable form. **Document analysis and understanding** are the sub-fields of artificial intelligence which aim to model some of these cognitive processes and reconstruct the logical structure of documents based on their visual presentation, and many of the methods described in this thesis are based on document understanding techniques. Chapter 2 describes these areas in detail.

1.2   The PDF Format

Since the release of the first PDF specification in 1993, the PDF format has evolved over the past two decades to become the *de facto* standard for sharing print-oriented documents on the Web. Its success can be attributed to its roots—the PostScript page description language—which gives ultimate control over the *appearance* of documents, and ensures that this appearance is preserved regardless of whether the document is viewed on screen or paper, and regardless of operating system, installed drivers or fonts, or printer hardware. In its early years, PDF was mainly popular in desktop publishing workflow applications. Because of the popularity of PostScript at the time, competitors such as *Tumbleweed Envoy* and *Common Ground Digital Paper* soon disappeared from the market.

The advent of the World Wide Web led to a dramatic increase in the amount of documents being exchanged electronically. Unfortunately, HTML, the main format of the Web, had—and still has—the following major drawbacks:

- it is *content oriented*, meaning that the structure of the content is, or at least should be, defined explicitly in the code (although many current websites do not do this). As a hypertext document format, HTML was designed with the idea that different computing platforms would adapt the final layout depending upon screen size and configuration. This is of great benefit to data extraction applications, but . . .

- the conversion of print-oriented documents into HTML is not straightforward and is (if at all possible) time-consuming;

- the output quality of HTML documents when printed (e.g. using a web browser) is not high enough for most professional applications. Perhaps this is because HTML has its roots in the scientific community, and not the publishing community.

The print-oriented nature of PDF does, however, have its own drawbacks. At the time of its creation, nobody could have foreseen the explosive growth of PDF as a format for document interchange on the Web. In comparison to HTML, the logical structure is, in most cases, not explicitly present in the document. In this sense, PDF is closer to a graphic than a hypertext document. Whereas in HTML it is possible to parse the source code to find the locations of certain structures such as headings and tables, such machine-readable information is usually missing from PDF files. This logical structure information is critical for automatic data extraction, and is also useful for applications such as search, indexing and accessibility for disabled persons or mobile devices. Later versions of the PDF format have attempted to get around this limitation by providing support for *tagging* [Johnson 2005 (Web)], but the fact remains that the vast majority of PDF files either do not include such meta-information, or this information is not rich enough to enable us to locate the data instances to be extracted. The process by which most PDFs are created is still based on printing the document to a *virtual printer driver*, which strips the document of all its meta-information. Any logical information must then be added manually after the PDF file has been created.

The up-and-coming PDF format was around at the right time to fill this niche, enabling documents to be published to the Web as easily as sending them to the printer, with the assurance that the layout and appearance will remain the same, regardless of computing platform. To help increase takeup, Adobe soon dropped the initial charge for the *Acrobat Reader* (now simply named *Reader*) software to view PDF files. As Adobe has always published the full specification of the PDF format [Adobe Systems Inc. 2009 (Web)], third-party applications also exist for viewing and manipulating PDF files. Since 2008, PDF is a published ISO standard.

1.3   Information extraction on the Web

The motivation of this work stems from information extraction on the Web. There are a large number of business applications that rely on the mass collection, aggregation and processing of data from the Web. However, the lack of structure of web information sources presents significant difficulties to their use in information retrieval systems. In the course of the previous decade, a large number of systems have appeared, both academic and commercial, for *wrapping* this data from the Web; this means navigating the data source, semi-automatically extracting the data and transforming it into a form suitable for data processing applications. Chang et al. provide the following definition of a **wrapper** program:

> Programs that perform the task of IE [information extraction] are referred to as extractors or wrappers. A wrapper was originally defined as a component in an information integration system which aims at providing a single uniform query interface to access multiple information sources. In an information integration system, a wrapper is generally a program that "wraps" an information source (e.g., a database server or a Web server) such that the information integration system can access that information source without changing its core query answering mechanism. In the case where the information source is a Web server, a wrapper must query the Web server to collect the resulting pages via HTTP protocols, perform information extraction to extract the contents in the HTML documents, and finally integrate with other data sources. Among the three procedures, information extraction has received most attention and some use wrappers to denote extractor programs. ([Chang et al. 2006], p. 1 [1411])

There are a number of ways to classify wrapping systems. [Tatbul et al. 2001] proposes three classifications based on how the specification files are generated: manual, automatic and semi-automatic. Manual techniques require the use of a wrapper programming language such as *PiLLoW* [Cabeza and Hermenegildo 2001] or *Jedi* [Huck et al. 1998] and have fallen out of popularity in recent years, as construction and maintenance of wrappers is time-consuming and their specification language presents a significant learning curve.

Fully automatic systems generate wrappers without any user input. The *RoadRunner* approach [Crescenzi et al. 2001] employs a *matching technique* to generate a wrapper based on several examples of documents from the same class and therefore does not require any labelled examples. *ExAlg* [Arasu and Garcia-Molina 2003] works in a similar fashion. The *MDR* (Mining Data Records) approach uses string-matching

[Liu et al. 2003] and partial tree alignment [Zhai and Liu 2005] techniques to locate examples of repeating data records on a Web page. The work of [Embley et al. 1999] uses an ontological model for a specific domain to extract data from arbitrary Web pages about this domain. In general, the advantage of fully automatic approaches is that they can be applied to any arbitrary document that has not been encountered before. However, they are neither as precise, nor can they extract such detailed information as semi-automatic approaches.

Semi-automatic systems allow the generation of wrappers based on user input. These approaches can broadly be split up into systems based on **wrapper induction** and systems based on **wrapper specification**. There is currently a wide range of *wrapper induction* systems in the literature, such as *WIEN* [Kushmerick et al. 1997], *SoftMealy* [Hsu and Chang 1999] and *Stalker* [Muslea et al. 1998], which use machine learning techniques to generate a wrapper program from a set of training examples provided by the user. In systems based on *wrapper specification*, the user is directly involved in programmatically creating the wrapper. In contrast to manual systems, wrappers are usually generated based on a few examples in an intuitive and interactive fashion. Examples of such systems include *Lixto* [Baumgartner et al. 2001], *DEByE* [Laender et al. 2002a] and *Wargo* [Raposo et al. 2002].

Semi-automatic approaches have also been referred to as *supervised* approaches in the literature. In order to clarify the distinction between approaches using *supervised learning* or *wrapper induction* and approaches based on *wrapper specification*, we refer to the latter as **user-guided** wrapper generation. These approaches could be seen as combining the power and precision of manual approaches with the user-friendliness and shallow learning curve of automatic systems.

Another distinguishing feature of wrapping systems is the underlying structure or representation of the Web content on which the extraction is performed. Whereas older systems such as Stalker work on a flattened representation of the HTML content, newer approaches such as Lixto work directly on the DOM tree, and are therefore arguably better suited for HTML pages. The MDR approach described above features both approaches. A third class of systems uses spatial (and other) relations to locate the desired data instances. Examples include the work of [Krüpl et al. 2005], Embley et al. (as described above) and our graph-based approach to wrapping PDF documents [Hassan 2009c], which is described in Chapter 6 of this dissertation.

Many of these approaches such as Lixto, Wargo and *Fetch* (based on Stalker), which started out in the academic domain, are now fully fledged commercial products. In this thesis we will concentrate on the Lixto suite of products, which comprise an interactive wrapping system to extract data from HTML files on the Web, and are originally a product of research at our institute.

In this section, we have provided a brief overview of web information extraction approaches related to our task of wrapping PDF documents. A widely cited taxonomy for characterizing Web data extraction tools is provided in [Laender et al. 2002b]; this survey also provides an excellent summary of work in this field up to its publication date. More recent summaries include [Chang et al. 2006; Ferrara and Fiumara 2010; Baumgartner et al. 2009].

The *Lixto Visual Developer* enables a non-expert user to create wrapping programs in a predominantly visual and interactive fashion by clicking on example instances on a visual rendition of the web page. In the background, the program locates the selected data on the HTML parse tree, a hierarchical representation of the page's source code. The path to the data is then generalized so that repeated instances of the data are found. These instances are then displayed to the user. The user can then fine-tune the selected data by adding or removing logical conditions. The system then generates a program in *Elog* [Baumgartner et al. 2001], a declarative logic-based language, to automatically extract this data from similarly structured sources, or from sources whose content changes over time.

The approaches described in this section, including Lixto, are primarily designed with HTML documents in mind. In this thesis, we are concerned with the following research question:

> **How can user-guided information extraction methods be applied to unstructured documents such as PDF files?**

Unfortunately, this is a difficult task. As mentioned in Section 1.2, HTML documents are inherently structured—and this structure somewhat represents the document's *logical* structure—enabling us to use this structure to locate wrapping instances. In most PDF files, this logical structure is not explicitly present, and we therefore need to resort to approaches based on document analysis to rediscover this structure and allow us to locate the data to be wrapped.

1.4   Related approaches to wrapping PDF documents

The approaches discussed in Section 1.3 are all geared towards information extraction from HTML documents. Wrapping from PDF, or other types of print-oriented documents, is a much less researched task. When we began addressing this issue in 2005, we knew of no prior work on user-guided data extraction from PDF. In 2006, the theoretical foundations of a method for wrapping PDF documents were proposed in

[Flesca et al. 2006], followed by a publication in 2008 with experimental results [Fazzinga et al. 2008]. In this method, a wrapper is defined as a set of hierarchically nested fuzzy logic conditions.

This method can be seen as being analogous to the graph-based approach described in Chapter 6, as the wrapping is performed directly on a set of blocks (which the paper names *tokens*) derived from the geometric structure of the document. The preprocessing steps necessary to obtain these tokens are not described. Compared to the graph-based approach, the wrapper specification language is more expressive, as potentially *any* conditions between tokens can be defined. However, because of this, defining a wrapper is also a much more complex task. In contrast, the underlying structure of the document in the graph-based approach is more intuitive, and creating a wrapper requires little more than selecting a single example instance.

Both systems have published experimental results, which show their applicability to a wide range of data extraction tasks. Both systems can also be extended by defining new predicates or by linking them to an ontology. An interesting feature of Flesca et al.'s system is the use of fuzzy logic to cope with the inaccuracies of document analysis and increase overall robustness. It would be interesting to also see test results of the system using only regular bivalent logic to see the extent of the improvement provided by fuzzy logic. In a similar way, we propose future work in the investigation of *error tolerant* graph matching methods Section 6.7 to improve the robustness of our methods.

In the document analysis field, the table analysis method proposed in [Green and Krishnamoorthy 1995], in which predefined table structure models are overlaid on a scanned image of a page, is also worthy of mention, as it could also be seen as a sort of wrapping system.

The work in this thesis deals with several fields of research, most notably *document analysis and understanding*, but also *table structure recognition.* Related work in these fields is described in Sections 2.2 and 4.2 respectively.

1.5 Contributions of the thesis

In this dissertation, two new approaches to user-guided data extraction from PDF documents are proposed and investigated, which will hereafter be referred to as the *conversion approach* and the *graph-based approach*:

- The **conversion approach** is based on a conversion from PDF into (X)HTML, in which structures such as lines, paragraphs, headings and tables are automat-

ically detected. The resulting XHTML file is then used as input to the Lixto Visual Wrapper, and wrapping proceeds in the same way as on a web page.

- The **graph-based approach** is designed to overcome some of the drawbacks of the intermediate HTML representation by working *more directly* on the document's visual structure. We extend our graph-based representation originally developed to aid page segmentation and locate data instances using an algorithm based on *subgraph isomorphism*.

Before work on this dissertation began, the problem of wrapping print-oriented documents had not been addressed before in scientific literature, to our knowledge. The graph-based approach was first proposed in the poster paper [Hassan and Baumgartner 2006]. Together with [Flesca et al. 2006], these represent the first systems for wrapping PDF documents which exploit the document's physical structure.

The conversion and graph-based approaches have been implemented as prototypes and are evaluated in this thesis. Whereas the conversion approach is a novel application of existing concepts, methods and algorithms, the graph-based approach presents a completely new, innovative method for supervised, interactive data extraction for poorly structured documents, which is much better suited to the intricacies of locating data items from print-oriented formats such as PDF.

Both of these approaches are underpinned by methods and algorithms in *document analysis and understanding*, and this topic forms the second main focus of the thesis. We have built a prototyping framework, *PDF Analyser*, to open a PDF file, extract page items as objects, perform segmentation, detect tables and display the output visually, overlaid on a bitmap rendition of the page. This output can then be written to XHTML (for the conversion approach) or displayed as a graph (for the graph-based approach). A number of novel methods and algorithms have arisen from this work, many of which have also been published in international conferences and workshops:

- an algorithm for table detection and structure recognition from PDF documents is described in Chapter 4. This is based on our paper at ICDAR 2007 [Hassan and Baumgartner 2007];

- the above algorithm was also compared to another system in the literature [Ruffolo and Oro 2009]. Several issues were encountered in the evaluation and comparison process, which made it difficult to ensure a fair comparison of both systems. These issues are documented in Section 4.4 and Appendix A. A publication about these issues is also planned in a future workshop;

- new techniques have been developed for working with PDF files at the *object level*; we have developed a method which parses the PDF file and determines
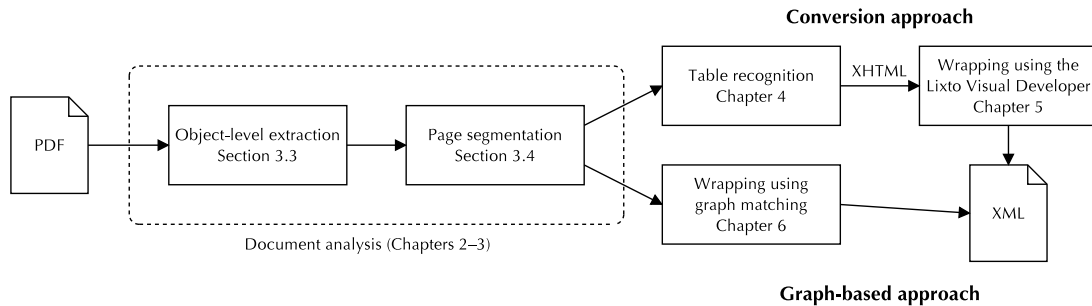
Figure 1.1: Outline of the two approaches to wrapping from PDF

which objects are important for document analysis and distinguishes between *structural* and *graphic objects* (see Section 3.3); Section 3.4 presents our *ordered-edge segmentation algorithm*, an efficient, robust, bottom-up approach to page segmentation, which takes advantage of the fact that higher-level knowledge about the page is only available towards the end of the segmentation process, and is shown to provide very good results, even on complex documents. Both of these document analysis techniques were published in our DocEng 2009 paper [Hassan 2009a];

- the graph-based approach to wrapping (Chapter 6) has been shown to perform accurately and be applicable to a variety of application scenarios. The approach was first proposed in a poster paper at WWW 2006 [Hassan and Baumgartner 2006] and the algorithm and experimental results were published at ICDAR 2009 [Hassan 2009c]. The prototype system *GraphWrap* was presented at the stand of the Austrian Computer Society at CeBIT 2009 and demonstrated at DocEng 2009 [Hassan 2009b].

1.6 Chapter summary

The conceptual structure of both wrapping approaches, including the relevant chapters and sections for each stage of wrapping process, is shown in Figure 1.1. As both approaches are underpinned by document analysis techniques, their first stages of processing are identical, and these steps are documented in Chapter 3. This chapter also introduces the *PDF Analyser* framework, which is used for prototyping and visualization of results throughout the thesis. Chapter 2 introduces the fields of *document analysis* and *document understanding* and provides an overview of related work.

The next two chapters are concerned with the conversion approach to wrapping: Chapter 4 describes our work on *table detection and structure recognition*, an essential

stage in this approach. An experimental evaluation of our algorithm is provided, and our system is compared with another system in the literature [Ruffolo and Oro 2009]. Chapter 5 introduces the *Lixto Visual Developer* and demonstrates by means of an example how the conversion approach can be used to wrap tabular data from PDF files. An appraisal of this approach is provided.

Chapter 6 introduces the graph-based approach to wrapping, including a summary of related work, a technical description of our algorithm and experimental results on three examples of wrappers that were generated by the current prototype system, *GraphWrap*. The GUI, which is built upon the *PDF Analyser* framework described earlier, is introduced, which enables a non-expert user to interactively generate such wrappers in minutes.

Finally, Chapter 7 presents a comparison of both wrapping approaches and summarizes the general directions in which this work could be developed further. Further work relating specifically to the graph-based approach is also proposed in Section 6.7.

Chapter 2

Document analysis and understanding

In this chapter, we introduce the research areas of *document analysis* and *document understanding* and describe related work in both fields. Based on an analysis of this work, we draw conclusions which influence the design of our prototype system described in the following chapter.

2.1 Introduction

A human reader is able to look at an arbitrary page and intuitively recognize its structure—the boundaries of individual structuring elements such as headings, paragraph text and images and whether these objects are nested or somehow hierarchically structured; the reading order of text areas on the page—and can also determine more complex relationships from data displayed in tabular form. Through both nature and nurture, we are able to detect these structural relationships effortlessly using the various **layout conventions** that have been used in the document's presentation. These elements together make up the **logical structure** of the document, which is of prime importance for a large number of document processing tasks, not least for information extraction.

In the past few decades, document processing has become a widely researched field, both in academia and, more recently, in the commercial software environment. In this period, the terminology used in the literature has somewhat varied. In this thesis, we will use the terms *document analysis* and *understanding* as defined by Klink et al., whose definitions we find correspond to those most commonly used in recent literature:

> Document processing can be divided into two stages: document analysis and document understanding. A document has two structures, namely the geometric (layout) structure and the logical structure. Extraction of the layout structure from a document is referred to [as] document analysis. Mapping the layout structure into a logical structure is referred to [as] document understanding. ([Klink et al. 2000], p. 1)

To clarify, we refer to **document analysis** as the transformation of a document from its source form (such as PDF or paper) into an initial **geometric** (or **physical**) **structure** in machine readable form, where each page is represented as a set of low-level blocks with Cartesian coordinates. These blocks generally represent areas, or *zones*, which are somehow *visually distinct* or *separate* from each other, for example due to a different font being used or due to separation by whitespace or ruling lines. From this representation, geometric relationships, such as *adjacency*, can easily be deduced.

In order to form these blocks, a **segmentation algorithm** is used to segment the page (or "cluster" it bottom-up) into an initial set of blocks suitable for further processing. It is worth noting that the precise *granularity* of these blocks is not defined at this stage and may vary depending on the source document format, the segmentation algorithm being used and the visual content of the document. For example, if extra whitespace is present between paragraphs of text, many segmentation algorithms will likely output these paragraphs as separate blocks. If no extra whitespace is present (only indentations, for example), the paragraphs will be merged together. Generally, *undersegmentation*, i.e. when two logically separate objects (e.g. two separate columns of text) are clustered together, results in extensive postprocessing operations being necessary and should therefore be avoided where possible.

Document understanding is the further processing of these initially segmented blocks to reconstruct logical structure information about the page. From the perspective of dealing with scanned images of documents, Haralick describes logical (page) structure recognition, or document understanding, as:

> Logical Page Structure [recognition] involves determining the type of page (page classification) assigning functional labels to each block of the page, and ordering the text blocks according to their read order. ([Haralick 1994], p. 4 [388])

Tsujimoto and Asada provide the following definition of logical structure:

> . . . the logical structure describes the configuration of articles and the semantic relationship between components of the articles. Reading order

> can be derived from the logical structure. We define the transformation of a geometric structure into a logical structure as document understanding. It should be noted here that the reverse transformation is not unique, because a logical structure corresponds to a variety of geometric structures. ([Tsujimoto and Asada 1990], p. 2 [552])

Whereas the main goal of the document *analysis* part of the process has always remained the same—to segment the pages of the document into blocks—there is a great variation in the particular *logical* structures that are discovered by the various document processing systems reported in the literature. Some systems, such as [Hadjar et al. 2004], restrict themselves entirely to geometric structures and do not extract any logical information at all. Many document-generic systems working on scanned images, such as [Aiello et al. 2002], categorize the blocks and attempt to find the reading order between these blocks. Other systems, such as [Rus and Summers 1997], do not restrict themselves to flat structures and attempt to discover *hierarchical* structures within the document.

Aiello et al. also distinguish between *specific* and *generic* document processing systems, the latter accepting a much broader class of documents as input, but returning only generic structuring elements that are common to most documents. Common classes of documents addressed by specialized systems include forms (e.g. [Cesarini et al. 1998]) and tables. In this section, we concentrate on systems designed for a broad class of documents. Section 4.2 describes systems which have been designed to detect and understand the structure of tables.

Although document analysis and understanding have traditionally been viewed as separate processes, it is often a grey area whether certain derived structures are *geometric* or *logical*. For example, paragraphs, which are logical structuring elements, often correspond to the distinct *physical* blocks returned by the segmentation algorithm. In a document, physical and logical knowledge are inherently interlinked and there is no clear boundary between document analysis and document understanding. This applies even more to tables, where the grouping of individual cells into rows and columns (which we refer to as *table structure understanding*) could be seen as a *logical* step. But in fact, the row-and-column structure of a table is also very much a physical structure, and researchers working in the area of *table interpretation* go one step further and aim to find out the abstract logical relations that a table presents. With this information, the same table can even be presented differently. This is described in more detail in Section 4.1.

In the literature, the terminology is not always consistent. Earlier publications from the OCR community working on scanned images generally used the terms *document image analysis* and *document image understanding*, before the term *image* was

dropped when other input formats started being investigated. Sometimes, one of these terms is used to refer to the entire process. For example, Schürmann et al. use the term *document analysis* to refer to both the analysis and understanding steps of document processing:

> Document analysis aims at the transformation of any information presented on paper and addressed to human comprehension into an equivalent symbolic representation accessible to any kind of computer information processing. ([Schürmann et al. 1992], p. 1 [1101])

Finally, it is also important to raise the distinction between the logical structure and the *semantic* structure of a document: the term *logical structure* relates to how the individual elements functionally relate to each other—their purpose in the context of the document's structure. It is not concerned with the meanings of individual words or the subject matter of the document itself.

The diagram in Figure 2.1 shows how a document can be represented in a number of different *levels of abstraction*, from the top-level semantic view in the author's mind, before a single word is put to paper, to the finished product; a bitmap image on the printed page. It is worth noting that PDFs are already at a higher level of abstraction than scanned images. The level in which common word processing and DTP packages (such as *Microsoft Word* or *Adobe InDesign*) represent a document lies somewhere in between. Because of the ubiquity of the word processor today, we believe that the design of everyday software such as Word has also somewhat influenced the development of document understanding systems.

The process of document analysis and understanding can therefore be seen as the reverse of the document authoring process; we are trying to rediscover the logical structure from the document's physical layout. Aiello et al. provide the following definition of document analysis (note that, as with Schürmann et al., they also use the term *document analysis* to refer to the entire process of document analysis and document understanding):

> Document analysis can be viewed as reversing the process of document authoring. It is therefore important to consider the choices an author makes. In the creation process, the author starts with a rough idea about the content. The author then structures his/her thoughts by considering the logical organization of the material, e.g., dividing the material in chapters and deciding on the intended reading order. When the final digital document is printed on paper, the underlying logical structure of the document is obscured by the actual layout conventions. The author

Document authoring

Document analysis
and understanding

Semantic knowledge

Document structure
knowledge

Typical word processors
and DTP applications

Low-level geometric
structure *(e.g. T$_E$X)*

Page description language
(e.g. PDF, PostScript, PCL)

Bitmap image
(or printed page)
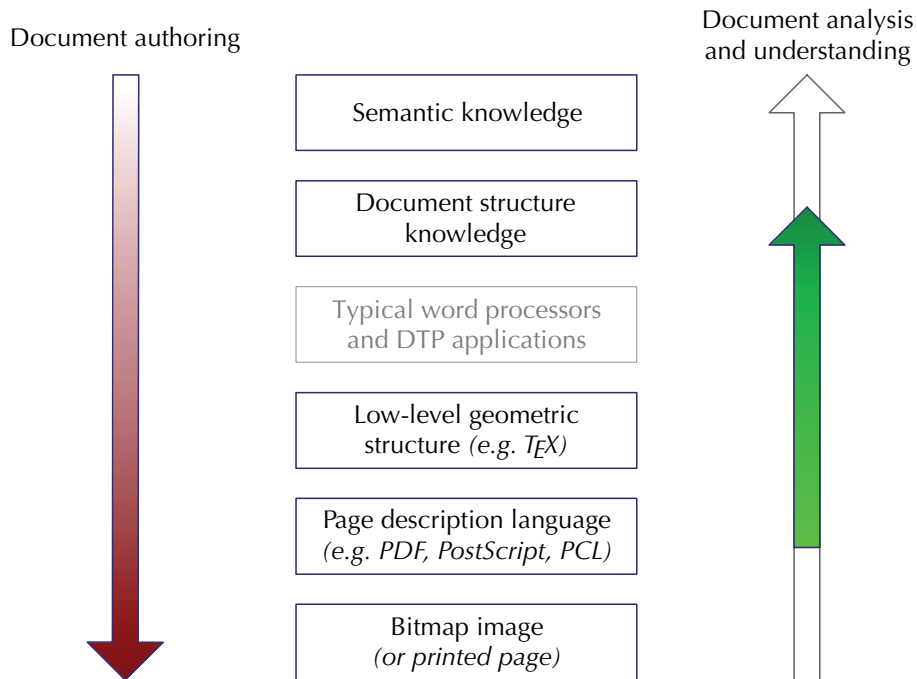
Figure 2.1: Document representation hierarchy

has, however, the possibility to encode some of the logical information us-
ing layout typesetting conventions, e.g., by using a specific font size, style
and arrangement on the page. Therefore, the layout structure of a printed
document carries, besides the artistic message, some information about
the logical structure. ([Aiello et al. 2002], p. 1)

The remainder of this chapter gives an overview of related work in the literature on
document analysis and understanding.

2.2   Related work

In this section, we discuss existing approaches to document analysis and understand-
ing. As the format of the input data plays a vital role in the design of these approaches,
we group them by the input format they accept. We concentrate on methods that are
applicable to a broad class of documents and place particular emphasis on the *doc-
ument analysis* phase of the process as per our definition in Section 2.1. Please note
that the topic of table recognition is covered in detail in Section 4. In this section,
we only briefly mention some table recognition systems from the perspective of their
approaches to document analysis (in particular, page segmentation).

2.2.1 Classical bottom-up pixel-based techniques

The term *document (image) analysis* has its origins in the OCR community, and there is a huge amount of literature describing document analysis systems that work on scanned images of pages. Examples of such systems are [Aiello et al. 2002; Altamura et al. 2001; Schürmann et al. 1992; Nagy et al. 1992; Shari 1986]. The vast majority of such systems use a *bottom-up* segmentation process: the image is first *deskewed* and *binarized* (or *thresholded*), and segmentation then is performed using pixel-based operations such as variants of the *run-length smoothing algorithm* (RLSA) [Wong et al. 1982] followed by *connected component analysis.*

Bottom-up techniques are very sensitive to local formatting changes in the document. As stated in [Randen and Husøy 1994] (p.1), "the run length smoothing method is sensitive to font-size, character spacing, line and column spacing and to some extent the orientation of the document." The run-length smoothing algorithm effectively segments the blocks using a given distance threshold. The ideal value of this threshold can depend on scanning resolution, relative darkness of the original print, as well as the type of document itself.

2.2.2 Top-down projection profile based methods

Earlier segmentation algorithms include top-down methods such as the *recursive X-Y cut* approach [Nagy and Seth 1984; Ha et al. 1995]. In this method, the page is recursively "cut" in half across a *visually salient* boundary, usually whitespace or a ruling line. Haralick et al. use horizontal and vertical projection profile methods such as the *whitespace density graph* (see Figure 2.2) to look for these salient divisions. Using heuristics, the *most prominent* division (e.g. widest peak in the graph) is found. In order to speed up computation, this method can be approximated by using bounding boxes of connected components, which are assumed to be filled uniformly with black pixels.

The main weakness of this approach is that it is not able to segment all page layouts completely. Specifically, the page layout must be *X-Y decomposable*. To illustrate this weakness, the results of our initial experiments with the X-Y cut algorithm and a bottom-up approach are shown in Figures 2.3 and 2.4. The main advantage of the X-Y cut algorithm is that it inherently generates a hierarchical structure, which via a number of relatively simple merging operations can be made to correspond to the logical structure of the document.

[Tsujimoto and Asada 1990; Ishitani 2003; Déjean and Meunier 2006] perform a *hierarchical* decomposition of the document image. Whereas Tsujimoto and Asada use
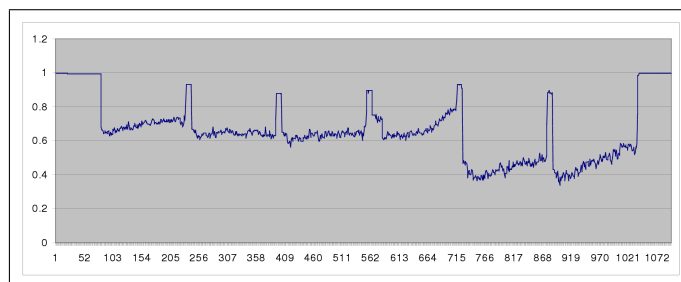
Figure 2.2: An example of the horizontal *whitespace density graph* of a newspaper page
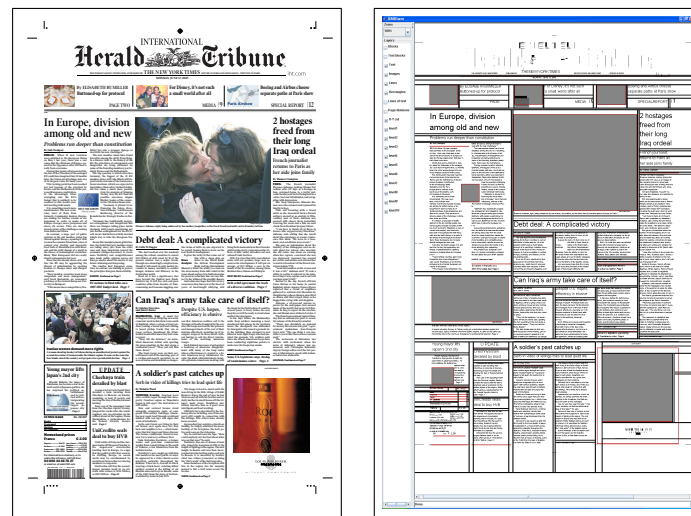
Figure 2.3: Front page of the *International Herald Tribune* newspaper (left) with its successful top-down segmentation (right)
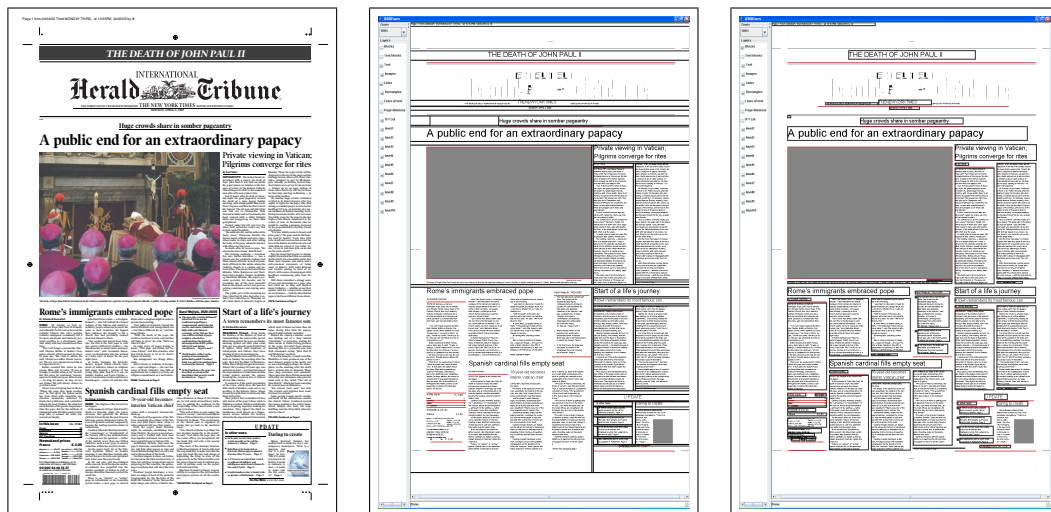
Figure 2.4: Front page of another issue of the *International Herald Tribune* (left), whose layout is not *X-Y decomposable*. Top-down segmentation (centre) fails to completely segment the page (the bottom-left quadrant is not segmented), whereas bottom-up clustering (right) succeeds

a more complex tree-based technique (which also copes with non-X-Y-decomposable layouts), the approaches of Ishitani and of Déjean and Meunier use techniques based on the X-Y cut algorithm. Ishitani introduces a technique, termed an *exceptional X-Y cut*, to successfully decompose layouts which are not X-Y decomposable. Using the X-Y decomposition, [Meunier 2005] also determines the reading order. Typically, these techniques are useful in more complex layouts, such as newspapers, which contain several articles on the page arranged in a hierarchical structure. The system proposed in [Hadjar et al. 2004] also groups individual blocks into articles before performing reading order detection, although the precise method is not described.

2.2.3 Manhattan layout

Bottom-up segmentation approaches are generally not restricted to any particular types of layout. In fact, they are theoretically able to segment any arbitrary two-dimensional shapes. However, as documents by convention are generally composed of rectangular blocks, most document analysis systems restrict themselves to "Manhattan" or "near-Manhattan" layouts, as shown in Figure 2.5. We have also taken this approach in developing our segmentation algorithm as described in Section 3.4. Haralick provides the following definition of **Manhattan layout**:

> A Manhattan page layout is one where the regions of the page layout are all rectangular and the rectangles are in the same orientation. Hence after an appropriate page rotation the sides of the rectangles will all be either horizontal or vertical. Furthermore, each pair of rectangles either is mutually exclusive or one contains the other. ([Haralick 1994], p. 1 [386])

Algorithms designed for Manhattan layouts generally fail when run on rotated, or *skewed* documents. This problem is a common occurrence in scanned documents, and there is an extensive body of literature that deals just with the problem of *skew detection* (a good summary can be found in [Nagy 2000]). Fortunately, for the digitally generated PDF documents that we encounter, this is not an issue.

2.2.4 Techniques for ASCII documents

A number of publications report on systems that process documents in ASCII format (and similar fixed-width text formats). As the ASCII format allows only a very restricted range of layout conventions to be used, some of which are specific to this format, such techniques cannot be applied directly to PDF documents. The work in [Rus and Summers 1997] uses a bottom-up method to merge lines of text, as obtained

Figure 2.5: The example on the left has a strict Manhattan layout. The layout on the right is what we term *near-Manhattan*: because of the inset graphic, the conditions for strict Manhattan layout have been breached, but bottom-up segmentation can still be performed

from an OCR package, into a hierarchical structure based on repeating indentation patterns. This technique assumes no prior knowledge about the formatting conventions being used. Multi-column and more complex layouts are not addressed, and are a rarity in this format. This paper also describes techniques for table detection and graphics recognition.

Kieninger et al. report on systems which detect tables in ASCII documents [Kieninger 1998; Klink et al. 2000; Kieninger and Dengel 1998b]. Their work is described in more detail in Section 4.2. They describe a bottom-up segmentation algorithm based on *vertical neighbourhoods:* due to what we term the *brickwork effect* (see Section 3.4.2), entire blocks of text can be built up this way. Because of the robustness of their method and its ability to also cope with layouts which are not strictly Manhattan, we have used some of their techniques in the development of our own segmentation algorithm (see Section 3.4).

2.2.5 Systems for analysing PDF documents

More recently, the PDF format has started to gain attention from researchers in document analysis. The bottom-up segmentation techniques using pixel-based operations as described at the beginning of this section could, of course, also be applied to a bitmap rendition of PDF, as in [Hadjar et al. 2004]. Compared to a scanned image, the rendition of a PDF is much "cleaner"; there is no noise and problems such as find-

ing an appropriate threshold for binarization and deskewing do not exist. However, in developing our system we have taken the view that analysing images of PDF files where the object data is available is taking a step backwards: the rasterization process can still introduce errors and creates unnecessary processing overhead. Furthermore, additional information created during creation of the PDF is lost (see Section 3.7).

A notable early publication which deals with PDF files is [Lovegrove and Brailsford 1995]. Here, segmentation is performed bottom-up using grouping algorithms on text extracted directly from the PDF data using the Acrobat SDK. [Anjewierden 2001] describes a method in which text and graphic objects are extracted from the PDF using a method built on top of the *xpdf* library, which includes some segmentation heuristics. [Déjean and Meunier 2006] uses a similar method for extracting the low-level objects from PDF. Unfortunately, neither paper describes in detail how the low-level text and graphic instructions are processed to generate the resulting objects. Our paper [Hassan 2009a] is intended to fill this gap in the literature. [Hadjar et al. 2004] also introduces a system for analysing PDF, but uses a bitmap image to perform segmentation. The text is obtained using a library[1] and matched at a later stage to the results of the layout analysis.

[Chao and Fan 2004] describes a hybrid method in which a combination of object-level and bitmap processing is used: text and image objects are obtained directly from the PDF code, whereas lines and vector objects are obtained from a bitmap image. A bottom-up segmentation algorithm, which works on rectangular text blocks obtained from the PDF, is described in detail, but, as with the above two papers, this paper is also rather short on details of how the initial objects are obtained from the PDF file.

[Futrelle et al. 2003] describes a system for graphics recognition from PDF. Here, the *Etymon PJ Tools* library [Etymon Systems Inc. 2009 (Web)] is used to obtain the graphics primitives in object form. Of course, for this application, the extracted information is at a much finer granular level than what we require for document analysis.

Today, document analysis of PDF files is still an ongoing research topic, and two research groups are particularly active in this field. The group led by Rolf Ingold in Fribourg, Switzerland, continues to publish methods to convert PDF documents into a structured XML format and represent them in such a way that virtually no information from the original file is lost [Bloechle et al. 2009; Rigamonti et al. 2005]. The group headed by David Brailsford in Nottingham is working on restructuring PDF files using *component object graphics* (COGs) to improve repurposability [Bagley et al. 2003]. In a similar way to the methods described in Section 3.3, their *COG Extractor*

---

[1]The paper analyses the results of several PDF extraction libraries (but not PDFBox, the library which we use here, as it was then at a very early stage of development), but does not state which library is used.

[Bagley 2006] parses the PDF content stream directly and groups the instructions to represent logical objects.

2.2.6 Document analysis as a computer vision task

All the methods described in this chapter up to now can be said to have taken a relatively *ad hoc* approach to document analysis, segmenting the page predominantly using fragments of knowledge, or *rules*. For example, the bottom-up approaches cluster blocks together using only low-level knowledge, i.e. the smallest pixel distance between the two blocks, to decide whether the blocks belong to the same segment. Top-down approaches use higher-level knowledge, such as rivers or whitespace of a minimum width, but ignore information at the lower levels. Because document layout generally follows rigid rules, which are in many cases known *a priori*, such approaches are able to produce satisfactory results on a wide range of documents.

However, if we consider the ultimate goal of document analysis research to be a segmentation algorithm that works successfully on *all* documents, these approaches are inherently limited, as they do not model the human vision process accurately enough. When a human reader analyses a document, several processes operate concurrently at multiple levels of granularity to allow him to understand the document more accurately. This is particularly important for more complex layouts or cases where layout conventions have been violated (such as poorly or non-professionally typeset documents), which cause considerable problems for current document understanding systems but remain understandable to humans. Already in 1992, Schürmann et al. wrote on the necessity of analysing multiple granular levels:

> It should be made clear from the beginning, however, that document analysis is only in simple cases suited for pure straightforward sequential operation. The document analysis task must be structured into several levels of interpretation and requires a combination of bottom-up and top-down approaches. At the lower levels ambiguities are quite frequent which can be resolved only at higher levels. ([Schürmann et al. 1992], p. 1 [1101])

In the computer vision domain, the problem of image segmentation is often defined as finding a solution that is in some way *optimal* in respect to several models at different levels of granularity, which conflict and complement each other. For example, Tu and Zhu state that:

> The objective of image segmentation is to parse an image into its constituent components. The latter are various stochastic processes, such

> as attributed points, lines, curves, textures, lighting variations, and deformable objects. Thus, a segmentation algorithm must incorporate many families of image models and its performance is upper bounded by the accuracy of its image models . . .
>
> . . . Real world images are fundamentally ambiguous and our perception of an image changes over time. Furthermore, an image often demonstrates details at multiple scales. Thus, the more one looks at an image, the more one sees. ([Tu and Zhu 2002], p. 1 [657])

Perhaps because of their relatively high implementational and computational complexity, such approaches have been slow to find their way into document analysis systems. [Ishitani 1999] describes one such approach, which uses a framework based on *emergent computation*, which supports flexible low-level interactions between four subsystems or *agents*, which cooperate with and compete against each other. Through these interactions, higher-level knowledge emerges.

When a human views a document from a distance, he can usually understand its overall structure using spatial and **textural** cues, even if the text is too small to read. This notion is used in [Randen and Husøy 1994; Jain and Zhong 1996], which describe texture-based approaches to segment page images.

The problem of combining knowledge on several granular levels was also addressed in the table structure recognition algorithm presented in Wang's doctoral thesis [Wang 2002], which uses a probabilistic model at several levels of granularity to obtain high detection rates.

2.2.7 Commercial software

A number of commercial off-the-shelf packages make use of some of the techniques described here. Apart from OCR software, this includes conversion programs that take a PDF as input and convert it into another, more structured format. Generally, we found that the approaches taken in such programs only perform a very limited understanding of the document, and typically use the layout features of the target format to compensate for the limitations of the document understanding process. The following products were investigated in detail:

- *Archilogue PDF to HTML Converter* [Archilogue 2006 (Web)]: This converter aims to preserve the original layout of the PDF. The only document understanding steps which are performed are line finding and reading order detection. These lines are then placed with transformed absolute coordinates (using `<div>` tags) on the resulting HTML page. To reduce the likelihood of overlapping text due to

font changes, the relative font size is reduced. Graphical elements are rasterized and displayed as a background image. Due to the line finding and reading order detection, it is possible to select text in most converted documents. However, the resulting HTML is not repurposable.

- *ABBYY PDF Transformer 3.0* [ABBYY 2006 (Web)]: We tested the PDF-to-Word conversion feature of this software. Generally, the resulting documents visually represented the originals and it was possible to select text and make minor edits. Tables were detected if ruling lines were present; otherwise, text aligned in columns was represented using the columns feature of Word.

With both converters, their layout-preserving nature was used to overcome the limitations of the document understanding process, and it was possible to make minor edits in the resulting documents. In such a situation, the user can be said to "complete" the document understanding process by using his own knowledge when making such edits. However, the logical structure of the documents was not fully rediscovered so that the documents could be repurposed or used for machine processing applications, such as data extraction.

2.3 Summary of our conclusions and approach

Based on an analysis of the previous work covered in this chapter, we have implemented *PDF Analyser*, a system for document analysis of PDF files, which is described in more detail in Chapter 3. As PDF Analyser uses solely *document generic* knowledge at several granular levels to process an arbitrary PDF document, we concentrate mostly on the document's physical structure, rather than the textual content. This processing task can be split up into two phases: *extraction* and *segmentation*.

The extraction phase works directly on the PDF content stream to obtain the page content as a set of objects using an extraction procedure based upon the *PDFBox* library. In addition to obtaining text from the PDF content stream as in [Chao and Fan 2004], we also obtain graphic objects directly from this stream, instead of using an intermediate bitmap representation. As graphic objects can be very complex in PDF, we employ additional heuristics to discard illustrations and retain only lines and boxes that are likely to be useful for document understanding. Section 3.3 describes these methods in detail.

Our methods for page segmentation are described in Section 3.4. The *ordered-edge segmentation algorithm* was developed to overcome the main limitation of bottom-up approaches without increasing complexity or execution time significantly. The

edges between neighbouring blocks are ordered in such a way that more ambiguous edges are visited later on in the process, after most of the edges have already been visited and significant higher-level information about the document's structure is also available for the decision making process.

Furthermore, our method of wrapping using graph matching was also inspired by our interactions with the PDF converters that we analysed, where layout preservation was used to compensate for the inaccuracies in document understanding. Our interactive wrapping approach, which is described in Chapter 6, is also based on the principle that the *user*, by interacting with a structure based on the visual appearance of the page, indirectly provides information to complete the document understanding process.

Chapter 3

A system for document analysis of PDF files

This chapter describes the development of a prototype system, *PDF Analyser*, to perform document analysis of PDF files. The system loads a given page of a PDF file, extracts the text and graphics objects, performs bottom-up segmentation on the text and simplifies the graphical data so that only the most important graphical objects which have a structural function (and are therefore necessary for document understanding) are represented. The results are displayed in graphical form to the user. Much of the content of this chapter can also be found in the author's publication [Hassan 2009a]. The system uses solely *document-generic* knowledge and is designed to work on an any PDF document which has been digitally generated from a computer application.

Section 3.1 describes the model that is used to internally represent the contents of the PDF. This is a simplification of the original PDF structure. Section 3.2 presents the user interface of the system. Section 3.3 describes how the low-level instructions in the PDF are processed to populate the model with the data on the page. Section 3.4 presents the *ordered-edge segmentation algorithm*, a bottom-up algorithm to segment the textual content into logical blocks. Finally, Section 3.5 presents an experimental evaluation of the system.

3.1 Internal representation model

In order to perform analysis of PDF documents and facilitate further processing, a model was devised to store all the physical items in the document. This model is based on rectangular objects, or **segments**, which may be hierarchically nested. Sub-

sequent chapters of this thesis also refer to the object types in this model. In Section 2.2.3, we stated that we are dealing predominantly with Manhattan or near-Manhattan layouts. Thus the limitation of rectangular boundaries for segments allows our model to be relatively simple, yet offers enough granularity to represent the document successfully for data extraction purposes.

This section describes this model and an algorithm to generate an *adjacency graph* from a set of blocks, which enables efficient access to a node's direct neighbours and is used at several processing stages.

3.1.1 Coordinate system

All segments are represented by their rectangular *bounding box* coordinates on a 2-D Cartesian plane. We use the same coordinate system as in PDF. The origin (0,0) is at the bottom left of the "canvas" and one *unit* is equal to $\frac{1}{72}$ inch[1]. Depending on how the PDF has been created, objects may also have negative coordinates, particularly if they occur outside of the printed area of the page. Section 3.3.1 describes page areas in more detail. When the results of our analysis are viewed onscreen, the units are translated into screen coordinates, which have their origin at the *top-left* of the screen.

3.1.2 Segment types

We define the following types of segment, which can occur on the page:

- **Basic segments** are defined simply by their bounding box coordinates (x_1, x_2, y_1, y_2), of which there are three subtypes:

  - **line segments** represent straight lines drawn on the page either horizontally or vertically;

  - **rectangle segments** represent rectangles drawn on the page; and

  - **image segments** represent bitmap images.

- **Text fragments** contain in addition to their bounding box coordinates a **text string**, **font** object and **font size**.

---

[1]This definition is also used in PostScript. PostScript and PDF units are often referred to as *points*, which are common units in the printing industry. However, this unit has varied considerably in the past and, as stated in the *Adobe PDF Reference*, there is no universal definition of a point.

- **Clusters** are composite text segments, which are built up of a collection of text fragments and/or clusters. The attributes *text string*, *font* and *font size* also apply; they relate to the combined text of all subelements and most commonly occurring font and font size respectively. The term **text segments** is used to refer to both clusters and text fragments.

A **page** is a collection of any of the above segments and also contains a bounding box definition corresponding to the page size, which is obtained from the PDF (see Section 3.3.1). A **document** is an ordered list of pages.

The use of hierarchically nested segments allows us to represent objects at various levels of granularity, and manipulate objects in intermediate processing steps in a consistent way. Some of these objects are shown in Figure 3.1. This terminology will be used throughout the dissertation.

3.1.3 Adjacency graph representation

During processing, we need to put the segments into a suitable data structure which allows fast access to each segment's neighbours. To this end, we have devised the **adjacency graph** representation, in which each segment is represented as a node and each neighbourhood with one of four attributed edges corresponding to its direction: **northOf**, **southOf**, **eastOf** or **westOf**. For each segment, we look for its first *direct* neighbour regardless of distance.

The algorithm for generating the graph, which takes a list of segments as input, is described in Algorithm 1.

Algorithm 1 Adjacency graph generation from a list L of segments

1. create two lists, H and V, which contain all segments in L sorted in horizontal and vertical order of midpoint coordinate respectively.

2. foreach segment s in L:

   (a) locate index of s in H and V. Starting from these positions, examine both lists in ascending and descending order. This corresponds to looking for the next neighbouring block in each of the four directions of the compass.

   (b) as soon as a segment t is reached whose midpoint y coordinate (if looking horizontally) or midpoint x coordinate (if looking vertically) intersects that of s *and vice versa*, store t as neighbour of s in that particular direction. Do not look further in this direction.

Because the relations **westOf** and **northOf** are opposites of the relations **eastOf** and **southOf**, we could only store relationships in these two directions. However, because we require fast access to neighbours in all directions, we do not do this. For consistency, we ensure that every relation is also expressed by its opposite. For one-to-one relationships, these are automatically found by the algorithm. For many-to-one relationships, as shown in Figure 3.2, only one **rightOf** relation is found for the image, although it has several neighbours to its right. However, as each line of text has the image as its **leftOf** relation, the missing opposite relations are automatically added in a postprocessing step.

Figure 3.2 shows an example of the adjacency graph at the initial text fragment level after preprocessing, which is used as input to the ordered-edge segmentation algorithm. Figure 3.3 shows the adjacency graph at the block level to join neighbouring blocks after segmentation.

The adjacency graph is designed only to represent a two-dimensional page structure, i.e. a *snapshot* of a stage of processing at a given level of granularity, and is, in most cases, planar. For nested objects, we choose a particular level of granularity for the graph. In the worst case, for n segments, the graph generation algorithm requires $2n(n-1)$ iterations, i.e. it is $O(n^2)$. However, the complexity is significantly reduced in practice, as the search in a given direction stops as soon as a neighbour is found.

Whereas the algorithm performed adequately for most documents that we encountered, we found execution time to be too high when run on complex documents such as newspapers at the initial text fragment level of granularity. Therefore, the initial preprocessing stages, as described in Section 3.4.1, have been introduced to reduce the number of segments to an acceptable level. Performance figures are given in Table 3.2.

This graph representation is also used as a basis for our graph-based approach to wrapping. See Section 6 for more details.

3.2 Visualization

We have developed a GUI to visualize the results of our processing algorithms. After selection of the desired filename, page number and *segmentation mode* or final processing stage, the PDF file is opened and processed accordingly.

Using methods from the *XMIllum* Java framework [Rigamonti et al. 2003; Univ. Fribourg 2002 (Web)], the rectangular objects are displayed in layers and overlaid on top of a bitmap rendition of the page. We refer to this as the **page view**. The individual layers correspond to the different segment classes as defined in this chapter and can

Figure 3.1: Screenshot of the GUI (see Section 3.2) showing some of these objects on the page. *Line segments* are marked in purple, *image segments* in blue. *Clusters* are shown in a cyan outline and their constituent *text fragments* are shaded in yellow

Figure 3.2: Example of a *many-to-one* adjacency relationship between the image and lines of text. Please note that the blue blocks represent the *clusters* at the *initial* stage of the segmentation process, before they have been merged together into larger blocks

Figure 3.3: A further example of *PDF Analyser*'s page view, based on methods in the *XMIllum* library, showing the results of our segmentation algorithm overlaid on a bitmap rendition of the page. Adjacency edges between clusters are also shown here

be shown or hidden by the user at will. The view can be zoomed in and out, and scrolled. The bitmap rendition is generated by *Ghostscript*, and is also represented as a layer, which can be shown or hidden at will.

The GUI also provides further options to fine-tune the processing steps for debugging purposes. For example, ruling object processing can be disabled and the system can be set to stop after a predefined number of iterations to display the result partway during processing.

PDF Analyser also interfaces with the *TouchGraph* library [TouchGraph LLC 2006 (Web)] to display the document's adjacency graph in a flexible, navigable form and allow the interactive generation of wrappers for the graph-based approach. Please refer to Section 6.4.1 for a detailed description of the **graph view** of PDF Analyser.

It is also possible to show edges between adjacent segments in the page view, although, due to clutter, they are not always clearly visible here. The examples in Figures 3.2 and 3.3 have the edges shown. The graph view's flexible arrangement of nodes overcomes this problem, and enables the precise graph structure to be studied in detail. Many further screenshots from the GUI are provided throughout this thesis.

3.3 Obtaining data from PDF

In this section, we describe how we open a PDF document, parse the contents of the PDF file and populate our data structures defined in Section 3.1 with object data obtained directly from the page's content stream. At this stage, we also perform some initial processing of the PDF data to reduce the initial number of objects for performance reasons.

The PDF specification [Adobe Systems Inc. 2009 (Web)] is published by Adobe and can be downloaded freely from their website. There have been a number of incarnations of the PDF format since its conception in 1993, and the latest specification represents a published ISO standard. The vast majority of documents encountered on the Web are published in earlier versions of the format (1.4 or earlier), and we have therefore concentrated on implementing features available in these versions.

3.3.1 Page objects

Each PDF file contains a *page tree* which contains the individual page objects which contain the page's data in a *content stream*, a sequential list of instructions in the form of operator and operand pairs. These instructions are based on the operators in the

PostScript printing language. It is these instructions that we process in order to obtain the text and graphic objects that are drawn on the page.

The page object also includes other important data such as a thumbnail image of the page and the page's dimensions. The latter is of particular importance to us, as it enables us to scale the objects correctly so that the entire page is displayed on the screen and aligns correctly with its bitmap rendition (generated separately by Ghostscript) when it is overlaid in the GUI.

The PDF specification actually allows up to 5 different page sizes or bounding boxes for each physical page to be defined: **MediaBox**, **CropBox**, **BleedBox**, **TrimBox** and **ArtBox**. These parameters enable crop marks and other markings to be drawn outside of the final cropped area of the page. They also enable a different page size to be specified for screen and printer. The **MediaBox** is the largest of these boxes and its definition should always be present in the PDF file. All objects, whether visible or not in the final printed output, should fall within the bounds of the **MediaBox**. The other dimension of interest is the **CropBox**, which defines the final page area in viewing applications such as Adobe Acrobat and Reader.

We did once come across a document which did not include a **MediaBox** definition, and therefore presumably did not conform to the PDF specification. As this document did include an **ArtBox**, it displayed without error in viewing applications such as Adobe Acrobat and Ghostscript. Our algorithm therefore looks for a suitable alternative bounding box (starting with the next largest), should the **MediaBox** definition be missing.

Many PDF documents did not include a **CropBox** definition; in this case, the specification states that the **MediaBox** is to be used as the final page area. We found Ghostscript to be somewhat inconsistent when choosing an appropriate bounding box when rendering the page, particularly if other bounding boxes were defined. The problem was solved by using the −dUseCropBox command line parameter, which forces the correct bounding box to be used.

A page may also be rotated by 90 or 270 degrees clockwise or anticlockwise (clockwise rotations are negative amounts). Certain versions of the Ghostscript executable rotate the page automatically by default, whereas other versions require this to be specified by a command-line parameter.

The Java library *PDFBox* is used to process the data held in the PDF file and arrive at the content stream of each page. This is described in more detail in the following section.

	Operators
Implemented by us	B, BI,c, CS, cs, Do, f, F, f*, h, K, k, l, m, n, q, Q, re, RG, rg, s, S, Tj, TJ, v, w, W, W*, y
Already implemented in PDFBox	BT, cm, d, ET, gs, T*, Tc, Td, TD, Tf, TL, Tm, Tr, Ts, Tw, Tz, \', \"
Not implemented	b, b*, B*, BDC, BMC, BX, d0, d1, DP, El, EMC, EX, G, g, i, ID, j, J, M, MP, ri, SC, sc, SCN, scn, sh

Table 3.1: A list of operators which are implemented in our system

3.3.2 Processing the page contents

By extending PDFBox's `OperatorProcessor` class, it is possible to define which actions are taken when a particular PDF instruction is encountered. As our goal was to obtain enough information to perform document understanding and text extraction, we did not need to create methods for all possible operators in the PDF specification. The operators that we implemented are shown in Table 3.1. In particular, we extract all text and bitmap image blocks, but only certain vector items likely to help us understand the page better, such as ruling lines and rectangles, and not logos or illustrations.

3.3.2.1 Text and graphics state

PDF has two coordinate systems: global and local. The local coordinate system can be changed by altering the transformation matrix with the cm *(concatenate)* operator. This way, parts of PDF code can simply be reused at different sizes and positions of the page without needing to be rewritten. In this way, external artwork such as advertisements or diagrams can easily be placed in a PDF. Fortunately for us, the existing PDFBox methods take care of all the translation operators.

3.3.2.2 Text elements

The PDF specification defines two operators for positioning text on the page: Tj *(show text)*, which takes a string as its operand, and TJ *(show text glyph)*, which takes an array of strings and numbers as its operand. Whereas the former simply places text on the page, allocating to each character its normal width as defined in the font, the latter operator allows the individual spacing between glyphs to be adjusted. As most desktop publishing packages use their own kerning algorithms, we found the TJ operator to occur more frequently.

By default, the methods in the PDFBox source code split each TJ instruction into its subinstructions and place each individually positioned block separately on the page. This results in initial text blocks of usually no more than 2–3 characters in length. We first tried to merge all text blocks together that were created from the same TJ instruction. In some documents, such the example in Figure 3.4, this gave us complete lines of text, whereas in other documents it made little or no difference to the result. Unfortunately, we also found that many tables were generated by using a single TJ instruction for a complete row, and that operands designed for kerning adjustments were used to jump from one column to the next (see Figure 3.5 for an example). It is worth noting that this only occurred in certain tables and never with columns of text.

As we did not wish to risk overmerging the blocks, we decided to keep our initial text fragments to the granularity of subinstructions of the TJ operator as well as individual Tj instructions. These fragments are then used as input to our segmentation algorithm as described in Section 3.4. Note that, in some cases, we found that it was not possible to completely avoid overmerging text fragments at this stage and we therefore need to split them later, as shown in the example in Fig. 3.9. This problem is described in Section 3.4.1.

Special cases The vast majority of documents did not represent spaces using space characters, but used kerning instructions to provide the appropriate gap between words. If space characters were included between words, this did not make any difference to the final result. However, in some documents containing monospaced (ASCII) text, we found that space characters were also included *before* and *after* the horizontal start and end positions of blocks of text. In fact, the entire width of the page's print area was represented as one Tj instruction, and spaces were used to position text horizontally, as on a typewriter. This was clearly a sign that the PDF was created by a legacy terminal application upgraded to output PDF files.

Unfortunately, such an input results in text segments with inaccurate x coordinates and tabular data being merged across the entire width of the page, as shown in Figure 3.6. In order to overcome this problem, we have developed a *monospace mode:* each text fragment is first split into its individual characters. Then, the individual characters are merged using the initial merging procedure with a higher threshold than normal. The result is shown in Figure 3.7.

Finally, it is worth noting that characters (or complete strings) are sometimes *over-printed* with a slight offset to simulate boldface type. As long as these instructions follow another, they are automatically detected and represented by a single text fragment with the boldface flag set to true.
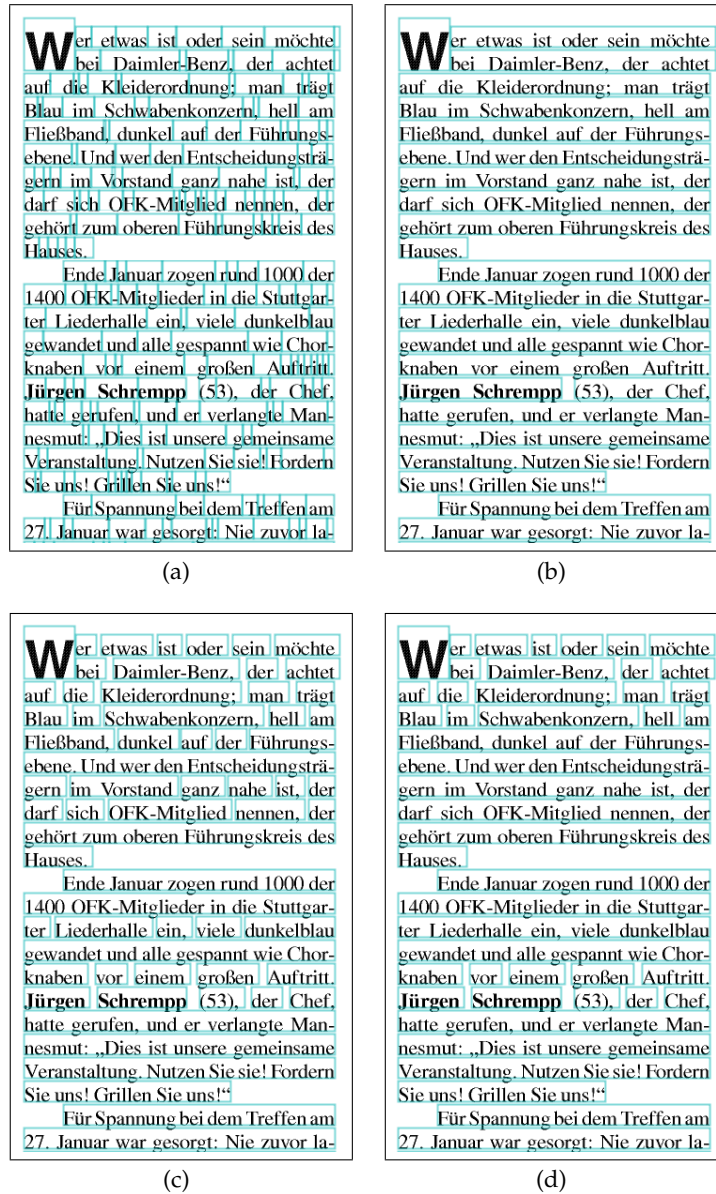
Figure 3.4: Figure (a) shows how the text fragments are represented in each individual subinstruction in the PDF. Figure (b) shows how merging across entire TJ instructions can result in complete lines of text being formed (except where font changes occur). However, as shown in Fig. 3.5, this method can also lead to overmerging. Figure (c) shows the result when the subinstruction blocks in (a) are merged using the first initial merging procedure to join blocks serially written to the PDF file as described in Section 3.4.1. Figure (d) shows the result of the second initial merging procedure, which joins closely neighbouring segments together, as applied to the blocks in (c). This is the the result that is used as input to the ordered-edge clustering algorithm

Figure 3.5: The example on the left shows how merging across TJ instructions can result in *overmerging* of segments, as kerning operators are used to jump across the individual table columns. Using subinstructions (right), this problem is avoided

Figure 3.6: Entire lines of monospaced text written to the PDF in a single instruction

Figure 3.7: The monospace mode, which splits these lines at areas of significant whitespace

38

3.3.2.3 Graphic elements

Bitmap images   Bitmap images are relatively straightforward to extract. An image is placed on the page either using the Do *(invoke)* instruction or as an inline image using the BI *(begin inline image)*, ID *(image data)* and EI *(end inline image)* instructions, together with its rectangular coordinates before scaling and transformation. The only main pitfall is that of clipping paths: we found it very common that the actual images would occupy a larger area than what was visible on the page, and that these extra parts of the image were clipped to their final size using a rectangular clipping path (see Section 3.3.2.3). We imagine that this is the result of the cropping functionality in common desktop publishing systems, which simply send the data to the printer in the most straightforward manner.

Vector elements   Vector elements are a greater challenge for us, as we need to differentiate between objects which are parts of vector images (such as illustrations and diagrams) and objects which play a dominant role in conveying the logical structure of the page to the reader, such as ruling lines and boxes. It is worth noting that, in the latter, curved segments are rarely used.

In PDF, vector graphics are drawn by defining a *path* which comprises one or more connected *subpaths*. A new subpath is begun by the m *(moveto)* operator. Straight line segments are drawn by the l *(lineto)* operator; curves by the c *(curve to)*, v *(curve to replicate initial point)* and y *(curve to replicate final point)* operators; and rectangles by the re *(append rectangle to path)* operator. The operator h *(close)* closes the subpath with a straight line back to the starting coordinate. A rectangle is equivalent to drawing three line segments and closing the subpath.

As our simplified model only includes line and rectangle objects, we approximate Bézier curves with straight lines through their coordinate parameters. (In fact, we discard all paths which include curves; we only need to store them at this stage in case they are later used to define a clipping boundary). Subpaths which include curves are flagged as such. We store all generated subpaths until they are either stroked by the S *(stroke path)* or s *(close and stroke path)* operator, filled by the f *(fill non-zero rule)* or f* *(fill even-odd rule)* operators or the path is ended. The n *(end path)* operator clears the path without stroking or filling; it is generally only used to clear the path after a clipping path has been defined (see "Clipping paths" below).

When we come across a stroking or filling operator, we first check that the current colour's grey value lies below a certain threshold. If so, we represent each subpath which contains only vertical and horizontal lines and/or rectangles with its respective objects in our simplified model. If a clipping area is active, we first clip the objects.

39

If the width or height is above a minimum threshold (defined as 3 × modal font size of all text blocks on the page) and if, according to our heuristic, no other smaller or curved graphic objects are nearby, these objects are then stored in our representation.

We find that the above treatment of PDF vector graphic instructions enables us to obtain a simplified representation of the most important lines and boxes which are *of material importance* for layout analysis, i.e. they are likely to be noticed immediately by a human reader just scanning through the page and are at the level of granularity we require for performing document analysis.

Rectangles and lines   In many cases, we found that ruling lines on pages were actually drawn as filled rectangles. Conversely, in some rare cases, rectangular-looking objects were actually drawn as very thickly stroked lines. After object extraction, we examine the dimensions of each rectangle and line and, if the shorter dimension is below or above a given threshold based on modal font size (usually about 5 pt), the object is reclassified if necessary.

Clipping paths   The PDF specification allows the use of any arbitrary path as a clipping path, which can be set using the `W` *(modify clipping path non-zero)* and `W*` *(modify clipping path even-odd)* operators. Thus it is possible to create interesting graphic effects or clip images in a non-rectangular fashion. As we are not aiming to precisely recreate the appearance of the PDF, these operators are not of particular interest to us. Even the current version of PDFBox does not yet provide support for this operator in its page rasterization methods. However, as mentioned above, we have found that clipping paths are often also used to rectangularly crop images and, in some cases, also ruling lines. We therefore approximate the result by storing the *bounding box* of the clipping path and clipping all objects to this rectangular area when they occur. We find that this gives satisfactory results for our purposes, as shown in Figure 3.8.

3.4   Page segmentation

In most document processing tasks, the first step is to decompose the page into its constituent segments, which somehow represent *logical units* of the document. As a page can be represented on several granular levels, these segments can later be merged together or further decomposed, depending upon the algorithm used.

Section 2.2 summarized existing approaches to document analysis and understanding, with particular emphasis on page segmentation. This section presents the *ordered-edge segmentation algorithm*, an efficient, robust, bottom-up page segmentation

Figure 3.8: Page display without clipping (left) and with clipping (right) of an image

algorithm which overcomes the "knowledge gap" of bottom-up techniques by order-ing the edges in such a way that more ambiguous edges are visited later on in the process, when significant higher-level information about the document's structure is also available for the decision making process. Experimental results show that the algorithm produces good results, even on complex layouts such as newsprint. Unlike most page segmentation methods in the literature, which work on a bitmap rendition of the page, our algorithm works directly on the objects that are obtained using the methods in Section 3.3. This algorithm has been published in [Hassan 2009a].

3.4.1 Preprocessing: Initial merging of horizontally adjacent blocks

We take as input a list of text fragments, which correspond to individual (sub)instructions for displaying text, may contain anything from one to several char-acters each, and are clearly oversegmented at this stage (see Figure 3.4 (a)). As a complex page could contain thousands of these segments, we first aim to reduce this to a more manageable number to keep processing time of adjacency graph generation and the later stages of our analysis within reasonable bounds.

Although the text fragments could be written to the PDF in any arbitrary order, we have found that the order usually somewhat corresponds to the reading order of the text, at least at the line level. Because text fragments corresponding to a single TJ instruction are always returned together, it makes sense to first process this list linearly (with relatively little processing cost) and join neighbouring segments if they are on the same line. In this situation, we use a *word threshold* of $0.25f$ and a *line threshold* of $1.0f$, where f refers to the mean font size of both text fragments. For pairs of blocks above the word threshold but within the line threshold, we merge the blocks but assume the characters belong to separate words and add a space between them. The results of this merging procedure are shown in Figure 3.4 (c).

After this initial step, a merging procedure merges horizontally neighbouring blocks which are not written sequentially to the PDF. We sort the blocks in y-then-x order; this means that blocks with similar baselines (defined as being within a threshold of $0.15f$) are returned together in left-to-right order, and that these individual lines of text are then sorted from top to bottom. We then join any neighbouring blocks if they are on the same line and so close together that they could not conceivably belong to different columns. Therefore we use a very tight threshold of $0.2f$.

The reason we allow for a greater threshold in the former case is because we are only comparing neighbouring items at this stage. As most text is written to the PDF in its reading order, the chances of overmerging are very low. Furthermore, the threshold of $1.0f$ is still low enough not to merge across neighbouring columns of text. Should overmerging occur, for example in tight tabular layouts as shown in the example in Fig. 3.9, a check at every iteration of the segmentation algorithm (see Section 3.4.3) will take care of up to two overmerged lines in a text block. In the latter merging process, we are comparing each block with every other, and the likelihood of overmerging is therefore greater.

The result of both initial merging procedures is an approximately five-fold reduction in the number of objects on the page (Figure. 3.4 (d)). With this list of objects as input, the adjacency graph generation procedure (as described in Section 3.1.3) is executed. This results in a significant performance improvement, as shown by the timings of the system on four test documents of varying complexity with and without the preprocessing stage in Table 3.2.

3.4.2 The "brickwork" effect

One important feature of text layouts is that the words in a paragraph are almost always laid out in a "brickwork" fashion. For a given pair of adjacent lines, spaces between words very rarely occur at exactly the same horizontal positions. This is

	Document	IHTfrontpage	CanonLenses	StatAustria	ChemList
	Complexity	high	high	medium	low
	Tabular?	no	yes	yes	yes
Without pre-processing (*sub-instructions*)	No. nodes	5564	4545	509	232
	AG generation (ms)	158133	83159	551	123
	Segmentation (ms)	50883	86217	1320	528
	Total proc. time (ms)	209706	170244	1936	705
With pre-processing (*text fragments*)	No. nodes	1034	2218	509	79
	AG generation (ms)	4958	20655	538	39
	Segmentation (ms)	3396	24576	1267	397
	Total proc. time (ms)	8479	46259	1917	462

Table 3.2: Comparison of performance timings for one page with and without pre-processing (averages from three runs)

more likely to occur in monospaced documents, which have a limited number of horizontal positions along a line. Kieninger et al. state that:

> In some cases a regular block might show some white space at the same x-position throughout the complete block. These so called *rivers of whitespace* are said to be *bad layout* and are tried to be avoided by modern typesetting programs and wordprocessors. They are more likely to occur in small blocks of only a few lines, using fixed width fonts (e.g. ASCII texts). ([Kieninger and Dengel 1998a], p. 7 [261])

If this were allowed to happen, these *rivers of whitespace* in the text would start to build up and the paragraph would no longer appear as a self-contained element to the human reader, but rather as individual columns of tabulated data instead. In a similar way to the *T-Recs* algorithm described in [Kieninger and Dengel 1998a], the ordered edge segmentation algorithm also relies heavily on this effect being present when merging adjacent rows of text. An example is shown in Figure 3.10.

3.4.3 The ordered-edge segmentation algorithm

After adjacency graph generation (see Section 3.1.3), each of the neighbourhood relations is represented as an attributed edge with attributes such as **fontSize**, the average font size of the two nodes, and **length**, the closest distance in points between the edges of both segments (for vertical edges, the distance between the *baselines*) relative to font size, as well as v_f and v_t, the edge's **nodeFrom** and **nodeTo** respectively, i.e. the two text blocks which the edge connects. Based on these attributes, the edges are sorted in such a way that the "easiest" edges are considered first.

Figure 3.9: Example of tightly arranged column headings, which need to be accounted for at a later stage of the segmentation process

Figure 3.10: An example of a paragraph as represented by the *text fragments* after initial merging and joined by edges representing adjacency. Here we can see the "brickwork effect": note that the entire paragraph could be built by joining just the vertical edges. The initial merging of successive pairs of blocks has resulted in the line with tight spacing being merged completely

---

Algorithm 2 Edge-sorting function for the ordered-edge segmentation algorithm

1. all vertical edges $e = (v_f, v_t)$ are considered before horizontal edges; for vertical edges:

   - edges where the font size of both segments v_f and v_t is approximately the same are considered first; then

     – edges where the average font size of both segments is smaller are considered first; then

       ∗ edges with a smaller edge length (line spacing) are considered first; if it's approximately the same, then

         · edges where the width of both *segments* is approximately the same are considered first

2. for horizontal edges:

   - edges are sorted by edge length in ascending order

The algorithm first clusters together segments joined by edges where it is obvious, from *just the information in the two segments alone*, that they belong to the same logical block. After most of these blocks have already been formed, the more problematic edges are then examined, where it is not always possible to determine *a priori* whether they should be clustered together or left apart. At this stage, a better decision can be made, as the higher-level block structure is already partly present. Thus, the algorithm overcomes the most significant limitation of other bottom-up approaches.

The first stage is to sort the edges into an appropriate order such that the most likely edges will be visited first. The ordering sequence is given in Algorithm 2.

Note that all edge lengths are always relative to font size, i.e.:

$$\text{edge length} = \frac{\text{shortest length between the blocks (or } \textit{baselines} \text{ for vertical edges)}}{\text{average font size}}.$$

As we are working only with text blocks, we ignore any edges which join text blocks to other objects or other objects to each other.

It is worth noting that vertical edges are deemed the most important in bottom-up page segmentation. In fact, it is usually sufficient to join only the vertical edges to obtain all blocks of text. Because of the "brickwork" effect, as described in Section 3.4.2, we can build most paragraphs completely just from the vertical edges alone (see Fig. 3.10). In fact, in the rare case that words in a paragraph do line up vertically, this already begins to appear as tabulated data to a human reader, and this is why

we need to exercise great care when joining horizontal edges. Therefore, horizontal edges are visited only after all vertical edges have been processed. Only at the very end of processing, any remaining unconnected horizontal neighbours (usually single lines) are joined together if necessary.

The algorithm is given in Algorithm 3 and refers to an external function **cluster-Together**. In practice, the implementation is somewhat more complicated, as hash maps are used to improve performance.

Algorithm 3 The ordered-edge segmentation algorithm

---

pre: $G = (V, E)$ is the *adjacency graph* of the blocks to be segmented

1. let L be a copy of E, sorted according to Algorithm 2

2. while (L is not empty)

    (a) get next edge $e = (v_f, v_t)$

    (b) if v_f and v_t not yet in output

        i. if **clusterTogether**(v_f, v_t, e)
            - create two new clusters with v_f and v_t as single subitems respectively and add to output

    (c) else if v_f not yet in output

        i. **clust**←find cluster containing v_t
        ii. if **clusterTogether**(v_f, **clust**, e)
            - add v_f to **clust.items**

    (d) else if v_t not yet in output

        i. **clust**←find cluster containing v_t
        ii. if **clusterTogether**(v_t, **clust**, e)
            - add v_t to **clust.items**

    (e) else (both nodes already in output)

        i. **clust1**←find cluster containing v_f
        ii. **clust2**←find cluster containing v_t
        iii. if **clust1** $\neq$ **clust2** $\wedge$ **clusterTogether**(**clust1**, **clust2**, e)
            - merge **clust1** with **clust2**

The function **clusterTogether** uses a number of heuristics to decide whether two given clusters belong to each other. For vertical edges, this method returns **true** if:

- the new item(s) to be added are consistent with the line spacing of the existing cluster; and

- the font sizes are approximately the same[2].

For horizontal edges, the nearest *vertical* neighbour of both **nodeFrom** and **nodeTo** is found. If **nodeFrom** and **nodeTo** have different nearest vertical neighbours, the closest (in terms of y-axis distance) is chosen. Based on this distance and the number of lines of text that each text block contains, a heuristic is used to compute a maximum width threshold.

This threshold is normally $0.75f$, but can be increased in the following cases:

- As blocks containing fewer lines of text are most likely to have been not fully clustered by the algorithm, the heuristic allows for an increased edge width threshold in such cases.

- Similarly, we have noticed that headings and other freestanding items of text often exhibit a wider character and word spacing. As long as they are not immediately surrounded by other text, it is clear to the reader that they still form a complete line of text. Therefore, the edge width threshold is also increased where the nearest vertical neighbour distance is large.

clusterTogether then returns **true** if:

- the new item(s) to be added are consistent with the font size of the existing cluster; and

- the edge width (i.e. the horizontal distance with respect to font size) does not exceed the above computed threshold.

Additionally, for each creation or modification of a cluster, a further check is carried out on the new cluster; if this check fails, merging of the edges is aborted. We have found that, in certain very tight tabular layouts, the column headings may be written so closely together that they appear *a priori* to be a single, contiguous line of text. In fact, the spacing between headings of adjacent columns can, in special cases, even be less than the normal word spacing, as shown in the example in Fig. 3.9. This can even occur if no ruling lines are present.

---

[2]This has the effect of leaving out superscript, subscript, and other small items of text which may occasionally occur in a paragraph. These are then added to their respective paragraphs at the end of processing.

The reader still recognizes the delineation between each individual column heading because of the clear column-based structure below, and because the headings are still consistently aligned with the data in these columns. We therefore check for such structures at every iteration of the segmentation process. After the columns have been clustered together, our heuristic detects that the text block has developed one or more *rivers of whitespace* and splits the headings (maximum 2 lines) appropriately.

3.4.4 Postprocessing

After execution of the ordered-edge segmentation algorithm, any resulting segments that are crossed by a ruling line are split across their constituent fragments. This way, any table cells that were erroneously merged together are split apart. Also, superscript, subscript and other small items of text located next to the boundaries of larger text blocks are merged together with these text blocks.

3.5 Experimental results

The algorithms for object extraction and segmentation were tested on a dataset containing the front pages of 50 different issues of *The Sydney Morning Herald*[3]. The objects that were obtained as a result of the processing step were visually compared against what a human reader would deem to be the correct result or "ground truth". The results are shown in Table 3.3.

The experimental evaluation raised two important issues: Firstly, in our case, the ground truth was very open to interpretation, as exemplified in the following questions:

- Which lines on the page are materially important in gaining an understanding of the document's structure and which are not?

- Should indented paragraphs, which are not separated by extra whitespace, belong to individual blocks?

There are, of course, several levels of granularity in which a document could be represented and the results of our algorithms can only be seen as a first step in the document understanding process. For example, indented paragraphs within blocks should then be detected by appropriate methods at a later stage. It was therefore very difficult to generate quantitatively measured results, as the evaluation process is

---

[3]*The Sydney Morning Herald*, `http://www.smh.com.au`
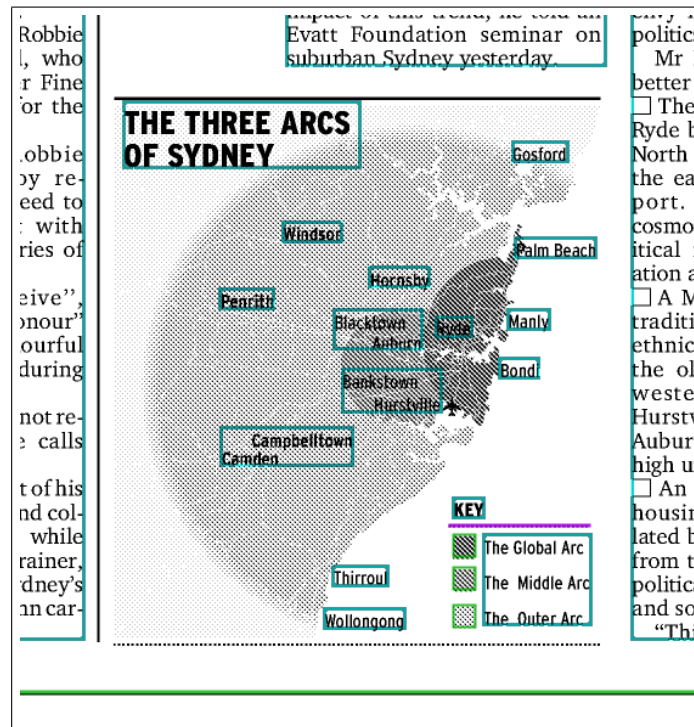
Figure 3.11: An example of segmentation errors in text inside a diagram
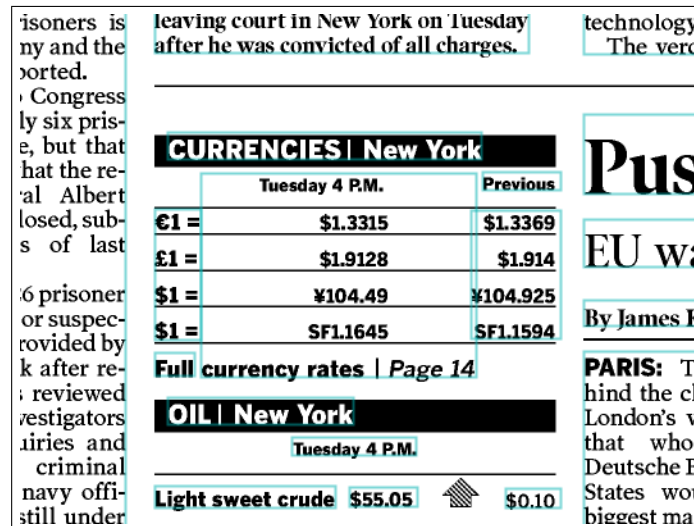
Figure 3.12: Example of the ordered-edge segmentation algorithm failing to correctly segment a table

Item type	Clusters	Ruling lines	Bitmap images	Rectangles
Total	3157	414	527	568
Detected	2978 (94.3%)	333 (80.4%)	510 (96.8%)	536 (94.4%)
False positives	13 (0.4%)	22 (5.3%)	0 (0.0%)	45 (7.9%)
Precision	99.57%	93.80%	100.00%	92.25%
Recall	94.33%	80.43%	96.77%	94.37%
F-measure	96.88%	86.61%	98.36%	93.30%

Table 3.3: Results of the document analysis system

subject to a degree of subjectivity. For this reason, we adopted a somewhat tolerant approach when judging whether a given object was represented correctly or not. In the case of paragraphs beginning with indentations, we allowed them to be merged, as we had not designed the segmentation algorithm to specifically cope with such layout conventions.

3.6 Discussion

In general, the ordered-edge segmentation algorithm was found to produce very good results, as objects were rarely split or overmerged. Because the dataset included a large number of diagrams with text labels, the ratio of correctly detected clusters was not as high as expected. As these diagrams do not have a Manhattan layout structure, the labels were frequently overmerged, as shown in the example in Fig. 3.11.

An alternative interpretation would be to class these labels as parts of images and therefore as false positives, which would lead to a significantly higher recall value. In practice, we are not interested in text in diagrams, which are ignored anyway later on in the processing pipeline. Unfortunately, we found that our evaluation strategy did not discriminate between unimportant errors in diagrams and catastrophic segmentation errors, for example when two columns of an article were merged together. Fortunately, the latter type of error was a seldom occurrence.

Our algorithms did also return some false positives, in particular for ruling lines, which were found, upon inspection, to be part of illustrations or diagrams. When designing the algorithms, we decided to err on the side of caution and output false positives rather than miss important line objects. For our purposes, this is not a big problem at all, as in our later processing steps, vector objects not in the vicinity of text are ignored anyway. Although the result is more than adequate for our purposes, further development on our vector diagram/image recognition heuristic should result in this number being significantly lower.

Even with digitally generated PDFs, certain graphic elements on the page (in particular advertisements) would have their text included in bitmap or vector form, rather than as text instructions. These text items were not found by the system. The same applies to logos. We found these to be generally text items of little interest.

Finally, a number of errors occurred where major ruling lines were either not detected on the page at all or in the wrong position. We found this to be due to a missing or incorrect implementation of the PDFBox code which handles the transformation matrix, rather than a problem with our approach.

3.7 Conclusion and further work

This chapter has presented in detail an approach for extracting textual and graphical data from PDF documents at a level of granularity suitable for document analysis purposes. A bottom-up segmentation algorithm was also presented, which reliably groups these segments into blocks representing individual logical items on the page and copes well even with complex layouts. As the resulting data is designed to be used for further processing, the numeric results cannot be directly compared to the precision and recall values of other document analysis systems. However, we are able to use the results of these processing steps in the further stages of the two wrapping approaches.

The ordered-edge segmentation algorithm was found to produce very good results by visiting the more ambiguous edges later on in the process, when significant higher-level knowledge was also available. However, some errors did still occur, particularly in tabular layouts, as shown in the example in Figure 3.12. Here, one can see that the middle column of the table was merged with the line of text below and the space between the words **Full** and **currency** was seen by the system as a column gap. This is due to the system's model being rather limited: at this stage of segmentation, no distinction is made between text blocks which are paragraphs and text blocks which are table columns. Instead, a generic set of conditions is used for each text block at this stage of the process (uniform line spacing and font size, etc.) The classification of text blocks into paragraphs and table cells/columns occurs during table recognition (see Chapter 4), where knowledge at an even higher level is available.

A further improvement could be achieved by modelling document knowledge at multiple granular levels more accurately and by improving the integration between the various processing steps. The ordered-edge segmentation algorithm can be seen as the first step in this direction. Section 7.2 describes this issue in more detail and proposes future work.

3.7.1 Exploiting hidden information in PDF documents

In developing the extraction algorithms from PDF, we noticed that the structure of the PDF and the ordering of the operators usually represent how the document would have been stored in the computer system's memory at the generation stage. There is, in fact, a wealth of extra information available in the source code of a PDF which is lost when the PDF is printed, rasterized or converted. For example:

- the order in which text blocks are written to the PDF usually resembles the reading order of the page;

- text in subinstructions within a single `Tj` instruction almost always belongs to the same logical text block (except in some tabular columns);

- the use of transformation matrices could provide hints for identifying complex objects and how the various parts of the page are grouped.

It is possible to code a PDF in a variety of different ways and still end up with the same visual result. However, most document authoring programs (such as DTPs and word processors) simply generate the PDF (or printout) in the most straightforward manner. Because the code structure cannot in all cases be relied upon to reflect the logical structure of the document, most PDF analysis approaches have ignored it completely. We believe that this information could, if correctly processed, be combined with traditional document understanding techniques and used in a probabilistic fashion to improve the robustness of such a system. In a similar way, *tagging* information, which explicitly denotes logical structure within a PDF document, could also be used in cases where it is available.

Chapter 4

Table recognition

Because of their ability to present information in a compact and easy-to-understand form, tables have long attracted attention from researchers in the information retrieval field. Tables occur in all shapes and sizes, and a multitude of different formatting conventions can be used to convey logical relations between data elements. As we are working on *document generic* data extraction techniques, we cannot make any presumptions about the formatting conventions that will be used and therefore require an approach that works on as wide a variety of tables as possible.

This chapter describes our work on table detection and table structure recognition from PDF files. Section 4.1 introduces tables in the context of information retrieval, and clarifies the distinction between the three stages of information extraction from tables: *table detection*, *structure recognition* and *table interpretation*. Section 4.2 presents related work and Section 4.3 presents our algorithm for table recognition and structure detection, which was also published in [Hassan and Baumgartner 2007]. In Section 4.4, we present comprehensive experimental results of our system and compare them to a more recently published system [Ruffolo and Oro 2009], which also performs table detection and structure recognition from PDF files.

4.1   Tables in the information retrieval field

[Hu et al. 2002] splits the task of table recognition into two stages: **table detection**, the detection of regions in the document which contain tabular data, and **table structure recognition**, the recognition of sub-structures, such as rows and columns, within the table. To begin to address the first stage, we may ask ourselves: "What is a table?" And what may seem a simple question at first becomes far more difficult when we try to precisely define a table. Quoting Lopresti and Nagy:

> A precise definition of "tabularity" remains elusive because some bureaucratic forms, multicolumn text layouts, and schematic drawings share many characteristics of tables. There are significant differences between typeset tables, electronic files designed for display of tables, and tables in symbolic form intended for information retrieval. ([Lopresti and Nagy 2000], p. 1 [93])

There have been several definitions of tables in the literature on information extraction, some of which attempt to be broad enough to encompass all tables. With reference to the information contained in tables, Green provides the following definition:

> Tables are rectangular arrays of image space within which information in row and column regions are related in some way. It is convenient to think of two types of tables, physical tables and logical tables. Physical tables are the printed manifestation of relational information. Logical tables are "relations", in a relational database sense (in fact, relations are called tables in SQL). It is a common practice to combine more than one relation via merges and joins, in the preparation of generating the data prior to printing it; thus a printed table may represent more than one relation. Also, the same relation or set of relations will have many possible physical table layouts. ([Green 1996])

Clearly, a very important characteristic of tables is their integral row-and-column structure, which allows the relationships between data items to be instantly recognized and understood. We need to ask ourselves which *visual features* tables exhibit, which make their structure—and existence—clear to the human reader. A variety of such cues, such as ruling lines, alignment, shading and whitespace, are commonly used in tables published in PDF format. From our experience with several data sets of tables, we can conclude that clearly demarcated rows are not always present in tables. But a *clear column structure* is almost always present. In fact, this column structure is what differentiates a table from other elements on the page. See Figure 4.1 for an example. Therefore, our algorithm begins by looking for such column-like structures, or **candidate columns**, on the page (see Section 4.3.1). This approach is reflected by Hu et al., who state that:

> Columns are the most visually dominant structural components of a table. ([Hu et al. 2001a], p. 1)

To address the second stage, table structure recognition, we refer back to Green's definition of *physical* and *logical tables*. Whereas physical structure refers to the rows and
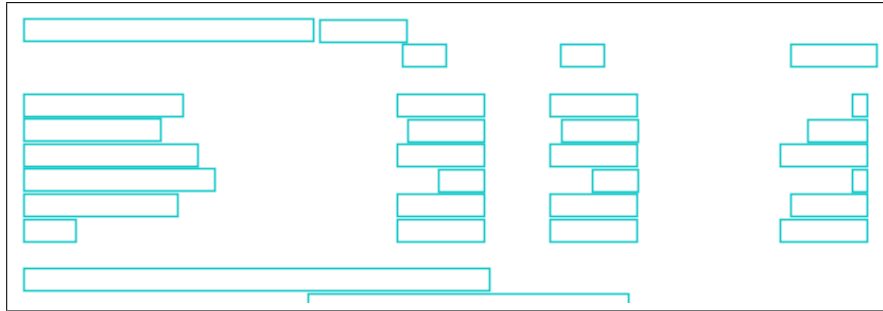
Figure 4.1: Even when all the information is removed and just the block outlines remain, the existence of a tabular structure is immediately obvious to the human observer

columns of the table, the logical structure refers to the logical relationships between the table's individual cells. [Wang 1996] defined the concept of an **abstract table** and has shown that the same logical information can be presented in tables with completely different layout structures. As a simple example, consider transposing any table so that the rows become columns and vice versa.

[Hurst 2000] (p. 156) notes that, whereas document analysis systems have traditionally strived to reconstruct the logical structure of other document elements, such as headings, paragraph text, images and captions, they have not attempted to infer *abstract* logical relations from the tabular content itself. This could have its roots in the design of modern word-processors, which provide markup for other logical elements of a document, such as styles, but only for the *physical* row-and-column structure of a table. Clearly, there is some ambiguity in the meaning of the term *"logical structure"* as used by the document understanding and information extraction communities. A purely physical, or geometric description of a table would not even include row or column sub-structures but simply define their cells by their co-ordinates or neighbourhoods. Therefore, the grouping of cells into rows and columns could already be seen as the first logical step, as it already relates the cells to each other. To avoid ambiguity, we use the following terms to define the three major sub-tasks in the table understanding process:

- **table detection**, the detection of regions in the document which correspond to tables;

- **table structure recognition**, the detection of the table's physical sub-structures, i.e. rows, columns and cells; and

- **table interpretation**, inferring abstract logical relations between the table's data cells from their physical structure.

In our application, we need to detect tables and convert them into the row-and-column model of HTML. Therefore, we only need to concern ourselves with the first two stages of table understanding. We may ask ourselves: "Why is this? Where does *table interpretation* take place?" The answer is, in the process of defining a wrapper, the *user* implicitly uses his understanding of the abstract information of the table to select which cells to extract. The only requirement is that all other wrapping instances are laid out in the same way: *the logical-to-physical mapping must remain the same*. Therefore, for our application it is sufficient just to detect the rows and columns in tables. The next section gives an overview of the literature in table structure recognition.

4.2   Related work in table structure recognition

A number of research groups have previously addressed the problem of table structure recognition in documents. Earlier publications have concentrated primarily on ASCII tables from legacy terminal applications which are set in a fixed-width font and contain no graphic objects. As the fixed-width ASCII output format began to lose popularity and the combination of technologies such as the Internet, increasing bandwidth and the PDF format enabled the circulation of graphical documents, the focus started to move towards tables in document images.

[Green and Krishnamoorthy 1995] describes an early system which analyses tables in scanned images using manually specified models. Using connected component analysis, the physical structure of the document (e.g. segments, whitespace and ruling lines) is obtained. A table is modelled using these components, some of which may repeat for each new record in the table. Once the model has been specified, it can then be used on other similarly structured tables. Because the model, which essentially represents the physical structure of the table, is specified by the user, this system could actually be seen as an early wrapping system for tables, which in fact bears a resemblance to our graph-based approach to wrapping (see Chapter 6).

[Rus and Summers 1997] describes an algorithm to locate and understand the structure of tables in ASCII documents. A whitespace density graph (see Section 2.2.2) is used to locate column boundaries. Lexical analysis on the columns is then used to determine whether a candidate table is indeed a real table. Because the input documents are all in fixed-width text format, only a limited number of layout conventions are supported. For example, the system does not appear to work on more complex or multi-column layouts (which very rarely occur in ASCII format, due to its relative simplicity). It is therefore only necessary to detect the vertical start and end positions of tables. Furthermore, the system does not appear to support spanning columns, which are anyway rare in such documents.

Kieninger et al. have also worked on recognizing similar tables mostly set in fixed-width fonts. In contrast to [Rus and Summers 1997], they use a bottom-up approach [Kieninger 1998] (as described in Section 2.2.1), which enables their algorithms to work on more complex layouts, as they are not significantly affected by other, non-tabular areas of the page. Their original *T-Recs* system [Kieninger and Dengel 1998b] clusters text bottom-up into columns. A number of rules are then used to split incorrectly merged columns and deal with other errors that may occur as a result of the bottom-up clustering process. Neighbouring columns are then merged into tables by overlaying a grid structure. Although the method claims to be applicable to any type of document, examples are only given on ASCII documents and it appears too simple to be suitable for documents with more complex layouts. As with many other systems in the literature, ruling lines are ignored. The improved version, *T-Recs++* [Kieninger and Dengel 2001], works in two phases: first, candidate columns are detected, which are clustered to form tables. A *plausibility check* is then carried out to ensure that the final object actually represents a real table, which counters one of the weaknesses of the previous system. Experimental results of the system applied to the business letter domain are given. We have borrowed some of these techniques in designing our table recognition algorithm (see Section 4.3).

[Wang 2002] has taken a different approach, using a probabilistic model for the structure of a table and formulating the table recognition problem as a statistical optimization problem on multiple granular levels. The method takes a set of segmented blocks as input (the system claims to work on both ASCII and fully graphical documents) and finds candidate columns by searching for vertical rivers of whitespace. The probabilities are modelled using several properties of the text, such as justification, baseline difference and leading. Using an iterative updating optimization algorithm *(statistical refinement)*, the preliminary detection results are adjusted and even segmentation and labelling errors can be corrected.

Hu et al. have developed techniques primarily for ASCII documents, but which also work on scanned document images. The table detection method described in [Hu et al. 2000] calculates scores for each line in the document to find the most likely table regions. Although the authors also perform experiments with scanned images of documents, the algorithm does not appear to support more complex layouts such as multi-column documents. Furthermore, it ignores non-textual elements such as ruling lines. Their table structure recognition algorithm [Hu et al. 2001a] uses hierarchical clustering combined with heuristics to determine where the region should be "cut" into columns. Table headings are detected, and spanning columns are supported in heading cells. Finally, row segmentation is performed using heuristics which merge rows according to missing data in columns.

An interesting table detection algorithm is presented in [Krüpl and Herzog 2006], which aims to detect "real" tables on Web pages. Although tables are usually explicitly represented as such on web pages, the misuse of the <TABLE> tag for general layout structures makes it increasingly difficult to detect real tables in HTML documents. Krüpl and Herzog's algorithm works on the visual level, much like the approaches to table detection for printed documents, to distinguish real tables from other data laid out using <TABLE> tags. Their system is built upon the Mozilla rendering engine and uses heuristics specific to the way that Mozilla renders tables, making it difficult to adapt for printed documents.

The problem of recognizing table structure has also been addressed in the commercial software domain, but publications are scarce. [Zuyev 1997] presents a method, which is highly dependent on ruling lines being present, to detect tables in scanned images of pages. This method has been integrated into the *FineReader* OCR product. The latest versions of *Adobe Reader* and *Acrobat* also allow tables to be exported from PDF files in a format that preserves their row and column structure. However, the user has to perform the "detection phase" and mark the table first.

Recently, a small number of publications have addressed the problem of detecting and extracting tables from PDF documents. [Yildiz et al. 2005] presents an algorithm that examines the page line-by-line. Heuristics are used to detect the presence of columns if a line contains more than one text element. These columns are then merged into tables. Because the *PDFTOHTML* tool [Kruk 2006 (Web)] is used to obtain information about the location of text blocks on the page, graphical and typographical information is not made available to this algorithm.

The *PDF-TREX* system [Ruffolo and Oro 2009] is designed to detect and recognize structure from PDF documents using text block objects. Text fragments are segmented into lines based on thresholds, each of which is labelled **text line**, **table line** or **unknown** using heuristics. Ruling lines appear to be ignored by the system. Tables are then built in a bottom-up fashion from lines labelled as **table line** or **unknown**. Finally, heuristics merge row lines which contain only one **table line** and several **unknown** lines. The algorithm is able to detect tables with spanning columns and empty cells. This algorithm was actually published two years after our method (as described in Section 4.3) was published. A further contribution of their work is the publication of a freely available dataset of PDF documents containing tables, with which we have evaluated both methods in Section 4.4.

58

4.3 The table recognition algorithm

This section describes our table recognition algorithm, which takes as input a page containing the clusters as found by the ordered-edge segmentation algorithm in Section 3.4.3 as well as the ruling lines that have been detected on the page. The algorithm is designed to detect both ruled and non-ruled tables; if ruling lines are present, they will be used to aid the recognition process.

The table recognition algorithm is divided into the following stages, which are described in detail in the following subsections:

- first, **candidate columns** are found from vertically neighbouring text on the page (see Section 4.3.1);

- the **table search** algorithm then attempts to find complete tables by grouping together horizontally neighbouring candidate columns. This algorithm searches for the maximal table that passes our validation check (see Section 4.3.2);

- the **table validation** procedure is used to verify whether a **candidate table** is indeed a single, valid table. In order to do this, the table's individual rows and columns are determined (see Section 4.3.3).

We model a table as a rectangular $m \times n$ matrix, where $m, n \geq 2$. Each cell can contain textual data or be empty. Additionally, each cell can span more than one column. Because it is not possible to reliably detect spanning rows from the physical structure of the table alone (see Section 4.3.3.2), our detection algorithm does not support row-spans. For the same reason, we also do not distinguish between *data* (`<td>`) and *access* or *heading cells* (`<th>`) as in HTML. In the remainder of this chapter, the term *data cell* is used to refer to a cell containing textual data, as opposed to an *empty* or *blank* cell.

Several illustrations throughout this dissertation show the result of our table structure recognition algorithm graphically, as displayed by the *PDF Analyser* user interface (see Section 3.2). Here, the bounding boxes of the detected rows and columns are superimposed on the page view. Columns are shaded in a green colour, and rows (and entire table areas) are shown in pink.

4.3.1 Candidate column finding

To begin with, we look for column-like structures, or *candidate columns*, on the page. To do this, the segmentation algorithm in Section 3.4.3 is reused to look for groups of vertically adjacent clusters and "cluster" them into **TableColumn** objects. This time, the following modifications are made to the procedure:

- the list of clusters, instead of (text) lines, is given as input and a list of **TableColumn** objects is returned as output;

- the vertical threshold is increased to $3.5f$ (f = average font size);

- the edges are sorted in a different order, which prioritizes edges that are aligned and narrow. In particular, vertical edges whose segments are left-, centre- or right-aligned are considered before other edges. Among these edges, edges for which the longest segment is the narrowest are considered first;

- a modified **clusterTogether** function is used, to ensure that spanning columns do not merge neighbouring columns together:

    - for every vertical edge that could potentially join two pairs of clusters, the rectangular containment expansion procedure (see Section 4.3.2.2) is executed. If extra clusters not belonging to either of the two original clusters are added to the result, **clusterTogether** returns false.

Fig. 4.2 shows the results of this procedure on a sample document containing non-ruled tables. Each text element, even paragraph text, is allocated to a candidate column which will initially be used in the search for tables. These paragraphs are later discounted in the validation procedure, either due to their attributes (e.g. width) or because they cannot be meaningfully merged into an existing table.

Note that, because of significant vertical gaps in the columns, many columns are not found as a whole, but rather in parts. This does not cause a problem for us, and is preferable to over-merging. Because of rectangular containment expansion (see Section 4.3.2.2), in most cases only one candidate column needs to be found for each vertical position in the table.

4.3.2 Table search

The table search procedure is shown in Algorithm 4. After the candidate columns have been found on the page, we begin our search procedure, which looks for *candidate tables* as groups of candidate columns within a rectangular bounding box. We first sort the candidate columns in ascending order of their *width ratio*, i.e. the width of the column divided by its height. In this way, the columns "most likely" to belong to a table are considered first. The purpose of this is twofold: Firstly, unnecessary iterations can be avoided, leading to improved performance of the algorithm. More importantly, as the search procedure does not reconsider candidate columns that have already been considered for another table, the ordering of candidate columns enables

Phosphorus (yellow or white)	7723-14-0	**Only** if it is a yellow or white form.
Sulfuric acid (acid aerosols including mists, vapors, gas, fog, and other airborne forms of any particle size)	7664-93-9	**Only** if it is an aerosol form as defined.
Vanadium (except when contained in an alloy)	7440-62-2	**Except** if it is contained in an alloy.
Zinc (fume or dust)	7440-66-6	**Only** if it is in a fume or dust form.

The qualifier for the following three chemicals is based on the chemical activity rather than the form of the chemical. These chemicals are subject to EPCRA section 313 reporting requirements only when the indicated activity is performed.

Chemical/ Chemical Category	**CAS Number**	**Qualifier**
Dioxin and dioxin-like compounds (manufacturing; and the processing or otherwise use of dioxin and dioxin-like compounds if the dioxin and dioxin-like compounds are present as contaminants in a chemical and if they were created during the manufacture of that chemical.)	NA	**Only** if they are manufactured at the facility; or are processed or otherwise used when present as contaminants in a chemical but only if they were created during the manufacture of that chemical.
Isopropyl alcohol (only persons who manufacture by the strong acid process are subject, no supplier notification)	67-63-0	**Only** if it is being manufactured by the strong acid process. Facilities that process or otherwise use isopropyl alcohol are not covered.
Saccharin (only persons who manufacture are subject, no supplier notification)	81-07-2	**Only** if it is being manufactured.

There are no supplier notification requirements for isopropyl alcohol and saccharin since the processors and users of these chemicals are not required to report. Manufacturers of these chemicals do not need to notify their customers that these are reportable EPCRA

Figure 4.2: Example of the candidate column finding procedure

Phosphorus (yellow or white)	7723-14-0	**Only** if it is a yellow or white form.
Sulfuric acid (acid aerosols including mists, vapors, gas, fog, and other airborne forms of any particle size)	7664-93-9	**Only** if it is an aerosol form as defined.
Vanadium (except when contained in an alloy)	7440-62-2	**Except** if it is contained in an alloy.
Zinc (fume or dust)	7440-66-6	**Only** if it is in a fume or dust form.

The qualifier for the following three chemicals is based on the chemical activity rather than the form of the chemical. These chemicals are subject to EPCRA section 313 reporting requirements only when the indicated activity is performed.

Chemical/ Chemical Category	**CAS Number**	**Qualifier**
Dioxin and dioxin-like compounds (manufacturing; and the processing or otherwise use of dioxin and dioxin-like compounds if the dioxin and dioxin-like compounds are present as contaminants in a chemical and if they were created during the manufacture of that chemical.)	NA	**Only** if they are manufactured at the facility; or are processed or otherwise used when present as contaminants in a chemical but only if they were created during the manufacture of that chemical.
Isopropyl alcohol (only persons who manufacture by the strong acid process are subject, no supplier notification)	67-63-0	**Only** if it is being manufactured by the strong acid process. Facilities that process or otherwise use isopropyl alcohol are not covered.
Saccharin (only persons who manufacture are subject, no supplier notification)	81-07-2	**Only** if it is being manufactured.

There are no supplier notification requirements for isopropyl alcohol and saccharin since the processors and users of these chemicals are not required to report. Manufacturers of these chemicals do not need to notify their customers that these are reportable EPCRA

Figure 4.3: The table in Figure 4.2, with the final rows and columns

us to avoid the search procedure getting stuck in local minima and therefore missing important tables.

We put the sorted candidate columns into an *unused column list* and remove the first column from this list. We check if this column intersects any objects, such as horizontal ruling lines[1]. By rectangular containment expansion (see Section 4.3.2.2), we then proceed to find the other items in the table. If no other objects are intersected at this stage, the rectangular containment expansion procedure will fail to add any new objects to the table. We then perform a validation check (see Section 4.3.3.3) to ensure that our candidate table meets the requirements for a (partial) table.

If the validation check is passed, we then attempt to expand the table by adding neighbouring candidate columns to the left and right. We begin by adding columns to the left. After each addition, we perform rectangular containment expansion and then proceed to validate the table. If validation fails, we return the table to the previous state and try expanding to the right. If there are no more items to the left, we also try expanding to the right. When, for one of the same reasons, we cannot expand to the right any more, we revert back to expanding left, as the possibly enlarged bounding box due to rectangular containment expansion may give us additional neighbouring blocks to the left.

When we cannot expand the table any further, we add it to the output result. If the table fails our validation procedure for a *full table* at this stage, all the columns (apart from the first) remain in our unused list. Otherwise, the candidate table is converted to a **Table** object and all the candidate columns are removed from the unused list.

4.3.2.1   Table classification based on ruling lines

We have found that, when horizontal lines are used to delineate rows in a table, they are practically always the entire width of the table. This allows us to terminate the search procedure early once a horizontal ruling line is encountered. First, we apply some preprocessing to all line objects in the document, to merge any dotted or touching lines that have been written in parts to the document. Secondly, we need to ensure that these lines represent ruling lines, and not the underline of headings, or separators for headers or footers on the page. We do this by checking that the line is significantly wider than the widest candidate column it intersects, and does not appear above or below the last or first two lines of paragraph text on the page respectively.

---

[1] We have found that, for tables which contain horizontal ruling lines, these ruling lines usually span the entire width of the table. Thus, the process of rectangular containment expansion allows us to find such tables using only one iteration.

Algorithm 4 The table search algorithm

---

pre: C, the list of unused columns, is sorted according to the columns' width ratio

1. while (C is not empty)

    (a) initialize T as new candidate table

    (b) remove first candidate column c_1 from C and add it to T's items

    (c) enter loop to start expanding T:

        i. first try expanding T by adding neighbours on the left:
            - add the neighbour directly on the left of T to T
            - perform rectangular containment expansion (Alg. 5), on the composite segment T
            - perform the validation procedure (Alg. 7) on T. If this procedure fails, set the contents of T to the last known correctly validated state
            - repeat steps A–C until validation fails or there are no more neighbours to the left of T

        ii. expand T by adding neighbours on the right (the steps are the same as in i.)

        iii. repeat steps i. and ii. until no new candidate columns are added to T

    (d) if T passes the validation check, add T to the output list. Ensure that all items within T have been removed from C

    (e) if T fails the validation check, return all but the first item in T to C

4.3.2.2 Rectangular containment expansion

Up to now, we have always grouped lower-level document objects into higher-level objects based on their neighbourhood. As mentioned in Section 2.2.3, this has allowed us to cope with layouts that are not strictly Manhattan in nature. For tables, we make the very realistic assumption that a table boundary will always be rectangular. This assumption allows us to efficiently build tables from their constituent columns by just considering their joint bounding box, using a procedure which we term *rectangular containment expansion*.

As an example, let us imagine that we have a candidate table that contains two columns. We find the smallest bounding rectangle, or *bounding box*, of this object group, and therefore of the candidate table. We then find that this bounding box intersects other objects on the page, so we proceed to add them to our candidate table, and find its new, enlarged bounding box again. We repeat this procedure until

---

Algorithm 5 Rectangular containment expansion

---

pre: S represents the composite segment to be expanded; P represent the page as a set of all its items $\{p_1, p_2, ..., p_n\}$ at the current level of granularity.

1. foreach p in P:

   (a) if p intersects the bounding box of S and is a text segment or horizontal line:

       i. add p to S
       ii. recalculate the boundary of S, i.e.:
          • if the bounding box of p extends outside of S's boundary, enlarge the bounding box of S such that it completely encloses p

2. if the bounding box of S has been altered in step 1, repeat this step

---

the bounding box is not grown any more, and therefore no more items are added. This procedure is shown in Algorithm 5.

Referring back to the example in Fig. 4.2, the lower table can be found using rectangular containment expansion, once at least one candidate column segment belonging to each of the three columns has been added. As this method does not consider non-rectangular tables, it enables us to explore the search space very efficiently.

4.3.3   Table validation and structure understanding

Once a candidate table is found, we perform the following procedure, which attempts to rediscover the table structure by determining its columns and rows (see Sections 4.3.3.1 and 4.3.3.2 respectively). Once the rows and columns have been determined, the validation procedure is called (see Section 4.3.3.3) to determine whether the table is indeed a real table. The following sections describe the algorithm in more detail. Figure 4.4 summarizes the table understanding and validation process.

Figure 4.4: Process of understanding and validating a table

As well as the result of the validation check (**true** or **false**), this procedure returns the detected rows and columns of the table, according to our table model as defined in Section 4.1.

4.3.3.1   Column finding

The column-finding procedure takes the list of candidate columns as input and attempts to merge them to form complete columns. First, an *adjacency graph* (see Section 3.1.3) is generated on the candidate column list. Then, the vertical edges of the graph are examined in turn. If two candidate columns are joined by a single vertical edge and no further edges exist from either column to another candidate column in the same direction, then these columns are merged together. Otherwise, one of the columns is determined to span more than one horizontal cell and its **colspan** attribute is set accordingly, as shown in the example in Figure 4.5. This procedure is given in Algorithm 6.

Algorithm 6 The column-finding procedure

---

1. create adjacency graph G on candidate columns C

2. let E be the list of southward pointing edges in G

3. while E is not empty

    (a) remove $e = e_1$, the first edge from E

    (b) determine whether e lies along a colspan interface. if it does:

        i. adjust the values of **colspan** for the respective columns accordingly

        ii. if either column does not already exist in the output list, add it to the output list

    (c) else

        i. merge both nodes of e to form a single, larger column and, if it is not already there, add it to the output list

---

4.3.3.2   Row finding

It is not always straightforward to determine where rows begin and end in a table. Many tables use spacing, some use ruling lines, and some use a combination of both. At this stage of the process, we still do not know whether the clusters represent single cells or groups of one-line cells. The row-finding procedure takes as input a list of

table cells; therefore, if the current list of clusters (items of the individual columns) contain more than one line per cluster, they need to be split into their individual cells.

As the ordered-edge segmentation algorithm (see Section 3.4.3) forms clusters based on distance between the segments, the clusters should already represent single rows if white space is used as the row delimiter. If the rows are delimited by horizontal ruling lines, the splitting will have already occurred during the ruling object postprocessing stage (see Section 3.4.4). The remaining class of tables is more difficult: many tables, particularly from the financial sector, consist solely of one-line rows, without any explicit delimiters. Because of the content of the rows (typically numerical data and sometimes alignment), it is obvious to a human reader that each line of text belongs to a separate row, even though no separator is present.

Because such tables can also include horizontal ruling lines to further *group* the rows, we cannot rely on the complete absence of horizontal ruling lines to signify that each line represents an individual row. (The alternative in this case would be a 1-row table, which does not fit our definition of a table.) We therefore examine the textual content in each of the columns to determine whether all multi-line rows need to be split. As long as at least *one* column is found with content that appears to require splitting into individual rows, the entire table is split in this way. We believe this to be similar to the way a human reader analyses such tables; in many examples such as Figure 4.6, it only becomes apparent from the numeric data columns that each line represents a separate row. It would be difficult to come to this conclusion by looking at the leftmost row alone (even if the reader understands Italian).

The heuristic function **splitRows** iterates through the columns of the table and analyses their content. If the average line length (in characters) of the column is below a certain threshold (which, to prevent false positives, depends on the height of the column), the entire table is deemed to require splitting of the rows into individual lines. Nonalphabetic characters, such as numbers, do not count towards the line length measurement, which also allows longer strings of numbers to occur.

It is worth noting that the methods presented in this section do not always correctly find the rows in a table. This is because it is not always possible to detect rows from the layout alone; an analysis of the content and, ideally, domain-specific knowledge is required. Although the function **splitRows** is a step in this direction, these techniques are outside the scope of the work presented in this dissertation. For this same reason, we do not aim to detect spanning rows or distinguish between *access* or *heading cells* and *data cells*.

After the start and end of each row has been determined, the table cells are merged horizontally into rows using the adjacency graph. Further checks on baseline align-

XXXXX XXXXX Spa – data stipula contratto: 06.03.02 – tipologia bene locato: Immobile – durata contratto: 06.02.14 – maxicanone					
Anno	Pagamenti	Onere finanziario esercizio	Impoprto rate non scadute a fine esercizio	Valorizzazione con metodo finanziario	
				Ammor.to (3%) e rettifiche di valore	Valore netto di bilancio fine esercizio
2002	34.457,00	8.115,85	383.358,10	9.000,00	291.000,00
2003	24.262,70	9.874,29	359.095,40	9.000,00	282.000,00
2004	26.468,40	9.901,34	332.627,00	9.000,00	273.000,00

Figure 4.5: Example of spanning columns in a table, which are not merged to their vertical neighbours by the column finding procedure

Esigibili entro l'esercizio successivo:		31/12/2003	31/12/2004
Clienti	€	203.961	166.000
Clienti fatture da emettere	€	60.000	0
Erario c/o Iva	€	23.557	10.781
Erario c/o ritenute su interessi attivi	€	51	181
Note di credito da ricevere	€	3.369	2.346
Acconti a fornitori	€	2.981	24.345
Fornitori c/anticipi	€	0	2.655
Altri crediti v/o erario	€	668	652
Tornano	*€*	*294.586*	*206.959*

Figure 4.6: Example of a table from the financial domain, in which each line of text represents an individual row

Figure 4.7: The table in Figure 4.2, after its conversion into HTML

ment are carried out to help avoid falsely merging rows (e.g. where spacing is tight or where heading cells spanning several rows are aligned differently).

4.3.3.3 Table validation

Once the rows and columns of the table have been found, the table validation function performs a number of checks to determine whether the row-and-column structure resembles an actual table. Tables in documents can occur in a multitude of configurations, sometimes with very complex layouts, and therefore we have designed this function to be very permissive and return a positive result even in cases where a small percentage of rows overlap. This also helps us in situations where the row-and-column structure has not been 100% correctly recognized: the majority of the table will still be correctly extracted in this case.

As candidate tables are validated at each iteration of the search process, the table validation function is also designed to return **true** for partial tables which have not yet completely been found. The conditions for a valid table are shown in Algorithm 7.

Algorithm 7 The table validation function

Return **true** if the following conditions hold, otherwise return **false**:

- the table must have dimensions of at least 2 × 2, otherwise it is considered invalid. If the number of columns *or* rows is less than or equal to 3, the table is deemed less likely to be valid and the *reduced tolerance* flag is set to **true**;

- no table cell may contain more than 6 lines of text (4 with reduced tolerance);

- at least 60% of rows (80% for reduced tolerance) must intersect at least 30% of the columns in the table;

- at least 55% of rows (80% for reduced tolerance) must not intersect another row in the table;

- at least 55% of columns (70% for reduced tolerance) must intersect at least 55% of the rows in the table;

- at least 55% of the columns (70% for reduced tolerance) must not intersect another column in the table.

This way, we ensure that all tables contain a sufficient grid-like structure, whilst avoiding false positives (which, in most cases, occur in paragraph text which has been split up into lines).

4.4 Experimental evaluation

In the OCR domain, an active research field in the previous 30 years, a number of ground-truthed datasets have been made available to researchers for the sole purpose of creating experimental results to enable different systems and approaches from various research groups to be compared with each other. In contrast, in the field of table structure recognition, which is still developing, no such dataset exists, particularly with respect to PDF documents. Although the well-known *University of Washington* datasets do include ground-truthed table areas within the document, they do not include any information on substructures such as rows and columns. Furthermore, the data is in scanned bitmap format and not in PDF.

In this section, we compare the results of our system to a later academic approach, the *PDF-TREX* system [Ruffolo and Oro 2009]. We are very grateful to the authors of PDF-TREX for providing us with a dataset of 75 documents and the output of their system on this dataset, which we have used to compare both systems. This dataset has

since been extended to 100 documents[2], and has recently been made freely available on the Internet [Ruffolo and Oro 2009 (Web)].

The biggest hurdle that we encountered was how to consistently evaluate the various types of structure recognition error (split cell, merged cell, etc.) that occurred. In Section 4.4.1 we describe previous approaches to evaluating such errors and the problems that we encountered. In Section 4.4.2 we propose a classification methodology for each type of error that we encountered, and how it could be consistently evaluated in the future. We hope that this represents a step towards creating common, repeatable experimental results that can be compared between different systems from different research groups.

We also encountered further difficulties in ground truthing of the dataset (Section 4.4.3) and in aggregation of the results for each cell and each table to create a set of figures for the complete dataset (Section 4.4.4). The numerical results of both systems are presented in Section 4.4.5 and a discussion of these results is provided in Section 4.4.6.

4.4.1 Structure recognition issues

A common way to generate numerical values for the performance of table structure recognition algorithms is to borrow the notions of *recall* and *precision* from the information retrieval field [Ruffolo and Oro 2009; Yildiz et al. 2005; Kieninger and Dengel 2001, 2005]. The PDF-TREX system was evaluated in this way, and separate figures for table areas and table cells were generated. The usual definitions of these measures are as follows:

$$\text{Recall} = \frac{\text{number of correctly retrieved data items}}{\text{total number of data items in dataset}}$$

$$\text{Precision} = \frac{\text{number of correctly retrieved data items}}{\text{total no. of retrieved data items}}$$

Essentially, recall measures the proportion of data that has been found correctly without regard to false positives, whereas precision is a measure of how good the algorithm is at avoiding false positives, without regard to recall. Many algorithms can be fine tuned to maximize recall at the expense of precision, and vice versa; and our algorithm is of no exception. The *F-measure*, which is defined as the harmonic

---

[2]The 75 documents used for our comparison correspond to the following documents in the publicly available dataset: 1–12, 14–16, 18–23, 25–34, 37, 38, 40–42, 45–58, 60, 61, 63, 66–69, 71, 72, 75, 79–81, 83, 84, 86, 87, 89, 90, 92–98

mean of precision and recall, is often used as a single-figure measure of the ability of the system.

The biggest problem that arises with this approach is the not unambiguous interpretation of the term *correctly retrieved* in this context. For example, let us consider the simple case where a cell is erroneously split into two cells by the system. Note that precision is usually defined as the proportion of data items that have been *correctly retrieved.* Has the data in the split cell been "correctly retrieved" and do we therefore count these two cells as one true positive and one false positive, or as one false negative and one false positive?

We found that the evaluation strategies used by Ruffolo and Oro [Ruffolo and Oro 2009] and by Kieninger and Dengel [Kieninger and Dengel 2001, 2005] would class at least one cell resulting from the split as having been *correctly retrieved*, even if the data was only partly retrieved (in the latter system, the *best match* according to the sub-objects is found and evaluated as being correct). The remaining cells of the split are classified as false positives, which results in this error affecting overall precision, but not recall.

This simple example highlights the problems with using such a simple model to represent errors in the recognition of more complex structures. In our system, table cells may span multiple columns or consist of several lines of text. We therefore need not only to deal with split and merged cells, but also with incorrectly detected colspans, for example. How should such an error be evaluated in terms of false and true positives?

Unfortunately, most previous publications in table structure recognition do not describe their evaluation strategy in sufficient detail to enable it to be reproduced precisely by a different team of researchers, in order to obtain a fair comparison of both systems.

In Section 4.4.2, we provide a wider range of table area and cell categories to explicitly represent the majority of detection errors that occur. In Appendix A, we provide a classification of all structure recognition errors that we encountered in comparing our system against the PDF-TREX dataset and how they were evaluated according to these categories. We hope that this provides a step towards a common method for evaluating table structure recognition results so that they can be directly compared between systems from different research groups.

4.4.2   A classification scheme for structure recognition errors

As described in the previous section, the relatively simple model of true positives, false positives and missed cells (true negatives) is not expressive enough to fully ex-

press the various types of errors that can occur. We have therefore defined a larger number of table area and cell categories, as shown in Table 4.1, to explicitly represent the majority of errors that can occur. This table also shows whether the category was evaluated as a true positive, false positive, true positive or false negative in our final numerical results. A classification of all types of errors that occurred in comparing both systems against the dataset and how they were evaluated according to these categories is given in Appendix A. After having calculated our initial results, we decided to create a second set of figures that better represented their usefulness for our application (data extraction), by reclassifying certain true positive categories as false positives.

In our results, the first occurrence of a split object was given the classification **split** and the extra occurrences arising from the split were classed as **extra**. By defining the classification **split** as a true positive, we obtain a similar evaluation metric to that used in [Kieninger and Dengel 2001; Ruffolo and Oro 2009].

An important criterion in evaluating table structure recognition errors is the gravity of the error itself, and not just the number of cells that are affected. This may depend on the particular application. Let us consider a further example where a single cell is split horizontally, causing an otherwise blank column to be added in between the data columns in a table. A data extraction algorithm that locates the cells based on their headings will still continue to function correctly for the other cells, as the data cells remain correctly aligned. We therefore introduced two sub-classifications for non-empty **split** cells: **split full** and **split data**. In the former, the textual data within the cell is not split; only extra (false positive) blank cells result; in the latter, the textual data itself is split across several cells. When calculating our results, we first classified both types of split cells as true positives. We then calculated a second set of totals by reclassifying **split data** cells as false positives, which we believe better reflects the usability of the result for our application.

A further question is whether blank cells should be counted at all. Whereas Ruffolo and Oro's evaluation strategy included blank cells, the strategy employed by Kieninger and Dengel appeared not to. As most data extraction applications only use the data cells, results which do not include blank cells in their totals could be seen as being more meaningful. Furthermore, for non-ruled tables, it is not always clear how many "blank" cells they contain, particularly in the case of cells along the edge of the table. In our case, we assumed each table to be rectangular in shape (according to our model), and represented any empty spaces along the table boundary as blank cells. For each set of results, we generated two sets of totals: one including both blank and data cells, and one excluding the blank cells.

Table areas			Data cells			Blank cells	
Found correctly	TP		Found correctly	TP		Found correctly	TP
Data cells found	TP						
Partially found	TP/FP						
Split table	TP/FP		Split full	TP		Split blank	TP
			Split data	TP/FP			
Extra table	FP		Extra data	FP		Extra blank	FP
Incorrect table	FP		Incorrect data	FP		Incorrect blank	FP
Merged into surroundings	TP		Merged	TP/FP			
Merged	TP/FP						
Not recognized	TN		Not recognized	TN		Not recognized	TN

Table 4.1: Classifications for table areas, data cells and blank cells

Regarding table areas, it was noted that, in the PDF-TREX result set, even partly detected table areas counted towards the recall score. Therefore, we first chose to classify **partially found** tables as true positives. Therefore, such an error is only penalized by affecting the cell recall figure. The numbers of **fully** and **partially found** table areas were counted separately. A common error that occurred with many table areas was that all the data cells were found, but the heading cells, which were located some distance from the table body, were not. For data extraction purposes, such a result would be adequate, as it would still be possible to extract all the data from the table. We therefore introduced a further classification, **data cells found**. A similar situation occurred with the classification **merged into surroundings**, where tables were typically merged with neighbouring lines or text above or below, but it was still possible to extract all the data from the table.

The complete list of classifications that we used is shown in Table 4.1. As well as **split data** cells, we also chose to reclassify certain other classifications for our second set of totals as false positives to correspond to a more strict interpretation of the data items having been *correctly retrieved* and better reflect the usability of the result.

4.4.3 Ground truthing issues

The problems inherent in ground truthing tabular datasets are well known and have been described in detail in [Hu et al. 2001b]. In this section, we describe the difficulties encountered in ground truthing the PDF-TREX dataset by the following examples:

- **table headings not properly aligned with the columns containing the data** (Fig. 4.8)**:** in this figure, several figures are misaligned with their headings. For example, it is not immediately clear whether the figure **118.011** belongs to the **Valore iniziale** column, or belongs to its own column. On closer examination, it

VARIAZIONI DELL'ESERCIZIO

VOCE DI BILANCIO	Valore iniziale	Acquisizioni	Alien.e stralci	Rivalut.	Amm.econ.	Storno f.do amm.	Totale
Fabbricati civili	62.911						62.911
Terreni e fabbricati industriali	618.277						618.277
Impianti e macchine		118.011	197			118.208	
Macch. d'ufficio elettroniche	2.535						2.535

Figure 4.8: Example of a table with unclearly aligned columns

	2004	2003	variazioni
Disponibilità liquide	**88.828**	**16.877**	**+ 71.951**
Sono relative a:			
- cassa contante	14.951	274	
- depositi bancari	73.877	16.603	
Totale	88.828	16.877	
	2004	2003	variazioni
Ratei e risconti attivi	**22.794**	**22.781**	**+ 13**

Figure 4.9: Example of a table split by intermediate headings

	2007			2008		
Previsioni di inflazione nell'area dell'euro dei principali organismi internazionali (1)						
	FMI *(set. 2006)*	OCSE *(dic. 2006)*	CE *(feb. 2007)*	FMI *(set. 2006)*	OCSE *(dic. 2006)*	CE *(nov. 2006)*
Italia	2,1	1,9	1,9	..	2,0	1,9
Francia	1,9	1,4	1,5	..	1,6	1,9
Germania	2,6	1,9	1,7	..	1,0	1,2
Spagna	3,4	2,7	2,5	..	3,2	2,7
Area euro	2,4	1,9	1,8	..	1,8	1,9

Fonte: FMI, Ocse e CE.
(1) Previsioni effettuate nel mese indicato fra parentesi.

Figure 4.10: Example of a table with spanning column headings

appears that this figure was mistakenly right aligned. Similarly, the erroneous left alignment of the column heading **Totale** causes confusion;

- **tables being split by intermediate headings** (Fig. 4.9)**:** are these separate tables, or do these subtables all belong to one single table? If the table was not interrupted by paragraph text, it was interpreted as a single table, which also corresponds to the interpretation used by Ruffolo and Oro. However, in this case, the following problem arises:

- **spanning column headings in non-ruled tables** (Fig. 4.10)**:** here, one often cannot tell from the layout alone how many columns are spanned by the text. Although the text may only be two columns wide, it could be seen to logically span all data cells or even the entire width of the table. Even with domain-specific knowledge, this can present an ambiguous situation;

- **spanning row headings in non-ruled tables** (Fig. 4.11)**:** here, the layout of the table might suggest that the heading of a group of rows only belongs to the top row of the group. But logically, the heading applies to the row(s) underneath too. In the example, the year and months apply equally to the rows following them;

- **other tabulated data with leading dots** (Fig. 4.12)**:** in this example, the page contains two ruled tables and additional tabulated data inbetween these tables, which is presented using leading dots. This special type of formatting is usually reserved for special use-cases such as tables of contents and indices in books. Because the data presented is of a tabular nature, we did consider this to be a table in our ground truth, in contrast to Ruffolo and Oro. However, because this was one formatting convention we did not consider when designing our algorithm, this table was not detected at all by our system;

- **one line wrapped from previous table** (Fig. 4.13)**:** here, is appears that a single row (the "total" row) of a table on the previous page was wrapped over to the current page. Because we define tables as having a minimum dimension of 2×2, we did not class this "orphan row" as a table. This also corresponds to the decision made by Ruffolo and Oro.

We found that many of the above problems, such as misaligned columns and orphan rows, occurred due to poor, unprofessional typesetting of the documents in question. Some of these documents even proved troublesome for a human reader to understand, who could at least use his domain-specific knowledge to help the understanding process when the underlying logical structure cannot be determined from the

2000 – gen.	393	13.153	2.506	707	139	16.898
	(203)	*(6.793)*	*(1.294)*	*(365)*	*(72)*	*(8.727)*
feb.	341	12.286	2.957	753	124	16.460
	(176)	*(6.345)*	*(1.527)*	*(389)*	*(64)*	*(8.501)*
mar.	331	11.294	2.689	889	196	15.399
	(171)	*(5.833)*	*(1.389)*	*(459)*	*(101)*	*(7.953)*
apr.	447	10.353	2.169	815	167	13.947
	(231)	*(5.347)*	*(1.120)*	*(421)*	*(86)*	*(7.203)*
mag.	416	11.447	2.446	1.125	147	15.620
	(215)	*(5.912)*	*(1.263)*	*(581)*	*(76)*	*(8.067)*

Figure 4.11: Example of spanning rows in an unruled table

Addizionale regionale Irpef	127,00	0	127,00	0
Erario c/rit. Amm.ri	1.215,03	0	1.215,03	0
Totale	**21.210,41**	**1.458,12**	**19.284,63**	**3.383,90**

In merito alle imposte sul reddito dovute dalla società si precisa quanto segue:

Imposta Ires

Imposta IRES dovuta. 8.014,00

Erario c/r.a. subite. 9,18

Erario c/ r.a. provv. 1.503,72

Acconti Ires versati anno 2004 . . . . .11.193,93

Credito Ires anno 2004 . . . . .4.693,00

Figure 4.12: Example of tabular data laid out using leading dots

COOPERATIVA SOCIALE XXXX XXXX A R.L.

| **TOTALE** | **118.088,00** | **5.448,00** | **24.370,00** | **99.166,00** |

B.2 **IMMOBILIZZAZIONI MATERIALI**

L'ammortamento delle immobilizzazioni materiali, la cui utilizzazione è limitata nel tempo, è stato operato in conformità al presente piano prestabilito:

Figure 4.13: Top of a page containing a one-line table, an "orphan row" wrapped from the previous page

layout alone (or, worse still, when the visual cues suggest a different logical structure to the correct one).

When we originally designed our system, we made two assumptions: Firstly, the input documents are correctly typeset and adhere to common layout conventions. Secondly, the logical structure of the data can be fully inferred from its layout. We found the PDF-TREX system to operate in a similar way, as it also encountered problems with the same documents.

We therefore pose the following questions for consideration regarding the dataset:

- Should documents containing obvious errors in their typesetting or layout (such as misaligned columns) be eliminated from the dataset?

- How do we deal with documents that have more than one correct interpretation of the ground truth?

- Perhaps a subset of documents could be defined, which are not reliant on domain-specific reasoning to be understood, and could be used to test systems which rely purely on the document's geometric structure. This could ensure less "noise" in the test results.

4.4.4 Aggregation of results

In common with a previous publication [Yildiz et al. 2005], the results of the PDF-TREX system were given using separate precision and recall values for tables and cells. Here, the authors used a *document-based* approach: they first calculated the average table area and cell recall and precision for each document, and averaged these figures throughout the complete dataset. It is, however, not possible to calculate precision values for documents where no tables or cells were detected at all. We therefore decided to skip the step of calculating the averages for each document and create our totals by averaging the total numbers of detected cells directly over the complete dataset. With this method, documents containing more information (more table areas/cells) are also given more weighting in the final result.

A number of other approaches have also been proposed in the literature for aggregating the results of table structure recognition algorithms on different levels of granularity. [Kieninger and Dengel 2001, 2005] propose a hierarchical model for representing the recognition result and the ground truth and redefine *table recall* and *table precision* based on their constituent objects. Thus, single values for recall and precision are returned. Because a strict hierarchy is used, only columns *or* rows can represented;

the authors choose to represent columns as this better represents how their algorithm works.

A potential issue arises in the bottom level of the hierarchy, which is stated to be the *word level*. As the precise granularity or bottom-level segmentation may differ across different systems, difficulties could arise when comparing different systems to each other.

[Hu et al. 2002, 2001a] use a *directed acyclic graph* structure to represent the recognition result and ground truth. This structure can be used to represent both rows and columns simultaneously. Because of the complexity of the graph isomorphism problem, the two graphs are compared by a sequence of *random graph probing* operations, which need to be carefully defined according to the measuring criteria. Although such an approach is well suited for automatic tuning of algorithm parameters for a particular application, it is less suitable for comparing different systems to each other, not least because of the random element of this approach.

A further noteworthy approach is that of [Cesarini et al. 2002], who provide a formula for the *Table Location Index*, which combines correctly located, split and merged tables into a single score, and is used for automatic optimization of their algorithm. However, this approach does not deal with table cells, but only with table areas.

4.4.5 Numerical results of both systems

The precision and recall measures of both systems are shown in Table 4.2.

4.4.6 Discussion

Broadly speaking, the results show that whereas the PDF-TREX system achieves better cell recall, our system achieves better table area recognition results and better precision (i.e. fewer false positives) overall. The largest differences can be observed in the precision of table areas and the recall of table cells. After redefining certain cell classifications as false positives, a significant decrease was noticed in the numerical results of the PDF-TREX system. This is because the new definitions give a higher penalty to errors which would likely hinder data extraction. Excluding blank cells led to higher numerical results for precision and recall, and gave the PDF-TREX system a slight advantage.

During testing, it became clear that many documents which caused problems for our system also caused problems for PDF-TREX and vice versa, which suggests that both systems work in a similar way. It is believed that a small amount of fine-tuning

Table areas before reclassification:

	Recall	Precision	*F*-meas.
Our system	93.0%	78.8%	85.3%
PDF-TREX	87.5%	61.5%	72.3%

Table cells before reclassification:

	Including blank cells			*Excluding blank cells*		
	Recall	Precision	*F*-meas.	Recall	Precision	*F*-meas.
Our system	87.3%	86.7%	87.0%	88.3%	96.1%	92.0%
PDF-TREX	96.5%	80.7%	87.9%	97.2%	94.2%	95.7%

Table areas after reclassification:

	Recall	Precision	*F*-meas.
Our system	76.9%	66.7%	71.4%
PDF-TREX	67.7%	47.6%	55.9%

Table cells after reclassification:

	Including blank cells			*Excluding blank cells*		
	Recall	Precision	*F*-meas.	Recall	Precision	*F*-meas.
Our system	87.3%	86.7%	87.0%	86.2%	96.0%	90.8%
PDF-TREX	93.5%	78.1%	85.1%	94.1%	91.0%	92.5%

Table 4.2: Precision and recall results of both systems for table areas and table cells

of both algorithms, for example by adjusting thresholds or by trading off precision for recall, could lead to significantly better numerical results. The higher table cell recall of PDF-TREX could partly be attributed to the fact that several large tables were not detected at all by our system; these same tables were detected by PDF-TREX but split into several individual tables, which explains its significantly worse table area precision.

The significant drop in table area precision of PDF-TREX after reclassifying **merged tables** as false positives could be explained by one particular document in the dataset, which contained 12 tables on one page. Whereas these tables were detected correctly by our system, they were erroneously merged together into a single table by PDF-TREX. The fact that our results were not generated by averaging the results for each document, but were averaged directly over the complete dataset, means that this error was given a much larger weighting in the final result.

Both systems showed a significant weakness in the detection of tables in *sparse* layouts. In our case, this was because the table searching procedure finds tables only by merging adjacent candidate columns to the left and right of the table. In a sparse layout, these candidate columns are sometimes not found in full, but broken up. This problem could be solved in one of two ways: either by a threshold-adjustment heuris-

tic based on a measure of the sparseness of the page, or by an additional vertical merging procedure.

Furthermore, a proportion of these documents also contained significant typesetting errors, which made logical structure detection difficult even for a human reader, who at least has the ability to use domain-specific knowledge. This class of documents was not considered in the initial design of our approach.

Finally, although every effort was made to ensure a fair test with the data, ultimately the PDF-TREX system had an inherent advantage, as its developers had provided the test dataset and therefore had more opportunity to develop and test their system on it. In an ideal scenario, these tests would be performed on an independently generated dataset, which none of the developers have seen beforehand, in a similar way to the *ICDAR Page Segmentation Competition* [Antonacopoulos et al. 2009].

4.5   Conclusion

In this chapter, we have presented an algorithm for table detection and table structure recognition from PDF files, which achieves good results on a wide variety of documents, and can be used as an input filter for the *Lixto Visual Wrapper* (see Chapter 5) to wrap data from tabular documents. It was compared to another, newer system in the literature and was found to achieve comparable results, but with increased precision. Further improvements to the algorithm could improve the detection of tables in sparse layouts.

We have also addressed the significant issue of *evaluating* systems for table detection and structure recognition. The use of precision and recall measures from the information retrieval field to model errors in table structure recognition, as in [Yildiz et al. 2005; Ruffolo and Oro 2009], can lead to many ambiguities. It is hoped that the extensive discussion in Section 4.4 will lead to a more consistent interpretation of these measures in the future, enabling the results of competing systems to be compared directly.

As already mentioned in Section 3.7, the detection of structures on a page relies on using knowledge at multiple granular levels of the page simultaneously. Although the search procedure operates only at the table column level, the use of the *validation procedure* in our algorithm (see Section 4.3.3.3) checks the table's objects at several granular levels to deem whether a valid table has been found. We believe that a more precise hierarchical model of the composition of a table, coupled with the ability to perform specific operations to correct common detection errors, such as in [Wang 2002], could lead to further performance benefits.

A further limitation of this approach is that it does not have the ability to correct errors made in the previous processing stage, page segmentation, nor can any uncertainties in the table detection process be conveyed to the following stages, such as wrapping directly on the tabular structure. This problem is discussed in detail in Section 7.2, where suggestions for future work are given.

Chapter 5

Wrapping using the *Lixto Visual Developer*

The *Lixto Visual Developer*, or *VD*, enables wrappers for HTML pages to be generated in a visual and interactive manner. The user interface uses the Mozilla engine to display a preview of the page, with which the user can interact. Wrappers are defined by hierarchically structured *patterns* to select data on the page, which can either be made available to the next hierarchical level or written directly to the output XML file.

Generally, wrapping tables in HTML documents is a relatively simple approach. After selecting the table(s) in a root pattern, first the rows (`<tr>`) are wrapped in a subpattern and then the individual cells (`<td>`) in the pattern below. Depending on the desired result, the user can choose to select either all or only particular columns from the table. Our conversion process does not distinguish between table heading and table data elements and all cells are represented using `<td>` elements. A screenshot of the Lixto VD in action is shown in Figure 5.1. In the pattern being shown, the rows of a table are being selected.

Our conversion approach is implemented by a plugin, which generates an XHTML file that is used as input to the Lixto VD. The default behaviour is to detect tables on the page, and include all text blocks as well as tables that have been found. Text blocks are represented as `<h1>` or `<p>` elements, depending upon their size, and tables are represented as HTML `<table>` elements.

In order to improve the presentation of the document, a number of heuristics have been employed to detect headings on the page and attempt to find the correct reading order of the blocks. As the focus of the system is on extracting data from tables, these heuristics are based on simple spatial relations, and may fail on more complex

layouts. Using parameters, the user can choose to hide all non-`<table>` objects and also has the option to force treatment of entire pages as tables. This option is useful in cases where the table detection algorithm produces split tables or fails to detect the table at all.

5.1   Case study example: Statistik Austria

The conversion approach to wrapping produces successful results for tables with a clear and consistent visual structure, which are suited to automatic detection and structure detection algorithms. As an example, the *Statistik Austria* website[1] contains large amounts of statistical data in PDF, some (but not all) of which is available in other formats, such as HTML. To demonstrate the conversion-based wrapping approach, a representative 18-page document [Statistik Austria 2009 (Web)] containing 669 data records in a table spanning over all these pages was chosen. Figure 5.2 shows a sample of the data in this document.

Our table detection algorithm was not found to work perfectly on this document. Only the data cells, and no heading cells were detected. Due to text in the final two columns being very narrowly spaced, these two columns were merged into one single column. However, it was possible to cope with both errors in the Lixto VD, as these errors occurred uniformly throughout the entire document.

As we were only interested in extracting the data cells of the document, it was actually beneficial that the heading cells were not detected. Otherwise, additional conditions would have been necessary to exclude the heading cells. As the data in the merged final two columns was separated by a space, a regular expression was used to extract the data belonging to each respective cell. Even if this error had not consistently occurred throughout the document (but only from time to time), it would still have been possible to define the wrapper to deal with *both* situations and produce correct output.

A number of issues also arose in *ground truthing* the original document. The first column actually contained two items of data, a code and a description (**Klassennummer** and **Kurzbezeichnung**), which were also extracted separately using regular expressions. Furthermore, after analysing the column headings it was found that they also did not have a flat structure: the headings **Beschäftigte im Jahresdurchschnitt** and **Waren- und Dienstleistungskäufe** actually apply not only to their particular column but also to the following column. The wrapper design was modified to reflect this hierarchical structure.

---

[1]*Statistik Austria*, `http://www.statistik.at`

Figure 5.1: Interactive wrapper creation using the *Lixto Visual Developer*

In this way, it is possible to wrap tables from PDF documents which have a clear structure and consistent formatting using the conversion-based approach. If not all cells are required, of course only the required data could be selected. The next subsection details the steps that were necessary in creating this wrapper.

5.2 Step-by-step wrapper creation

The following interaction steps were necessary to create the wrapper described in Section 5.1:

1. Create a new **URL action** to open (and import) the desired PDF file into the Lixto VW. **Autotable** mode must be selected to enable table detection

2. Create a new **data extractor**

3. First, create a new **pattern** to extract all the relevant tables:

Klassen Nummer und Kurzbezeichnung (ÖNACE 2003)	Unternehmen	Beschäftigte im Jahresdurchschnitt 2007 insgesamt	darunter unselbst.	Personalaufwand in 1.000 EUR	Erlöse und Erträge in 1.000 EUR*	Umsatzerlöse in 1.000 EUR*	Produktionswert in 1.000 EUR*	Waren- und Dienstleistungskäufe 1) insgesamt in 1.000 EUR*	dar. zum Wiederverkauf in 1.000 EUR*	Bruttowertschöpfung zu Faktorkosten in 1.000 EUR*	Bruttoinvestitionen in 1.000 EUR*	Code
INSGESAMT	**294.099**	**2.611.293**	**2.357.784**	**93.759.729**	**657.051.122**	**613.598.729**	**380.324.893**	**415.539.018**	**205.286.689**	**162.797.470**	**40.299.429**	
C Bergbau und Gewinnung von Steinen u.Erden	**345**	**6.157**	**5.949**	**326.821**	**2.166.102**	**2.075.938**	**1.958.522**	**1.200.605**	**173.404**	**867.524**	**433.773**	**C**
10 Kohlenbergbau, Torfgewinnung	**6**	**G**	**G**	**G**	**G**	**G**	**G**	**G**	**G**	**G**	**G**	**C10**
103 Torfgewinnung und -veredlung	6	G	G	G	G	G	G	G	G	G	G	C103
1030 Torfgewinnung und -veredlung	6	G	G	G	G	G	G	G	G	G	G	C1030
11 Erdöl- und Erdgasbergbau	**7**	**1.015**	**1.012**	**94.095**	**1.014.425**	**978.220**	**882.691**	**468.033**	**109.221**	**472.639**	**279.461**	**C11**
111 Erdöl- und Erdgasbergbau	3	1.001	1.001	93.710	1.006.921	970.991	875.737	464.079	109.147	469.829	278.006	C111
1110 Erdöl- und Erdgasbergbau	3	1.001	1.001	93.710	1.006.921	970.991	875.737	464.079	109.147	469.829	278.006	C1110
112 Erbr.v.Dienstl.f.d.Erdöl-u.Erdgasbergb.	4	14	11	385	7.504	7.229	6.954	3.954	74	2.810	1.455	C112
1120 Erbr.v.Dienstl.f.Erdöl- u.Erdgasbergb.	4	14	11	385	7.504	7.229	6.954	3.954	74	2.810	1.455	C1120
13 Erzbergbau	**2**	**G**	**G**	**G**	**G**	**G**	**G**	**G**	**G**	**G**	**G**	**C13**
131 Eisenerzbergbau	2	G	G	G	G	G	G	G	G	G	G	C131
1310 Eisenerzbergbau	2	G	G	G	G	G	G	G	G	G	G	C1310
14 Gew.v.Steinen u.Erden, sonstiger Bergbau	**330**	**4.968**	**4.767**	**222.867**	**1.120.515**	**1.070.482**	**1.045.983**	**713.731**	**64.167**	**384.893**	**151.863**	**C14**
141 Gewinnung von Natursteinen	48	578	550	23.078	88.804	79.721	80.100	44.430	2.049	36.484	12.822	C141
1411 Gewinnung v.Naturwerksteinen	40	499	475	19.589	67.695	65.676	65.125	36.193	1.995	29.875	11.343	C1411
1412 Gew.v.Kalk, Dolomit, Gips u. Anhydrit	8	79	75	3.489	21.109	14.045	14.975	8.237	54	6.609	1.479	C1412
142 Gewinnung von Kies, Sand, Ton u. Kaolin	267	3.758	3.589	162.524	876.331	843.187	816.659	561.546	50.606	296.291	95.919	C142
1421 Gewinnung v.Kies und Sand	262	3.686	3.523	160.288	868.238	835.277	810.480	557.103	48.722	292.725	95.320	C1421

Figure 5.2: An example of the tabular data in the Statistik Austria test document

```xml
    - <waren-_und_dienstleistungskaeufe>
        <insgesamt>G</insgesamt>
        <darunter_zum_wiederverkauf>G</darunter_zum_wiederverkauf>
      </waren-_und_dienstleistungskaeufe>
      <bruttowertschoepfung_zu_faktorkosten>G</bruttowertschoepfung_zu_faktorkosten>
  </item>
  - <item>
      <klassennummer>11</klassennummer>
      <kurzbezeichnung>Erdöl- und Erdgasbergbau</kurzbezeichnung>
      <bruttoinvestitionen>279.461</bruttoinvestitionen>
      <code>C11</code>
      <unternehmen>7</unternehmen>
    - <beschaeftigte_im_jahresdurchschnitt>
        <insgesamt>7</insgesamt>
        <darunter_unselbstaendig>1.015</darunter_unselbstaendig>
      </beschaeftigte_im_jahresdurchschnitt>
      <personalaufwand>94.095</personalaufwand>
      <erloese_und_ertraege>1.014.425</erloese_und_ertraege>
      <umsatzerloese>978.220</umsatzerloese>
      <produktionswert>882.691</produktionswert>
    - <waren-_und_dienstleistungskaeufe>
        <insgesamt>468.033</insgesamt>
        <darunter_zum_wiederverkauf>109.221</darunter_zum_wiederverkauf>
      </waren-_und_dienstleistungskaeufe>
      <bruttowertschoepfung_zu_faktorkosten>472.639</bruttowertschoepfung_zu_faktorkosten>
  </item>
  - <item>
      <klassennummer>111</klassennummer>
      <kurzbezeichnung>Erdöl- und Erdgasbergbau</kurzbezeichnung>
      <bruttoinvestitionen>278.006</bruttoinvestitionen>
      <code>C111</code>
      <unternehmen>3</unternehmen>
    - <beschaeftigte_im_jahresdurchschnitt>
```

Figure 5.3: The resulting XML output from the *Lixto Visual Developer*

(a) Select the pattern's **filter**

(b) Click anywhere on the table to select. A table cell should be selected

(c) Click the up arrow, which removes the last element of the XPath expression. Repeat until the entire table is selected

(d) Under **occurrence type**, select **multiple**

(e) Name this pattern **table**

4. Follow step 3 to create a pattern to select each record (row) in the table, as a child pattern of the **table** pattern. Name this pattern **item**

5. For each desired column of the table, create a pattern as a child of the **item** pattern. This time, set the **occurrence type** to **single**

6. If any hierarchical groupings are required, create a "dummy" pattern for these groupings and leave the filter blank

7. To split a column that has been merged, create two sub-patterns. For each of these sub-patterns:

(a) Select the pattern's **filter**

(b) Change the type to **text** and select **regexp**

(c) Enter the desired regular expression

8. For each of the leaf patterns, as well as any desired groupings, check the box headed **Write to output**

9. The wrapper is now ready to be executed

5.3 Discussion

In this chapter we have shown how the methods developed in Chapters 3–4 can be used to wrap tabular data with the Lixto Visual Developer. Apart from the final record, which was split up into two records due to a layout error in the original document, the wrapper was able to extract the remaining 668 records correctly and completely.

An obvious drawback of the Lixto import function is that the resulting HTML document does not visually resemble the PDF original. As the visual coordinates of each detected text block are known, it would be possible to develop a "page view" in the Lixto VD, which would allow the user to interact directly on a bitmap rendition

of the original document. The detected tables and other structures could then be shown directly on this rendition. This could result in an improvement to the usability of the system. However, care must be taken that it does not confuse the user, as the underlying HTML structure, which does not necessarily represent the structure of the document image, would then be somewhat hidden. Using this wrapping approach, it would not be possible to completely do without the HTML view, particularly in cases where detection errors such as merged columns result.

Ultimately, there are a number of drawbacks of wrapping data from PDF files using HTML as an intermediate format. Not only is it unintuitive for the user to work with, it also does not allow us to express all the relationships we might need to locate wrapping instances. The user is limited to using elements of the rediscovered logical structure, such as headings and tables; explicit *geometric* relations between objects are unavailable. Furthermore, it is crucial that the document understanding process detects these structures and converts them into HTML correctly. Although the example with the Statistik Austria document has shown that certain minor errors can be coped with, document understanding is inherently an inaccurate process. If a table is only partly detected or not detected at all, as can happen with more complex layouts, its data cannot be extracted.

The next chapter presents the graph-based approach to wrapping, which enables wrappers to be defined *more directly* on the visual structure of a PDF document. This approach is designed to address these limitations of the conversion approach to wrapping.

Chapter 6

The graph-based approach to wrapping

This chapter describes a novel approach to user-guided data extraction, which is based on the *adjacency graph* representation described in Section 3.1.3. The idea of using graph-based techniques for wrapping was first proposed in our poster paper [Hassan and Baumgartner 2006] and a prototype system was published in [Hassan 2009c,b]. This chapter includes content from all of these publications, as well as several yet unpublished improvements to the algorithm.

6.1   Background

Up to now, the work described in this thesis was carried out with one specific goal in mind: to recognize structures such as tables in such a way as to enable a good conversion from PDF into HTML, in order to enable data to be extracted using a tool such as the *Lixto Visual Wrapper*. Although a large proportion of documents can be wrapped this way, this approach has some obvious drawbacks, as explained in Chapter 5:

- it is unintuitive for the user to work with HTML as an intermediate format as its appearance does not closely resemble that of the original document;

- HTML does not allow us to express all the relationships we might need to locate wrapping instances;

- it is crucial that the document understanding process detects the desired structures and converts them to HTML correctly. As document understanding is inherently an inaccurate process, this approach lacks robustness.

We therefore decided to experiment with a more direct way to wrap data from the document. Inspired by the graph-based representation in Section 3.1.3, we decided to use this structure as a basis for the investigation of graph-matching techniques to locate wrapping instances. This approach addresses the concerns above, specifically:

- our graph-based representation very closely resembles the physical structure of the page and is much more intuitive for the user to work with (see Section 6.4.1);

- our adjacency graph representation can be extended by the addition of logical and semantic relationships, enabling our wrapper generation language to be as expressive as necessary;

- most wrapping applications can be performed solely by using adjacency or part adjacency/part logical relationships and are not reliant on complete structures, such as tables, being correctly detected. The graph-based approach has a much higher chance of success in cases where the document understanding algorithms do not perform correctly. Furthermore, it enables a wider range of documents, such as the example in Figure 6.1 with irregular structures, to be wrapped.

A closer investigation of the wrapper generation process has enabled us to identify the three main data structures within a PDF document that could be used to locate instances of data to be wrapped:

- geometric structure (explicit in the coordinates of each object)

- logical structure (inferred from the layout)

- content and content attributes (the text itself, as well as font, style, size, etc.)

Whilst our HTML conversion allows us to use the content and logical structure to identify wrapping instances, it does not give us direct access to the document's geometric structure. The graph-based approach described in this chapter allows a combination of all three of these structures to be used, essentially shifting some of the burden of the document understanding process to the user, which compensates for the inherent inaccuracies and limitations of document understanding. The examples in this chapter only use the geometric structure and content attributes for wrapper definition; the relatively straightforward extension to logical relationships is proposed as further work.

6.2   Technical implementation

This section describes how our prototype system for wrapper generation using graph matching was implemented. Section 6.2.1 describes how the graph structure is generated from a page of a document. Section 6.2.2 introduces the problem of *graph matching* and gives a brief overview of relevant literature in the field. The remainder of the section presents our algorithm for locating wrapping instances, which is based on the Ullmann algorithm [Ullmann 1976].

6.2.1   Creation of graph structures

In Section 3.1.3 we developed the *adjacency graph* representation of a document, which represents all segments at a given granularity as nodes in a graph. Two nodes are joined with an edge if their respective segments are *direct neighbours* of each other. These edges are annotated with their respective directions: **north**, **south**, **east** or **west**. The precise definition of direct neighbourhood and the algorithm which determines the direction label are given in Section 3.1.3. As two of the directions are direct opposites of the other two, we choose to only explicitly represent the **south** and **east** directions.

When the user interactively selects the desired example instance on the graphical user interface (see Section 6.4.1), the underlying sub-graph of the document graph is found. The process is as follows:

- all the nodes whose centre co-ordinate (**Xcen**, **Ycen**) intersects the marquee box are found and form the nodes of the example graph;

- all the edges in the document graph which join any two nodes, both of which are in the example graph, are added to the example graph.

The example graph is then furnished with the respective attributes of its nodes and edges and is known as an *attributed relational graph* (ARG). For horizontal edges, the *edge length* is equal to the shortest horizontal physical distance between the nodes that it joins. For vertical edges, the distance between the respective baselines is used.

This graph is then shown to the user in the right-hand panel of the user interface. The user can now make changes, such as adding or removing nodes or conditions. For the remainder of this chapter, we will use the terms **document graph** for the ARG of the document and **example graph** for the ARG of the example instance, which is a subgraph of the document graph in which extra wrapping conditions may be defined. See Figure 6.8 for a screenshot of the user interface. Essentially, the example graph defines the wrapper.

91

Figure 6.1: Example of a document containing wrappable information with an irregular layout. The brackets represent the individual records

6.2.2   Introduction to graph matching

In recent years, the use of **graph matching** techniques in pattern recognition applications has become more widespread. In particular, graph matching has found several uses in computer vision applications, e.g. [Li and Lee 2000; Belongie and Malik 2000]. In document processing, graph matching has also been used for optical character recognition [Lee and Liu 1999; Suganthan and Yan 1998] and symbol recognition [Lladós et al. 2001; Changhua et al. 2000]. A related technique, *graph partitioning*, has also been applied to clustering documents for information retrieval [Dhillon 2001; Zamir and Etzioni 1998]. We believe that the method presented in this chapter is the first application of graph matching techniques to data extraction from documents on the *physical level*. The term *graph matching* is indeed very broad and imprecise, encompassing a number of techniques for the structural comparison of (sub)graphs. Conte et al. describe graph matching as:

> ... the process of finding a correspondence between the nodes and the edges of two graphs that satisfies some (more or less stringent) constraints ensuring that similar substructures in one graph are mapped to similar substructures in the other. ([Conte et al. 2004], p. 2 [266])

The most stringent form of graph matching is *graph isomorphism*. This form of matching searches for a correspondence between each node in both graphs. These correspondences must be one-to-one. Furthermore, the mapping must be *edge preserving*, i.e. if two nodes in the first graph are linked by an edge, the corresponding nodes in the second graph must be linked by an edge as well.

It is clear that such a matching procedure is not very useful for our application as our example graph, for all practical purposes, must be smaller than the document graph against which it will be matched. Thus, we require an algorithm for *subgraph isomorphism* which finds not one, but *all* possible matching subgraphs in our document. Only this way can we find repeating data sets such as table rows, etc.

Furthermore, since we are working with attributed relational graphs, the edges in our graphs also convey significant meaning. Therefore, it is not enough that the matching be edge preserving; the relationship represented by the edge in the corresponding document graph must also somehow match the relationship expressed by the edge in the example graph. We refer to this type of matching as *attributed matching*. It is also worth noting that this necessitates an algorithm suitable for matching *directed* graphs.

Finally, since we are matching datasets where the structure can differ slightly, our procedure needs to allow for the fact that the underlying graph structure will not

```
0 000 007   QSP-KOSTEN
                 TE-MIN:    90                            KOST:1733

    FEHLER          : Für den entstandenen Aufwand werden Ihnen die aufge-
                      führten TE-Minuten berechnet. (90 TE-Min. = 73,50 EUR)

    ENTSCHEIDUNG    : Bearbeitungsaufwand


6872134/04  LU LI FEDERBEIN-STOSSDAEMPFER VO
    TC716M       TE-MIN:    30                            KOST:4135

    FEHLER          :     1X Passt nicht
                      AVS ID: 1903817112
                      Falscher Barcodeaufkleber 619137134 vorhanden
                      TE:        30

    FEHLERDATUM     : 19.11.2005

    ENTSCHEIDUNG    : Rücklieferung oder Gefahrgut
```

Figure 6.2: Example of error tolerance: in the two records shown above, the row headed **Fehlerdatum** is missing from the first record.
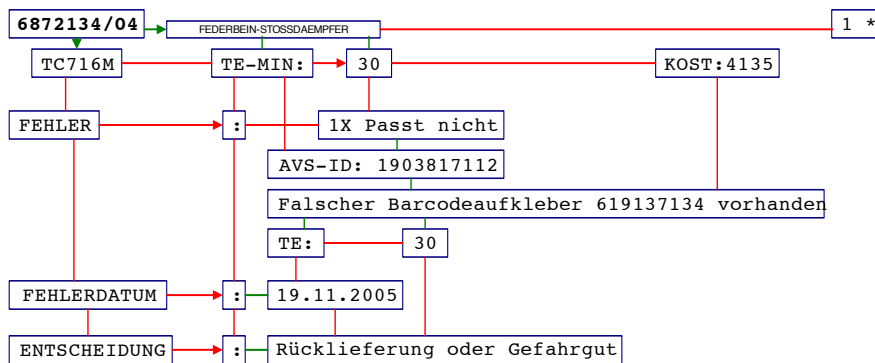

Figure 6.3: Graph structure of the bottom record in Figure 6.2

always be exactly the same. For example, in the layout of a given record, a field present in the original wrapper definition may be missing or may wrap to two lines instead of one. Or, because of the shorter length of an entry, certain edges between nodes may be missing. One such example is shown in Figure 6.2. Our graph matching procedure must allow for such situations.

There is a special class of matching algorithms known as *error tolerant* algorithms, which aim to find inexact matches where some correspondences between both graphs are missing. However, by using the techniques detailed in Section 6.3, which allow the wrapper designer to *specify* the structural changes that can occur, we have found it unnecessary to resort to an error tolerant algorithm with its additional computational complexity.

Many of these graph matching problems, including subgraph isomorphism, are known to be *NP complete*. We have chosen to build our wrapping method around

the Ullmann algorithm because of its flexibility; this is described in more detail in the next section. [Messmer 1996] (p. 134) states the worst-case runtime of the algorithm as $O(m^n n^2)$ for a model graph with m and an input graph with n vertices. Due to the limited size of the graphs in our application, the algorithm's performance was found to be more than adequate in practice. Many other algorithms have been proposed in the literature, which offer reduced complexity but often present further restrictions to the graphs being matched, making them difficult to adapt to our application. To name just a few examples, [Hopcroft and Wong 1974; Aho et al. 1974] propose polynomial time algorithms for planar graphs and trees respectively. [Messmer and Bunke 1999; McKay 1980] propose efficient algorithms for comparing a large library of graphs against a single model graph. [Cordella et al. 2004] presents an efficient algorithm for (sub)graph isomorphism suitable for matching large graphs. As this method also supports attributed matching, it presents a viable alternative to the Ullmann algorithm if computational complexity becomes a problem. A detailed summary on graph matching algorithms and their applications in pattern recognition and related fields can be found in [Conte et al. 2004].

6.2.3 The Ullmann algorithm

The Ullmann algorithm [Ullmann 1976] was chosen as a basis for our wrapping method because of its flexibility and suitability as a basis for our application. It offers the following advantages:

- the algorithm can be modified to match *directed graphs* (this modification is given in the paper);

- the algorithm finds *all* possible *subgraph isomorphisms* in the input (document) graph;

- the algorithm can be relatively easily modified to support *attributed matching*, i.e. the use of *attributed nodes* and *edges* with *matching conditions*;

- at the start of the algorithm *and at each step*, it is possible to determine exactly which nodes and edges have been matched or are still being considered for matching. Thus, it is possible to prune unlikely matches immediately from the search tree to keep time and space complexity at a minimum.

This last point is very important: the basic algorithm is simply a brute-force enumeration of all possible combinations of nodes and edges. Even on a modern computer, it becomes inpractical to execute this algorithm on all but very small graphs. The

refinement procedure, which prunes impossible search paths, results in a dramatic re-
duction in search space and processing time. Should this procedure not suffice, fur-
ther application-specific heuristics can be introduced to reduce the search space even
further, although these might not guarantee that all subgraph isomorphisms will be
found.

The basic algorithm is simply a tree search of the entire search space. In this chap-
ter, will use the same terminology as in [Ullmann 1976]. The algorithm is designed to
find all the isomorphisms between a given graph $G_\alpha = (V_\alpha, E_\alpha)$ and subgraphs of a
further given graph $G_\beta = (V_\beta, E_\beta)$ where V and E refer to the *points* and *lines* of the
respective graph. The numbers of points of G_α and G_β are p_α and p_β respectively. The
adjacency matrices of G_α and G_β are $A = [a_{ij}]$ and $B = [b_{ij}]$ respectively.

The current state of a node in the search tree is stored in a $p_\alpha \times p_\beta$ element matrix
$M = [m_{ij}]$. At the root of the search tree, the start matrix M^0 is constructed where:

$$
m_{ij}^0 = \begin{cases} 0 & \text{if we know } a\ priori \text{ that the } j\text{th point of } G_\beta \text{ could not correspond to} \\ & \text{the } i\text{th point of } G_\alpha \text{ in any subgraph isomorphism,} \\ 1 & \text{otherwise} \end{cases}
$$

Thus, the start matrix M^0 represents a *one-to-many* correspondence between the
points in V_α and the points in V_β. The matrices at the terminal nodes of the tree, M',
all represent distinct *subgraph isomorphism candidates* and contain exactly one number
1 in each row of the matrix. Thus, they represent distinct *one-to-one correspondences*
between the points in V_α and the points in V_β.

The tree search procedure is illustrated in Figure 6.4. At each level, a given row
of M is changed to only contain one 1. In other words, at each successive branch of
the tree, the correspondence between one point in V_α and another point in V_β is fixed
until, at the final level, all the correspondences have been fixed. Therefore, the search
tree has a depth d equal to p_α. A given subisomorphism whose matrix corresponds to
M' is said to be a *subisomorphism under M* if its terminal node is a successor of M, i.e.:

$$
\underset{\substack{1 \le i \le p_\alpha, \\ 1 \le j \le p_\beta}}{(\forall m'_{ij})} \quad m'_{ij} = 1 \quad \longrightarrow \quad m_{ij} = 1
$$

Finally, although the subgraph isomorphism candidates at the terminal nodes M' rep-
resent distinct combinations of correspondences between the points of V_α and V_β, a
final check needs to be carried out to reject correspondences which are not compatible
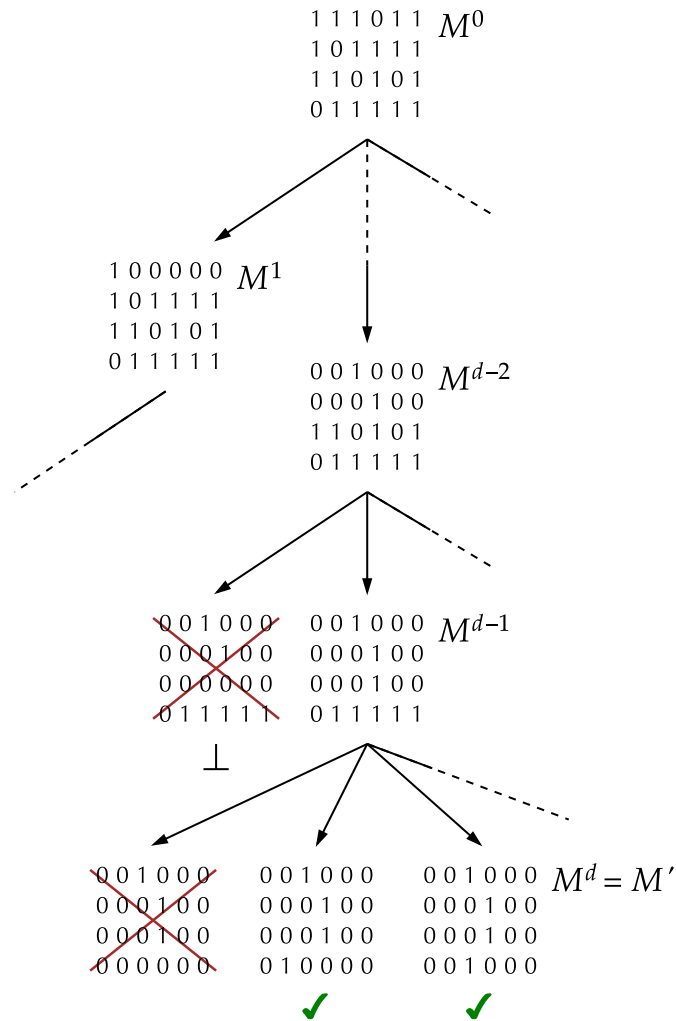with the structure of G_α and therefore do not represent valid subgraph isomorphisms.

$$
\begin{array}{ll}
\begin{matrix}
1\ 1\ 1\ 0\ 1\ 1 \\
1\ 0\ 1\ 1\ 1\ 1 \\
1\ 1\ 0\ 1\ 0\ 1 \\
0\ 1\ 1\ 1\ 1\ 1
\end{matrix} & M^0
\end{array}
$$

$$
\begin{array}{ll}
\begin{matrix}
1\ 0\ 0\ 0\ 0\ 0 \\
1\ 0\ 1\ 1\ 1\ 1 \\
1\ 1\ 0\ 1\ 0\ 1 \\
0\ 1\ 1\ 1\ 1\ 1
\end{matrix} & M^1
\end{array}
$$

$$
\begin{array}{ll}
\begin{matrix}
0\ 0\ 1\ 0\ 0\ 0 \\
0\ 0\ 0\ 1\ 0\ 0 \\
1\ 1\ 0\ 1\ 0\ 1 \\
0\ 1\ 1\ 1\ 1\ 1
\end{matrix} & M^{d-2}
\end{array}
$$

$$
\begin{array}{lll}
\begin{matrix}
0\ 0\ 1\ 0\ 0\ 0 \\
0\ 0\ 0\ 1\ 0\ 0 \\
0\ 0\ 0\ 0\ 0\ 0 \\
0\ 1\ 1\ 1\ 1\ 1
\end{matrix} &
\begin{matrix}
0\ 0\ 1\ 0\ 0\ 0 \\
0\ 0\ 0\ 1\ 0\ 0 \\
0\ 0\ 0\ 1\ 0\ 0 \\
0\ 1\ 1\ 1\ 1\ 1
\end{matrix} & M^{d-1}
\end{array}
$$

$$\bot$$

$$
\begin{array}{lll}
\begin{matrix}
0\ 0\ 1\ 0\ 0\ 0 \\
0\ 0\ 0\ 1\ 0\ 0 \\
0\ 0\ 0\ 1\ 0\ 0 \\
0\ 0\ 0\ 0\ 0\ 0
\end{matrix} &
\begin{matrix}
0\ 0\ 1\ 0\ 0\ 0 \\
0\ 0\ 0\ 1\ 0\ 0 \\
0\ 0\ 0\ 1\ 0\ 0 \\
0\ 1\ 0\ 0\ 0\ 0
\end{matrix} &
\begin{matrix}
0\ 0\ 1\ 0\ 0\ 0 \\
0\ 0\ 0\ 1\ 0\ 0 \\
0\ 0\ 0\ 1\ 0\ 0 \\
0\ 0\ 1\ 0\ 0\ 0
\end{matrix} \quad M^d = M'
\end{array}
$$

Figure 6.4: The tree-search procedure of the Ullmann algorithm for subgraph isomorphism

6.2.3.1 The refinement procedure

The refinement procedure as defined in [Ullmann 1976] presents a significant improvement to the basic tree-based enumeration, which discounts unfruitful search paths at an early stage of the process. It corresponds to a check that is carried out directly on the matrix M at each node in the search tree.

Let $v_{\alpha i}$ be the ith point of V_α and $v_{\beta j}$ be the jth point of V_β. Recall that, at a given node M in the search tree, for all values of i and j, $m_{ij} = 1$ means that the correspondence between $v_{\alpha i}$ and $v_{\beta j}$ is being considered for all subisomorphisms under M.

From the definition of subgraph isomorphism, for every adjacent point $v_{\alpha x}$ of $v_{\alpha i}$ in V_α, there must also be a point $v_{\beta y}$ in V_β that is adjacent to $v_{\beta j}$. If this does not hold, we can preclude the correspondence between $v_{\alpha i}$ and $v_{\beta j}$ for all subisomorphisms under M. In this case, m_{ij} is set to 0.

At the terminal nodes M', the above condition holds for all values of m_{ij}. This is a necessary and sufficient condition for subgraph isomorphism. Therefore, it is no longer necessary to perform an additional check that the terminal node represents a valid subgraph isomorphism.

6.2.3.2 Initial experiments

In our initial experiments, a few changes were made to the procedure to make it suitable for attributed relational graphs. As described at the beginning of this section, the *start matrix M^0* allows us to preclude correspondences between points in V_α and V_β where we have an *a priori* reason that they could not correspond to each other in any subgraph isomorphism between V_α and V_β. This provides us with an ideal opportunity to preclude correspondences between nodes which do not fulfil the *matching conditions* that have been set in the wrapper definition.

Whereas the starting matrix allows us to consider attributed *nodes* this way, it still assumes that all edges are equal. In order to preclude the matching of attributed edges that do not fulfil the matching rules, each attributed edge is also represented as a point in the graph. These "points from edges" are then joined by (unattributed directed) lines at the node-edge and edge-node interface, as shown in Figure 6.5.

Thus, for an example graph with m nodes and n attributed edges, we generate a graph G_α with $p_\alpha = m + n$ points and $q_\alpha = 2n$ (directed unattributed) lines. Similarly, for a document graph with p nodes and r attributed edges, we generate a graph G_β with $p_\beta = p + r$ points and $q_\beta = 2r$ lines.
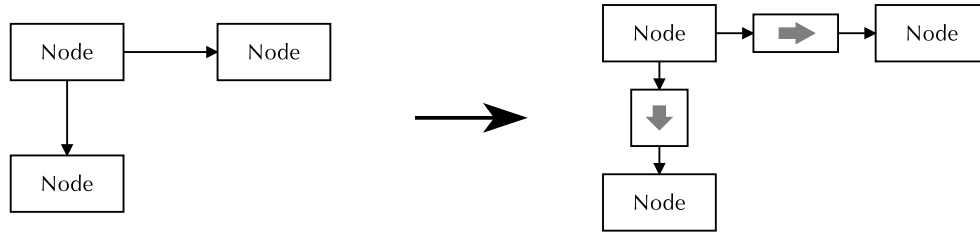
Figure 6.5: Example of edges being represented as nodes in the graph

To avoid confusion, we will use the terms **vertex** and **line** to refer to the points and lines of G_α and G_β; the terms **node** and **edge** will be reserved for the original graphs as shown to and manipulated by the user.

The starting matrix M^0 is therefore generated as follows:

$$m_{ij}^0 = \begin{cases} 1 & \text{if } v_{\alpha i} \text{ and } v_{\beta j} \text{ represent the same type of item (node or edge);} \\ & \text{in the case of } v_{\alpha i} \text{ and } v_{\beta j} \text{ being edges, they represent the same} \\ & \text{relationship; and if all additional matching conditions laid out} \\ & \text{in the wrapper specification are met} \\ 0 & \text{otherwise} \end{cases}$$

The algorithm for directed subgraph isomorphism in [Ullmann 1976] was implemented and initial tests were carried out on both the basic tree search algorithm and the algorithm with the refinement procedure. Whereas the basic algorithm became intractable when run on graphs with more than about four or five nodes, the refinement procedure was found to provide a dramatic reduction in time complexity, resulting in more than adequate performance for our application, even with much larger graphs. The performance results in Table 6.1, run on a document graph containing 20 nodes and 31 edges with example subgraphs of varying size, show the difference according to our experiments.

	without refinement step			with refinement step		
No. nodes/edges	2/1	3/2	4/3	2/1	3/2	4/3
No. iterations	6080	9 449 181	???	1743	2789	3061
Execution time (ms)	1102	324 925	???	90	120	170

Table 6.1: Effect of the refinement procedure on the performance of the Ullmann algorithm. With four nodes and three edges, the algorithm would have taken too long to complete without the refinement step

The only exception is when the example graph is not fully (weakly) connected. This can occur if the wrapper has not been specified properly. A node which is not

connected to any other node in the graph hinders the refinement procedure's ability to prune the search tree effectively. As the result is also meaningless, we define weak connectivity as a requirement for the wrapper definition.

6.3 Inexact matching

After implementing the algorithm in Section 6.2.2 and experimenting with real-life data sets, we realized that the required "error tolerance" could be achieved without resorting to an inexact matching algorithm. The two main features of our approach, *incomplete matching* and *multiple match edges*, are described in the following subsections.

6.3.1 Incomplete matching

In most situations where records have a variable structure, we found that it was possible to simply remove certain nodes from the example graph to match all the records in the document. Thus, we were essentially defining what must be present in the document for the wrapper to detect it as a record. Any additional nodes or edges that may have been present had no effect on the result.

We found the notion that a wrapper can be precisely defined preferable to using an inexact matching algorithm, which with its added complexity could generate unpredictable results with a significantly higher computational cost.

Using the process of *rectangular containment expansion*, as described in Section 4.3.2.2, we then add all *unmatched* nodes that fall within the bounding box of all the matched nodes to the result.

6.3.2 Multiple match edges

In situations where the simple removal of required nodes did not suffice, this was because additional lines of text or records were present either in the example instance or in the document. Thus, it was decided to increase the flexibility of our wrapper specification system to allow for **multiple match edges**. Whereas normal edges must have a one-to-one relationship between example and document in a wrapping result, multiple-match edges can match continuous runs of edges in the document in a particular direction, allowing tabular structures and listings with arbitrary numbers of rows to be wrapped with ease.

6.3.2.1 Multi-step matching algorithm

In the original prototype system as published in [Hassan 2009c], the wrapper graph is split into sub-graphs (**sub-examples**) along its multiple-match edges, which are then matched separately. These **sub-results** of the matching are then combined using a further matching step into the final, complete result.

When the wrapper graph is split along a given multiple-match edge into its sub-examples, this edge remains as part of *both* sub-examples, as shown in the example in Figure 6.6 (b).

This method for matching requires that each multiple match edge splits the wrapper graph completely into two sub-graphs. Thus, no edges apart from that particular multiple match edge may be present between the two sub-graphs. Geometrically, this restriction is easy to visualize and was not found to hinder the practical creation of wrappers.

Behind the scenes, the individual sub-results are found using the Ullmann algorithm with the refinement procedure for each sub-example. These sub-results are then combined in a further graph matching step. Two graphs are created, G_γ and G_δ, with V_γ and V_δ representing all sub-examples and all sub-results respectively. As the edges this time are not attributed (but they are still directed), they can be represented directly as lines in the adjacency matrix. Thus for adjacency matrices C and D of G_γ and G_δ respectively:

$$c_{ij} = \begin{cases} 1 & \text{if sub-examples } v_{\gamma i} \text{ and } v_{\gamma j} \text{ are connected by a multiple-match edge,} \\ 0 & \text{otherwise} \end{cases}$$

$$d_{ij} = \begin{cases} 1 & \text{if } v_{\delta i} \text{ and } v_{\delta j} \text{ represent sub-results of differing sub-examples and there} \\ & \text{exists a } \textit{maximal path} \text{ from } v_{\delta i} \text{ to } v_{\delta j}, \\ 0 & \text{otherwise} \end{cases}$$

We refer to edges in a sub-result, which have been matched to a multiple-match edge in the respective sub-example, as *matched multiple match edges*. A **path** exists between two sub-results $v_{\delta i}$ and $v_{\delta j}$ if, from a matched multiple match edge of $v_{\delta i}$, it is possible to continue along edges in the same direction (as long as they meet the conditions specified in the wrapper definition) to reach a matched multiple match edge of $v_{\delta j}$, *or* vice versa. This path is **maximal** if it is not possible to continue further to reach another matched multiple match edge.

The requirement of a maximal path allows unwanted overlapping results to be avoided. Consider the example in Figure 6.7, where the number of lines in a record may differ, but the record is always separated by whitespace. By setting a maximum

for the edge length, we can use multiple match edges to wrap such structures. Because the algorithm only looks for *maximal paths*, only the path between the top and bottom lines of the record will be found, extracting only the complete record. Because of the maximum edge length condition, the algorithm does not continue to the next record. In other words, the algorithm looks for the longest path that fulfils the conditions in the wrapper definition. Because the paths between intermediate lines are not maximal, these overlapping part-records, which we do not wish to extract, are avoided.

After execution of the matching procedure, the resulting matrices M' now represent compatible combinations of each of the sub-results, each of which can be joined together to produce an integrated result, as shown in Figure 6.6.

6.3.2.2 Performance issues of the multi-step algorithm

Initial experiments were carried out on the three wrappers described in Section 6.5. The largest weakness of the multi-step matching algorithm was not found to be the requirement that each multiple-match edge split the wrapper graph into two, but performance with certain wrappers using multi-match edges, particularly on complex documents.

Whereas the **travel** wrapper performed very quickly indeed, the other two wrappers were significantly slower. We attributed this to the fact that these documents contained a larger number of records per page on average. What surprised us, however, was that certain pages of the **travel** dataset, which contained *no* data to be extracted, took very long to process (typically over 20 s). On closer examination, these pages contained densely-packed classified information, much like in the **classifieds** dataset.
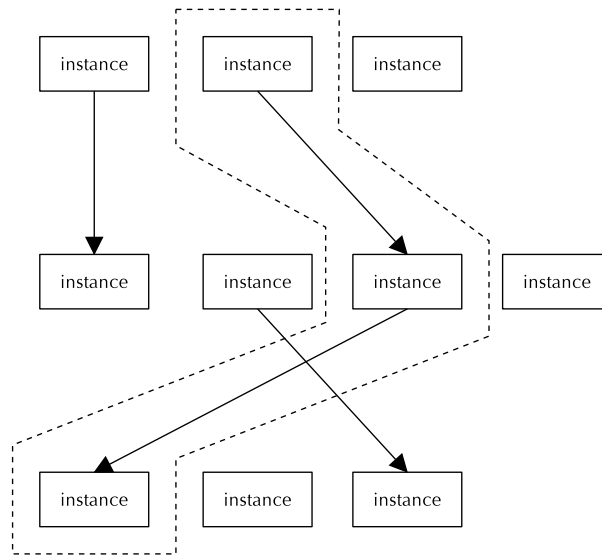
We found out that the reason for the slow processing time of these pages was the multi-step approach to graph matching. Although no results were returned at the end of the matching procedure, the wrapper was defined in such a way that each sub-graph that was matched against the page returned almost every object on the page as an intermediate result. All of these intermediate results were then compared against each other to find the final result, of which there was not a single matching pair.

This limitation led to the development of a new, *one-step* algorithm to detect wrapping instances using multiple match edges. By reworking the refinement procedure to also take multiple match edges into account, we not only prune unfruitful search paths even earlier in the matching process but, more importantly, we avoid the necessity of several graph matching operations with potentially hundreds of intermediate

a) Simple wrapper with multiple-match edges,
   which are marked with dotted lines

b) The wrapper is split into sub-graphs along the
   multiple-match edges and each of these sub-graphs
   is matched independently

c) A sub-instance association graph is formed, in which all pairs of
   instances are joined with an edge if it is possible to get from one
   to the other by extending a multiple-match edge. An (undirected)
   matching step is then used to extract the final solution

Figure 6.6: Integration of sub-results using graph matching in the multi-step algo-
rithm

results, leading to significant performance increases. This algorithm is described in the following subsection.

Algorithm 8 The one-step refinement procedure **onestepRefinement**

---

1. forall m_{ij} in M, $1 \leq i \leq p_\alpha$, $1 \leq j \leq p_\beta$:

   - if $m_{ij} = 1$

     (a) let v_α be the ith vertex of V_α and v_β be the jth vertex of V_β

     (b) evaluate the following condition:
        forall (v_γ, e_γ) where v_γ is a neighbour of v_α connected by edge e_γ:

        i. if e_γ is of type **singleMatch**:
           - there exists a vertex v_δ where:
             * v_δ is a neighbour of v_β connected by edge e_δ and
             * e_δ meets the matching criteria specified in e_γ and
             * a correspondence between v_γ and v_δ has not been precluded at this point in the search, i.e.:
               $m_{kl} = 1$ where k and l are the indices of v_γ in V_α and v_δ in V_β respectively

        ii. else if e_γ is of type **multipleMatchLast** or **multipleMatchFirst**:
           - the function **existsMultipleMatchPath**$(v_\alpha, v_\gamma, v_\beta, v_\delta, e_\gamma)$ evaluates to **true**

     (c) if the condition in step (b) evaluates to **false**, set $m_{ij} = 0$

2. repeat step 1 if any value of m_{ij} has been changed

6.3.2.3 One-step matching algorithm

With the one-step matching algorithm, all checks against matching conditions involving single and multiple match edges are incorporated into the refinement procedure itself. With this method, is it no longer necessary to represent attributed edges as special types of points in the graph, and the requirement that each multiple-match edge splits the wrapper graph into two also no longer applies.

The starting matrix M^0 is formed according to the matching criteria as with the multi-step algorithm in Section 6.2.3.2. The only difference is that only the nodes are included in the matrix, as edges are no longer represented as points. The refinement procedure is given in Algorithm 8 and uses the function **existsMultipleMatchPath** as defined in Algorithm 9. As before, the one-step refinement procedure is a necessary

---

Algorithm 9 The function **existsMultipleMatchPath**

---

pre: v_α and v_δ are joined together in the wrapper graph G_α by edge e_γ; v_β and v_δ are the corresponding nodes of v_α and v_δ in the document graph G_β respectively

1. return **true** if a path exists between v_β and v_δ that satisfies the following conditions:

   (a) all edges in the path are in the same direction as and fulfil the matching conditions defined in e_γ;

   (b) if e_γ is of type **multipleMatchLast**:

      i. the path from v_β to v_δ consists of *one or more* nodes fulfilling the matching criteria in v_β, followed by *one or more* nodes fulfilling the matching criteria in v_δ

      ii. the path is *maximal*, i.e. cannot be extended backward or forwards, i.e.:

         • there exists no edge pair (v_x, e_x), where e_x points from v_x to v_β, v_x fulfils the matching criteria in v_β and e_x fulfils the matching criteria in e_γ; and

         • there exists no edge pair (v_y, e_y), where e_y points from v_δ to v_y, v_y fulfils the matching criteria in v_δ and e_y fulfils the matching criteria in e_γ

   (c) else if e_γ is of type **multipleMatchFirst**:

      i. the path is *minimal*, i.e. there is no intermediate node between v_β and v_δ that satisfies the matching conditions of v_β and/or v_δ.

---

and sufficient condition for subgraph isomorphism, and checks all nodes, edges and multiple match edges against their respective matching conditions. Therefore, the resulting matrices M' correspond to distinct wrapping results, and no further checks are required.

6.3.2.4 Two types of multiple match edge

To extend the number of possibilities for wrapper generation, it was decided to add a further type of multiple mach edge, **multipleMatchFirst**. When this type of edge is used between two given nodes v_α and v_γ, both nodes should have matching conditions set. When these nodes are matched to v_β and v_δ respectively, the matching algorithm will look for and match the last node before v_β and the first node after v_δ that match the respective conditions.

This allows, for example, the first and last lines of a record to be matched using conditions for both nodes. As long as these conditions do not match any intermediate lines in the record, this type of edge will match the first line, the last line and all rows in between.

The function **existsMultipleMatchPath**, as given in Algorithm 9, precisely defines both types of multiple-match edge.

6.4 Wrapper creation

In our prototype system *GraphWrap*, wrappers can be created or edited in two ways. The easiest most common way is to use the user interface, as described in Section 6.4.1. As wrappers are stored in XML format, it may sometimes be quicker to edit the files directly. Also, some advanced features are only available by editing the files directly.

6.4.1 Interactive wrapping using the user interface

The GUI of *PDF Analyser*, as described in Section 3.2, has been extended to support a **graph view** of the document. Whereas the graph structure can also be overlaid on the *page view*, the flexible layout of the graph view enables the user to zoom in and out and manipulate the wrapper interactively. The current wrapper can be executed at any time, and the visual result will be shown immediately in the page view.

The graph view has been built on top of the *TouchGraph* library [TouchGraph LLC 2006 (Web)]. As the screenshot in Figure 6.8 shows, green edges represent adjacency edges pointing eastwards, and blue edges point southwards, irrespective of their physical direction. Nodes which belong to the document but are not part of the current wrapper graph are shown in a faded colour. These can then be added to the wrapper definition, if desired. The bottom part of the window allows the matching conditions of the current node or edge to be viewed and modified.

The following steps summarize the interactive creation process of a wrapper:

1. The appropriate *granularity* is selected, i.e. **lines** or **blocks**, and the desired document is opened

2. On the page view, the user selects one example record by drawing a marquee box around it. Its corresponding subgraph is found automatically and displayed in the graph view. This subgraph already fits the definition of a *valid wrapper*

Figure 6.7: Example of the significance of the *maximal path* between multiple match edges. The example on the left shows the record which is being wrapped. The wrapper is specified using a multiple-match edge between the **NAME** and **E-MAIL** nodes. If all paths between the multiple match edges are considered, this leads to several overlapping incomplete records (centre). Considering only the maximal path delivers the result that we are looking for (right)
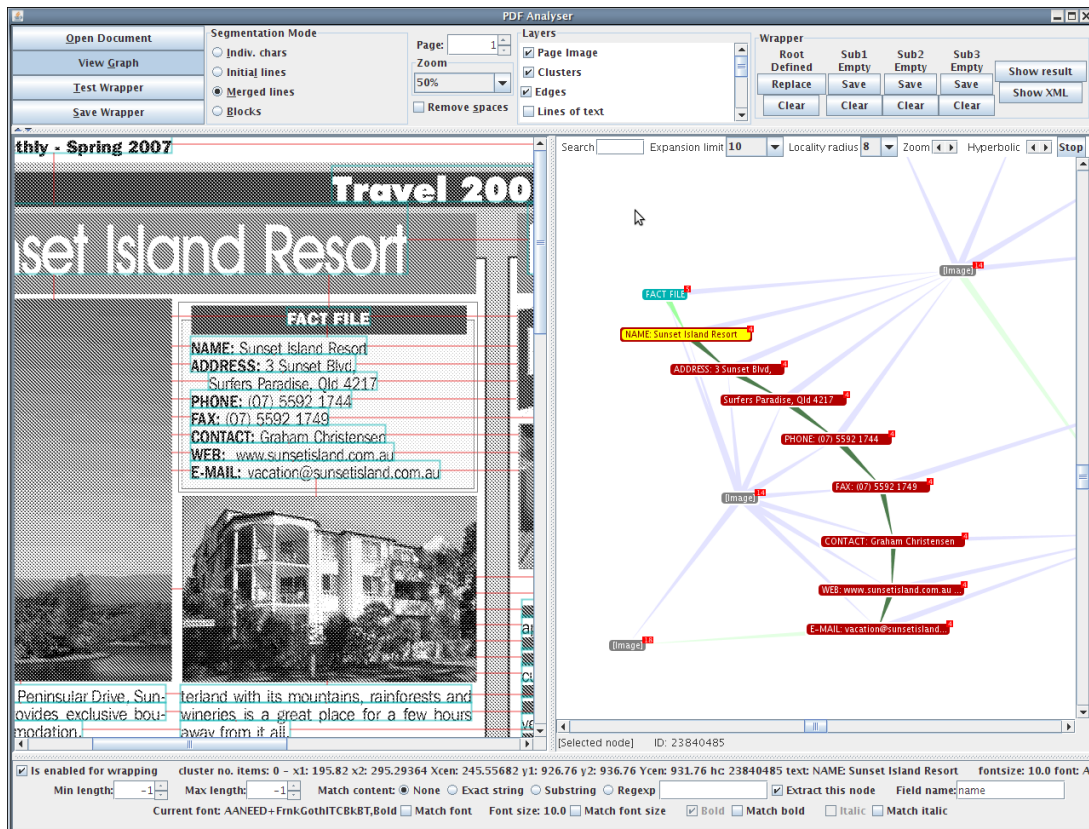

Figure 6.8: Example of the GUI in action. The left pane shows the *page view*; the right pane shows the *graph view*, allowing the wrapper to be edited and additional wrapping constraints to be added. When the wrapper is executed, the found records are shaded on the page view

Coastal Point

Classifieds

EMPLOYMENT | **EMPLOYMENT**

NOW HIRING:
Front Desk/Maint.
Experience preferred
Housekeepers
Needed thru Oct 19th
Holiday Inn Express
39642 Jefferson Bridge Rd.~ Bethany Beach, DE
Apply in person

Experienced
Wait Staff
Year Round
Full-Time & Part Time
Open from Monday - Friday, 1 - 3:
302.537.4101
THE QUAIL Restaurant

RUTH'S CHRIS STEAK HOUSE
Are you ready for a change?
• **Server Assistants**
• **Hosts/Hostesses**

ANTIQUES & COLLECTIBLES

RECORDS: 45s FROM 60s, 70s, 80s, 90s. 33 1/3 - Elvis, Michael Jackson, Motown and more. Best offer: (302)539.0224

DOLLS - INCLUDING (10) Barbie dolls; doll clothing. Can email pictures. PIck up West Fenwick or Ocean City. (302)436.5679

(2) LIMITED EDITION Porcelain Cabbage Patch dolls signed by Xavier Roberts in original boxes. Retail: $350; asking: $250

APPLIANCES

AIR CONDITIONER, window unit. 5,000 BTU 115V by Goldstar Ice. Cold air. $40 (302)436.2197

DRYER, GOOD condition: $75. (302)381.2217

AIR CONDITIONER: 5,000 BTU. Frigidaire window unit. Works well. $40 Ocean View (302)829.8399

SEARS KENMORE ELITE microwave. 1,200 watts, 2.0cf capacity, black. Like new.

COMPUTER ASSISTANCE

COMPUTER
Repair, Setup, & Networking Services
Affordable Prices
WeAreSynthetic.com • 302.682.0573

CLOTHING

LONG BLACK LEATHER coat with genuine blue fox collar. Size 2-4. Beautiful condition. Asking $75. (302)344.3414

ELECTRONICS

NEW DESKTOP COMPUTER Dual-core CPU 2.7GHz, 4GB memory, sound and video cards. DVD burner with

---

KF-Spannring, Alu-Druckguss
Clamping ring, die-cast aluminium

| Nennweite Nominal width | Abmessungen Dimensions | | | | Artikel-Nr. Order No. | Preis Price |
	A (mm)	B (mm)	C (mm)	D (mm)		(€)
DN 10/16	42	61	16	22	100 1016	7,00
DN 20/25	54	72	16	32	100 2025	8,00
DN 32/40	70	90	16	46	100 3240	10,00
DN 50	95	123	25	62	100 0050	21,50

Andere Abmessungen und Ausführungen auf Anfrage. Other dimensions and versions on request.   5

Travel 2007-2008   LOCAL MEDIA ADVERTISING FEATURE

Sunset Island Resort

FACT FILE
NAME: Sunset Island Resort
ADDRESS: 3 Sunset Blvd, Surfers Paradise, Qld 4217
PHONE: (07) 5592 1744
FAX: (07) 5592 1749
CONTACT: Graham Christensen
WEB: www.sunsetisland.com.au
E-MAIL: vacation@sunsetisland.com.au

Nestled at the end of Peninsular Drive, Sunset Island Resort provides exclusive boutique holiday accommodation.

Centrally located on a quiet street, you will find Sunset Island Resort is just a short walk to Cavill Mall, and on to the renowned and pristine beaches of Surfers Paradise, Gold Coast, in the heart of Queensland's celebrated and sought after holiday destiterland with its mountains, rainforests and wineries is a great place for a few hours away from it all.

You will love every minute of your stay with us here at Sunset Island Resort – your ideal home away from home in Surfers Paradise, tucked away from the bustling streets but yet so close to the action…the ideal holiday getaway – look no further…Sunset

Emerald Sands Holiday Apartments

Emerald Sands APARTMENTS

FACT FILE
NAME: Emerald Sands Holiday Apartments
ADDRESS: PO Box 1712, Surfers Paradise, Qld 4217
PHONE: (07) 5526 7588
FAX: (07) 5538 6522
CONTACT: Sherin Marriott
WEB: www.emerald-sands.com
E-MAIL: john.marriott@bigpond.com

Discover the jewel of the Gold Coast Beachside
■ 18 luxury fully self-contained boutique apartments
■ 50 metres to the sand, no roads to cross
■ Sparkling heated pool, spa and barbecue in walled garden setting
■ Magical ocean views from all apartments
■ 200 metres to restaurants, 24-hour convenience store
■ 20 minutes walk to central Surfers, or Broadbeach shopping, dining
■ Secure underground parking, security lift to all floors
■ 4-star rating
A holiday you'll treasure. At a price you can afford

Figure 6.9: Example records from the three datasets

3. If the user were to execute the wrapper at this stage, it would likely return hundreds of overlapping results. Therefore, it is necessary to *set conditions* to one or more of the nodes. By clicking on a node or edge in the graph view, the available conditions will be displayed on the bottom of the screen, and can be edited. Nodes can also be added to or removed from the wrapper if necessary

4. By clicking the button **Test wrapper**, the current wrapper can be executed and the results shown to the user. It is also possible to try out the wrapper on a different page or different document using the GUI

5. When the user is satisfied with the wrapper, it can be exported by clicking **Save wrapper**. The resulting wrapper can then be used with the command-line tool `graphwrap` to automatically extract data from other PDF documents and output it in XML format

The following conditions are available for nodes and edges:

Node conditions	Edge conditions
Text matches string, substring or regexp	Minimum and maximum length
Minimum and maximum text length	Alignment left, right, centre
Font size, font name, bold, italic	Crosses ruling line (yes, no, any)
Ruling line above, below, left, right	Multiple match (single, first, last)

6.4.2 Hierarchical wrapping

In order to wrap nested structures, such as individual fields within records, wrappers can be *nested* inside each other. There are three modes of nesting:

- **subgraph:** here, only the nodes that have been *explicitly* matched by the parent wrapper will be made available to the child wrapper;

- **area-based:** in this mode, *rectangular containment expansion* (see Section 4.3.2.2) is carried out on each result of the parent wrapper to add nodes that have not been explicitly matched, but fall within the bounding box of the result. This mode is useful if multiple match edges have been used in the definition of the parent wrapper;

- **whole page:** in this mode, the entire page of the document is made available to the child wrapper, but only results of the child wrapper that intersect the area of the parent wrapper are output. This allows tabular structures to be wrapped. See Section 6.5.2 for an example application of this technique.

6.5 Experiments

In order to provide an indication of the effectiveness of our wrapping approach, we created three wrappers on three different data sets, which represent possible use-case scenarios of our system. Figure 6.9 gives an example of each of the three types of document. These wrappers are described in the subsections below. Numerical results are given in Table 6.2.

6.5.1 *Coastal Point* Classifieds

The **classifieds** wrapper was designed to extract the textual content of all normal (unboxed) classified advertisements from the *Coastal Point* newspaper[1] and was run on eight issues. An example of the content is shown in Figure 6.9 (top). In order to allow for varying amounts of text in a classified advertisement, an edge of type **multipleMatchLast** was employed in the design of the wrapper, with the maximum length set to the distance between two lines.

After executing the first version of the wrapper, it was found that the various issues of the *Coastal Point* newspaper across our dataset employed slightly different layout conventions. For example, whereas in the example in Figure 6.9 (top) whitespace is present between each advertisement, this was not always the case. Also, certain issues used bold text for the telephone number and contact information.

The initial version of the wrapper also delivered a number of false positives on the page. The following changes were made to reduce this number:

- the top node of an advertisement, containing the title, was always in bold text and in capitals. However, certain items offered for sale did not begin with text, but with a number. Therefore, the following conditions were set:

  - **match bold: true**
  - **match text: regexp**
  - **match text string:** the regular expression [A-Z]{4,}, i.e.: the string must contain at least four capital letters in sequence

- the multiple-match edge joining both nodes was given the additional condition:

  - **crosses ruling line: yes**

---

[1]*Coastal Point* classifieds,
`http://www.thecoastalpoint.com/_files/classifieds/classifieds.pdf`

We also experimented with setting **isAligned** to **left**. This reduced the number of false positives even further. However, the few results that were not left aligned were not returned, so this condition was abandoned.

Ideally, a *maximum width* condition for the nodes would have been available, which would have prevented practically all false positives which occurred in this situation.

6.5.2 Pink GmbH component catalogue

The **catalogue** wrapper was designed to test our hierarchical approach to wrapping tables, and was run on a 72-page catalogue of technical components[2]. An example of one of the tables is shown in Figure 6.9 (centre). Here, the wrapper was designed to extract all items that have a nominal width, order number, price and measurement A; as well as measurements B, C and D if these were given.

The top-level wrapper was created to extract each individual row. The font and size were fixed and the leftmost node was set to begin with the string **DN** (the first two letters of the order number). A multiple match node was used to extract the remainder of the column.

At the second level, several wrappers were created, one for each field. Again, a string condition was used to limit the wrapper to the desired field. The wrappers were then combined using the *whole page* mode.

Of the three datasets that were used, the catalogue had the most consistent formatting throughout. Unsurprisingly, this led to an almost perfect wrapping result with no false positives. However, a few inconsistencies were found, for example one column was headed **C(mm)** instead of **C (mm)** (i.e. the space was missing). One column was also set in a different font size. After all these inaccuracies were accounted for in the wrapper specification, it was possible to extract all the data items correctly with the wrapper.

6.5.3 *Travel Monthly* archives

Finally, we decided to test our system on a much larger data set, a collection of back issues from the *Travel Monthly* magazine[3], which contains 190 PDF files with 445 pages in total. Each destination contains a "fact file" box with contact information, which we extracted with the wrapper `travel`. An example of two such records on a page

---

[2]Pink GmbH Vakuumtechnik component catalogue, `http://www.pink.de/pdf/katalog.pdf`
[3]*Travel Monthly* archives, `http://www.travelmonthly.com.au`

Wrapper	classifieds	catalogue	travel
No. docs	8	1	190
No. pages	24	72	445
No. data items	1095	4835	1385
Precision	98.16%	100%	99.78%
Recall	98.16%	100%	97.11%
Processing time/page	7.86 s	8.48 s	4.3 s
Match time/page	3.34 s	6.13 s	0.68 s

Table 6.2: Experimental results of the graph-based approach on three datasets

is given in Figure 6.9 (bottom). The condition for the top node was set to include the substring **FACT FILE** or **Fact File**. In order to allow for varying amounts of text in a classified advertisement, an edge of type **multipleMatchLast** with a maximum length condition was also used here.

After fine-tuning the conditions, this wrapper was also able correctly extract almost all records, apart from a small minority which had a vastly different layout (e.g. double line spacing). Due to the larger size of the dataset (several years' worth of issues), a variety of formatting conventions were encountered. Some records included a graphic (e.g. logo) between the heading **FACT FILE** and the data. This graphic was represented in the graph as an image node. As image nodes are not yet supported by the system, these records could also not be retrieved.

6.6 Discussion

This chapter has presented a novel approach to user-guided wrapping from print-oriented documents using graph matching techniques. Through experiments we have shown that this method can be used in a variety of wrapping scenarios, and perform accurately and quickly. Although the task of graph matching is often associated with high computational complexity, we have shown this not to be a problem for our application in practice. The *one-step* algorithm presented in this chapter keeps search and time complexity to a minimum.

In a similar fashion to the *Lixto Visual Developer* for wrapping HTML files, we have found that robust wrapper generation required adding a large number of matching conditions, to ensure that false positives are avoided. A major limitation of this prototype system is the limited number of matching conditions that can be set for a given node or edge. The next step would be to develop the user interface to allow for any combination of conditions on any property to be set, as is already possible with Lixto.

We also found the notion of *multiple match edges* not as simple to understand as we first thought it would be. Although the addition of a third category, **multipleMatch-Last**, has increased the number of possibilities for wrapper generation, it has also complicated matters further. The categories **first** and **last** determine the conditions that apply to extra nodes that are added between the matched multiple-match edges. It would probably be much simpler for the wrapper designer to set these conditions separately, rather than have them inherited from the neighbouring nodes.

The current implementation of multiple match edges makes it easy to define wrappers where two or more repeating rows occur. However, these wrappers will not work in cases where only one (or no) row is present. Extending the edge attributes to allow for *optional* edges would be one way to allow for such situations. Alternatively, a number of different graphs or *patterns* could be used, in the same way as in the Lixto VD, which could even be generated automatically by the user interface, to allow for each situation (0 rows, 1 row, 2 or more repeating rows).

A further improvement to the system would be a function to *automatically create multiple match edges*. After the user has selected the respective *from* and *to* nodes in the example instance, this function would automatically create a multiple match edge of the appropriate type with the relevant matching conditions, ensuring that it matches all edges and nodes between the two selected nodes.

Finally, the flexible *graph view* based on the *TouchGraph* library was not always very easy to work with. An improved layout algorithm which keeps the graph upright whilst allowing the layout to remain flexible, so that most edges point more or less in their real direction, would improve usability. Also, a direct superimposition of the flexible graph structure on the page view or the ability to click on either view to select nodes would improve the user-friendliness of the system considerably.

In this section, we have summarized the main limitations of the prototype system as applied to the methods proposed in this chapter. In the next section, we propose future research directions in graph-based wrapping.

6.7 Future directions in graph-based wrapping

In order to improve the user-friendliness, power and robustness of the graph-based approach, the following future research directions are proposed:

- **Logical relations between nodes:** in the present system, the edges only represent *adjacency* relationships in a given direction. It would be interesting to experiment with the use of logical relations as discovered in the document understanding process, such as *superior-to-inferior* or *reading order*, to see if they

could simplify the wrapping process or make new wrapping applications possible:

- *spatio-logical* **relations**: the use of relations which express concepts that refer both to the logical and spatial structures, for example: "neighbour above, but within same text column", could be particularly useful for creation of wrappers;

- **wrapping across multiple pages:** the current system does not allow for situations where a record spans from one page to the next. By using a logical edge of e.g. type *reading order* to join content from one page to the next, such wrapping situations could also be enabled.

- **Error tolerance:** although the current system allows for a certain amount of flexibility via *incomplete matching* and *multiple-match edges*, we found from our experiments that even professionally typeset documents can contain minor layout or formatting errors (or purposeful changes due to space reasons). These include changes in font, font size, alignment, line spacing and adjacency. In the present system, it is often difficult and time-consuming to define the matching conditions in such a way that all records are found by the algorithm. Therefore, in order to improve the flexibility of the system, the following investigations are proposed:

  - **use of an error-tolerant matching algorithm:** in the present system, the use of a real error-tolerant matching algorithm was avoided for reasons of complexity. However, as the search procedure can be tightly controlled by the refinement step of the Ullmann algorithm (see Section 6.2.3.1), it may be possible to allow for small changes in the graph structure without increasing the search complexity to an unacceptable level. This would improve robustness in situations where slight changes to the graph structure occur;

  - **"fuzzy" evaluation of matching conditions:** many formatting errors affect only the attributes of the respective nodes, and not the graph structure itself. The use of a scoring-based or similar system to evaluate the matching conditions could lead to significant improvements in robustness, even if an error-tolerant matching algorithm is not used. The amount of "fuzziness" could be interactively set by the user at a global or local level.

- **Measures to automatically generate wrappers via** *wrapper induction*: although the use of a graph-based structure has made the creation process easier and more intuitive, the creation process is not yet as straightforward for the user as

114

it could be. It would be worthwhile to investigate the possibility of automatic *wrapper induction* methods based on two or more example records selected by the user.

- **Structural changes to remove discontinuities:** currently, the document analysis (segmentation and graph generation) and wrapping stages of the process are completely separate from each other. This means that any decisions made in the wrapping stage cannot override or correct any errors made in the segmentation stage. By using a *hierarchical* or *pyramid* graph structure, several granular levels could be represented simultaneously. By using a *Voronoi graph*, adjacency relations in the four directions could also be deduced in a more robust way. The *discontinuity problem* is discussed in more detail in Section 7.2.

- **Wrapping from other formats:** our model of a PDF document, which essentially represents the page as a set of rectangular objects (see Section 3.1), could also be used with other formats, most notably PostScript, but also HTML [Krüpl et al. 2005]. Even though the idea of graph-based wrapping was partly inspired by existing HTML wrapping approaches such as Lixto, it would be interesting to investigate whether this approach could also be beneficial to HTML wrapping.

Chapter 7

Conclusions and further work

This dissertation has investigated two approaches to user-guided data extraction from PDF documents, the *conversion approach* and the *graph-based approach*, both of which were developed to a prototype stage and evaluated experimentally. Whereas the conversion approach was based on a novel application of largely existing methods and principles, the graph-based approach involved the creation of new methods and the application of graph matching algorithms to a domain in which they had not, to our knowledge, been used before.

The conversion approach was shown to be suitable for documents of moderate complexity containing regular tabular layouts. An example case study on a sample document [Statistik Austria 2009 (Web)] is provided in Section 5.1. Experiments carried out on the graph-based approach show its ability to extract data from a wider range of documents with high precision and recall rates. A summary of the main advantages and disadvantages of both approaches is given later on in this chapter (Section 7.1).

In the course of investigating both approaches, several problems in related fields, in particular *document analysis* and *document understanding*, were also addressed. This thesis has therefore also made the following additional contributions:

- in Section 3.1 a model for storing print-oriented data for document analysis purposes was defined, and in Section 3.3 methods were proposed to populate this model with PDF data, simplifying the data where necessary;

- in Section 3.4 the *ordered-edge segmentation algorithm*, an efficient page segmentation algorithm for data held in this model, was proposed;

- in Section 3.5 we presented the results of experiments on the above two stages of the process, which demonstrate the suitability of these methods for complex page layouts;

- in Section 4.3 an algorithm for automatic table recognition and structure recognition was presented;

- in Section 4.4 the above algorithm was compared experimentally to another algorithm in the literature, which was published two years later. Both algorithms were found to give similar results and be able to detect tables in a wide variety of documents. The problem of evaluating table recognition results was also presented. Appendix A lists common errors that are encountered in table recognition and proposes how they be consistently evaluated to produce precision and recall figures comparable among different systems.

The various sub-problems have been discussed in detail in their respective chapters, and future research directions have been proposed. The remainder of this chapter provides a comparison of both approaches to wrapping and proposes future work to address an important issue which affects not only data extraction from documents, but document analysis and processing in general.

7.1 Comparison of the conversion and graph-based wrapping approaches

In order to wrap data using the conversion approach, this data needs to be located in tables, headings or other structures which are recognized by the PDF-to-HTML conversion methods in Chapters 3–4. If this data is understood correctly by the conversion method, wrapping can be performed on the HTML conversion. This means that the conversion approach is inherently limited in the number of wrapping applications in which it can be employed.

As described in Section 5.3, the graph-based approach was found to address most the main weaknesses of the conversion-based approach and enable a much wider range of wrapping programs to be created. Specifically:

- **the wrapping is performed directly on the document's visual structure:** the conversion to an intermediate format, such as HTML, which does not necessarily represent the physical layout of the document, is unintuitive;

- **the graph-based approach is more expressive**[1] by allowing three types of structure to be used: the physical structure, the detected logical structure and the attributes of the blocks. In the conversion approach, *only* the detected logical structure can be used;

- by not necessarily requiring that the entire page be understood correctly, **the graph-based approach is more robust,** as it can essentially ignore any errors in segmentation, graph generation or structure detection outside of the region of interest. In the conversion approach, such errors could affect the top-level structure of the document, which would very likely prevent the wrapper from functioning correctly.

7.2   Further work: Addressing the *discontinuity problem*

The biggest weakness that we see in the present system, which is common to many other approaches based on document analysis and understanding, is what we term the **discontinuity problem:** each step in the document analysis, understanding and data extraction process is seen as a fully independent stage, which assumes correct input and has no opportunity to correct any errors made in the previous stages. Even the segmentation process is performed using knowledge specified on two different granular levels, which do not always interact with each other. Unfortunately, document analysis is inherently an inaccurate process and errors on one level propagate to higher levels, which presents a real limitation to the possible robustness that such a system can achieve.

In the present system, we have tried to address this problem in the following stages:

- **segmentation:** the *ordered-edge segmentation algorithm* (see Section 3.4.3) orders the neighbourhood relationships in such a way that the most obvious neighbourhood relationships are examined first, using only local knowledge. The more difficult decisions are made towards the end of the process, when more information about the complete structure, i.e. higher-level knowledge, is also available;

---

[1]By representing the detected tables (or other structures) as logical relations and by creating a wrapper using multiple match edges, the graph-based approach can "simulate" the overwhelming majority of use-cases of the conversion approach. However, in many of these cases a simpler, more robust wrapper could be generated using just the physical structure of the document. For example, it was possible to extract all data cells of the *Statistik Austria* document in Section 5.1 without error using this approach.

- **table structure recognition:** recall that the table search algorithm builds candidate tables by clustering together candidate columns. The *validation procedure* (see Section 4.3.3), which is executed at each iteration of the table search algorithm, examines the table on several granular levels to ensure that it is valid.

However, neither of these procedures is able to *correct* or *reverse* decisions made in previous stages of the process. Whereas further improvements to the individual algorithms could lead to slight improvements, we believe that we are now reaching the point of diminishing returns, and the next stage of development is to address the *discontinuity problem* in a methodical and systematic way.

We have already touched on this problem in various sections of this thesis. In Section 3.7, we suggested that an improved model of knowledge on several levels of granularity would lead to further improvements. In Section 4.4.6, where we discussed the table detection algorithm, we suggested that a multi-granular approach along the lines of [Wang 2002] could deliver significant improvements in robustness and accuracy. In Section 6.7, where we discussed the graph-based approach, we proposed replacing the flat graph representation with a hierarchical or pyramid-like structure to represent the page content on multiple granular levels. The edges between neighbouring nodes could also have fuzzy values.

Chapter 2 introduced the field of document analysis and the somewhat imprecise concept of *layout conventions*, which are used by the document author to communicate the logical structure of the document to the user. Recall that there are two types of layout conventions, those *specific* to a particular document class or publication and those *generic* to a broad range of documents. It is the firm belief of the author that, whereas document specific conventions are of a firmly **syntactical** nature, document generic conventions can be split up into syntactic and **perceptual** rules. Examples of such rules are as follows:

- **document specific:** "Sub-headings are set in Garamond Semi-Bold, 14 pt., left aligned." "Body text is set in Garamond Regular, 10 pt., justified."

- **document generic:**

  - **syntactical:** "Sub-headings occur above text and are usually set in more salient (e.g. larger or bolder) text. However, the text is not as salient as that of headings."
  - **perceptual:** "A set of objects with the same alignment, in close proximity to each other and equally spaced apart, belong to one higher-level element."

Whereas syntactical rules can be thought of as referring to the knowledge *acquired* by humans, perceptual rules refer to the human visual system and the associated *hereditary* knowledge. We believe that this hereditary aspect has not been given the attention it deserves in the literature, even in systems designed for a broad class of documents. Compared to natural scene images, real-world documents have a much simpler visual structure. But simplicity deceives. The majority of approaches in the literature are based on *ad hoc* methods on a given level of granularity, which suffice for the interpretation of syntactical conventions, but not for perceptual conventions. Section 2.2.6 introduced some approaches to document analysis borrowed from the computer vision field, which do not limit themselves to using just syntactical knowledge about a document. However, these approaches did not rediscover such a wide range of structures from the document.

To continue this work, we propose the creation of a **document model**, in which the most important layout conventions can be represented at multiple levels of granularity concurrently. Perceptual conventions can be modelled by the most important *stimuli* that are present in document layouts, such as alignment, separation boundaries (ruling lines and whitespace rivers), proximity and similarity. Using a probabilistic framework such as that described in [Wang 2002], syntactic conventions could also be encoded in a fuzzy way, so that that classifications of a given block *compete* against each other. For example, columns in an unruled table and columns of text share many properties (see Section 3.7), and the final classification may depend on higher-level knowledge, e.g. whether a table has been detected or whether the column structure fits in with the text columns on the page.

Once an accurate, probabilistic document model has been created, the next task is to search the problem space in order to find the global maximum. In a similar way to Wang's approach, this could be achieved by using the result of existing algorithms (such as those presented in this thesis) as a starting point, followed by iterative improvements in a series of edit operations, until the (hopefully global) maximum is achieved. An alternative approach would be a search based on an exhaustive enumeration of all possibilities. However, unless the search tree is extensively pruned to discount unfruitful search paths, such an approach is likely to be intractable.

We believe that such developments could overcome the most significant limitation of today's document processing systems and redefine the state of the art in document analysis and structure recognition in PDF documents. The resulting data could then be represented in an appropriate structure, such as a hierarchical graph, in which any remaining uncertainties could also be represented by edges with fuzzy values. Using a graph-based wrapping method based on the approach described in Chapter 6, it would be possible to wrap data directly on this structure with unparalleled robustness and ease of use.

Bibliography

A

ABBYY. Advanced PDF to HTM Converter 1.9. `http://pdftransformer.abbyy.com/`, 2006 (Web).

Adobe Systems Inc. *PDF Reference and Adobe Extensions to the PDF Specification*, June 2009 (Web).

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974. ISBN 0201000296.

Marco Aiello, Christof Monz, Leon Todoran, and Marcel Worring. Document understanding for a broad class of documents. *International Journal on Document Analysis and Recognition*, 5(1):1–16, 2002.

Oronzo Altamura, Floriana Esposito, and Donato Malerba. Transforming paper documents into XML format with WISDOM++. *International Journal of Document Analysis and Recognition*, 4(1):2–17, 8 2001.

Anjo Anjewierden. AIDAS: incremental logical structure discovery in PDF documents. In *ICDAR 2001: Proceedings of the 6th International Conference on Document Analysis and Recognition*, pages 374–378, 2001.

Apostolos Antonacopoulos, Stefan Pletschacher, David Bridson, and Christos Papadopoulos. ICDAR 2009 page segmentation competition. In *ICDAR 2009: Proceedings of the 10th International Conference on Document Analysis and Recognition*, pages 1370–1374, 2009.

Arvind Arasu and Hector Garcia-Molina. Extracting structured data from web pages. In *SIGMOD 2003: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 337–348, 2003.

Archilogue. Advanced PDF to HTM Converter 1.9. `http://www.archisoftint.com/logiciels/recr_us.htm`, 2006 (Web).

B

Steven R. Bagley. COG extractor. In *DocEng 2006: Proceedings of the 2006 ACM Symposium on Document Engineering*, pages 31–31, 2006.

Steven R. Bagley, David F. Brailsford, and Matthew R. B. Hardy. Creating reusable well-structured PDF as a sequence of component object graphic (COG) elements. In *DocEng 2003: Proceedings of the 2003 ACM Symposium on Document Engineering*, pages 58–67, 2003.

Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual web information extraction with Lixto. In *VLDB 2001: Proceedings of the 27th International Conference on Very Large Databases*, pages 119–128, 2001.

Robert Baumgartner, Wolfgang Gatterbauer, and Georg Gottlob. Web data extraction system. In *Encyclopedia of Database Systems*, pages 3465–3471. 2009.

Serge Belongie and Jitendra Malik. Matching with shape contexts. In *CBAIVL 2000: Proceedings of the IEEE Workshop on Content-based Access of Image and Video Libraries*, pages 20–26, 2000.

Jean-Luc Bloechle, Denis Lalanne, and Rolf Ingold. OCD: An optimized and canonical document format. In *ICDAR 2009: Proceedings of the 10th International Conference on Document Analysis and Recognition*, pages 236–240, 2009.

C

Daniel Cabeza and Manuel Hermenegildo. Distributed WWW programming using (Ciao)-Prolog and the PiLLoW library. *Theory and Practice of Logic Programming*, 1 (3):251–282, 2001.

Francesca Cesarini, Marco Gori, Simone Marinai, and Giovanni Soda. INFORMys: A flexible invoice-like form-reader system. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(7):730–745, 1998.

Francesca Cesarini, Simone Marinai, L. Sarti, and Giovanni Soda. Trainable table location in document images. In *ICPR 2002: Proceedings of the 16th International Conference on Pattern Recognition*, volume 3, pages 236–240, 2002.

Chia-Hui Chang, Mohammed Kayed, Moheb Ramzy Girgis, and Khaled F. Shaalan. A survey of web information extraction systems. *IEEE Transactions on Knowledge and Data Engineering*, 18(10):1411–1428, 2006.

Li Changhua, Yang Bing, and Xie Weixin. Online hand-sketched graphics recognition based on attributed relational graph matching. In *WCICA 2000: Proceedings of the 3rd World Congress on Intelligent Control and Automation*, volume 4, pages 2549–2553, 2000.

Hui Chao and Jian Fan. Layout and content extraction for PDF documents. *DAS 2004: Proceedings of the 6th International Conference on Document Analysis Systems*, 3163:213–224, 2004.

Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(3):265–298, 2004.

Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. RoadRunner: Towards automatic data extraction from large web sites. In *VLDB 2001: Proceedings of the 27th International Conference on Very Large Databases*, pages 109–118, 2001.

D

Inderjit S. Dhillon. Co-clustering documents and words using bipartite spectral graph partitioning. In *KDD 2001: Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 269–274, 2001.

Hervé Déjean and Jean-Luc Meunier. A system for converting PDF documents into structured XML format. In *DAS 2006: Proceedings of the 7th International Workshop on Document Analysis Systems*, pages 129–140, 2006.

E

David W. Embley, Douglas M. Campbell, Yuan S. Jiang, Stephen W. Liddle, Deryle W. Lonsdale, Yiu-Kai Ng, and Randy D. Smith. Conceptual-model-based data extraction from multiple-record web pages. *Data & Knowledge Engineering*, 31(3):227 – 251, 1999.

Etymon Systems Inc. Etymon PJ Tools. `http://www.etymon.com/epub.htm`, 2009 (Web).

F

Bettina Fazzinga, Sergio Flesca, Andrea Tagarelli, Salvatore Garruzzo, and Elio Masciari. A wrapper generation system for PDF documents. In *SAC 2008: Proceedings of the ACM Symposium on Applied Computing*, pages 442–446, 2008.

Emilio Ferrara and Giacomo Fiumara. Web data extraction, applications and techniques: A survey. Technical report, Lixto Software GmbH, 2010.

Sergio Flesca, Salvatore Garruzzo, Elio Masciari, and Andrea Tagarelli. Wrapping PDF documents exploiting uncertain knowledge. In *CAiSE 2006: Proceedings of the 18th International Conference on Advanced Information Systems Engineering*, pages 175–189, 2006.

Robert P. Futrelle, Mingyan Shao, Chris Cieslik, and Andrea Elaina Grimes. Extraction, layout analysis and classification of diagrams in PDF documents. In *ICDAR 2003: Proceedings of the 7th International Conference on Document Analysis and Recognition*, volume 2, pages 1007–1013, 2003.

G

Edward Green. *Model-based analysis of printed tables*. PhD thesis, Rensselaer Polytechnic Institute, 1996. Adviser: Mukkai Krishnamoorthy.

Edward Green and Mukki Krishnamoorthy. Model-based analysis of printed tables. In *ICDAR 1995: Proceedings of the 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 214–217, 1995.

H

Jaekyu Ha, Robert M. Haralick, and Ihsin T. Phillips. Recursive X-Y cut using bounding boxes of connected components. In *ICDAR 1995: Proceedings of the 3rd International Conference on Document Analysis and Recognition*, pages 952–955, 1995.

Karim Hadjar, Maurizio Rigamonti, Denis Lalanne, and Rolf Ingold. Xed: a new tool for extracting hidden structures from electronic documents. In *DIAL 2004: Proceedings of the First International Workshop on Document Image Analysis for Libraries*, pages 212–224, 2004.

Robert M. Haralick. Document image understanding: geometric and logical layout. In *CVPR 1994: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 385–390, 1994.

Tamir Hassan. Object-level document analysis of PDF files. In *DocEng 2009: Proceedings of the 9th ACM Symposium on Document Engineering*, pages 47–55, 2009a.

Tamir Hassan. GraphWrap—a system for interactive wrapping of PDF documents using graph matching techniques. In *DocEng 2009: Proceedings of the 9th ACM Symposium on Document Engineering (Demonstration)*, pages 247–248, 2009b.

Tamir Hassan. User-guided wrapping of PDF documents using graph matching techniques. In *ICDAR 2009: Proceedings of the 10th International Conference on Document Analysis and Recognition*, pages 631–635, 2009c.

Tamir Hassan and Robert Baumgartner. Table recognition and understanding from PDF files. In *ICDAR 2007: Proceedings of the 9th International Conference on Document Analysis and Recognition*, volume 2, pages 1143–1147, 2007.

Tamir Hassan and Robert Baumgartner. Using graph matching techniques to wrap data from PDF documents. In *WWW 2006: Proceedings of the 15th International Conference on the World Wide Web (Poster track)*, pages 901–902, 2006.

John E. Hopcroft and Jin K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *STOC 1974: Proceedings of the 6th Annual ACM Symposium on Theory of Computing*, pages 172–184, 1974.

Chun-Nan Hsu and Chien-Chi Chang. Finite-state transducers for semi-structured text mining. In *Proceedings of the IJCAI-99 Workshop on Text Mining: Foundations, Techniques and Applications*, 1999.

Jianying Hu, Ramanujan Kashi, Daniel Lopresti, and Gordon Wilfong. Medium-independent table detection. In *Proceedings of Document Recognition and Retrieval VII*, 2000.

Jianying Hu, Ramanujan Kashi, Daniel Lopresti, and Gordon Wilfong. Table structure recognition and its evaluation. In *Proceedings of Document Recognition and Retrieval VIII*, 2001a.

Jianying Hu, Ramanujan Kashi, Daniel Lopresti, Gordon Wilfong, and George Nagy. Why table ground-truthing is hard. In *ICDAR 2001: Proceedings of the 6th International Conference on Document Analysis and Recognition*, pages 129–133, 2001b.

Jianying Hu, Ramanujan Kashi, Daniel Lopresti, and Gordon Wilfong. Evaluating the performance of table processing algorithms. *International Journal on Document Analysis and Recognition*, 4(3):140–153, March 2002.

Gerald Huck, Peter Fankhauser, Karl Aberer, and Erich Neuhold. Jedi: Extracting and synthesizing information from the web. In *Proceedings of the IFCIS International Conference on Cooperative Information Systems*, page 32, 1998.

Matthew Hurst. *The Interpretation of Tables in Texts*. PhD thesis, University of Edinburgh, 2000.

I

Yasuto Ishitani. Document transformation system from papers to XML data based on pivot XML document method. In *ICDAR 2003: Proceedings of the 7th International Conference on Document Analysis and Recognition*, pages 250–255, 2003.

Yasuto Ishitani. Logical structure analysis of document images based on emergent computation. In *ICDAR 1999: Proceedings of the 5th International Conference on Document Analysis and Recognition*, page 189, 1999.

J

Anil Jain and Yu Zhong. Page segmentation using texture analysis. *Pattern Recognition*, 29(5):743–770, 1996.

Duff Johnson. What is tagged PDF? `http://www.planetpdf.com/enterprise/article.asp?ContentID=6067`, 2005 (Web).

K

Thomas Kieninger. Table structure recognition based on robust block segmentation. In *Proceedings of the 5th SPIE Conference on Document Recognition*, pages 22–32, 1998.

Thomas Kieninger and Andreas Dengel. Applying the T-Recs table recognition system to the business letter domain. In *ICDAR 2001: Proceedings of the 6th International Conference on Document Analysis and Recognition*, pages 518–522, 2001.

Thomas Kieninger and Andreas Dengel. An approach towards benchmarking of table structure recognition results. In *ICDAR '05: Proceedings of the Eighth International Conference on Document Analysis and Recognition*, pages 1232–1236, 2005.

Thomas Kieninger and Andreas Dengel. A paper-to-HTML table converting system. In *Proceedings of Document Analysis Systems III*, 1998a.

Thomas Kieninger and Andreas Dengel. The T-Recs table recognition and analysis system. In *LNCS: Selected Papers from the Third IAPR Workshop on Document Analysis Systems: Theory and Practice*, 1998b.

Stefan Klink, Andreas Dengel, and Thomas Kieninger. Document structure analysis based on layout and textual features. In *DAS 2000: Proceedings of the International Workshop of Document Analysis Systems*, 2000.

Mikhail Kruk. PDFTOHTML conversion program. `http://pdftohtml.sourceforge.net/`, 2006 (Web).

Bernhard Krüpl and Marcus Herzog. Visually guided bottom-up table detection and segmentation in Web documents. In *WWW 2006: Proceedings of the 15th International Conference on the World Wide Web (Poster track)*, pages 933–934, 2006.

Bernhard Krüpl, Marcus Herzog, and Wolfgang Gatterbauer. Using visual cues for extraction of tabular data from arbitrary HTML documents. In *Proceedings of the 14th International Conference on the World Wide Web (Poster track)*, pages 1000–1001, 2005.

Nicholas Kushmerick, Daniel S. Weld, and Robert Dorrenbos. Wrapper induction for information extraction. In *IJCAI 1997: Proceedings of the International Joint Conference on Artificial Intelligence*, pages 729–737, 1997.

L

Alberto H. F. Laender, Berthier A. Ribeiro-Neto, and Altigran S. da Silva. DEByE—data extraction by example. *Data & Knowledge Engineering*, 40(2):121–154, 2002a.

Alberto H. F. Laender, Berthier A. Ribeiro-Neto, Altigran S. da Silva, and Juliana S. Teixeira. A brief survey of Web data extraction tools. *ACM SIGMOD Record*, 31(2): 84–93, 2002b.

Raymond S.T. Lee and James N.K. Liu. An oscillatory elastic graph matching model for recognition of offline handwritten Chinese characters. In *Proceedings of the 3rd International Conference on Knowledge-Based Intelligent Information Engineering Systems*, pages 284–287, 1999.

Wen-Jing Li and Tong Lee. Object recognition by sub-scene graph matching. In *ICRA 2000: Proceedings of the IEEE International Conference on Robotics and Automation*, volume 2, pages 1459–1464, 2000.

Bing Liu, Robert Grossman, and Yanhong Zhai. Mining data records in web pages. In *KDD 2003: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 601–606, 2003.

Josep Lladós, Enric Martí, and Juan José Villanueva. Symbol recognition by error-tolerant subgraph matching between region adjacency graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(10):1137–1143, 2001.

Daniel P. Lopresti and George Nagy. A tabular survey of automated table processing. In *GREC 1999: Selected Papers from the 3rd International Workshop on Graphics Recognition, Recent Advances*, pages 93–120, 2000.

William Lovegrove and David Brailsford. Document analysis of PDF files: methods, results and implications. *Electronic Publishing—Origination, Dissemination and Design*, 8(3):207–220, 1995.

M

Brendan D. McKay. Practical graph isomorphism. In *Congressus Numerantium*, volume 30, pages 45–87, 1980.

Bruno T. Messmer. *Efficient Graph Matching Algorithms for Preprocessed Model Graphs*. PhD thesis, University of Bern, 1996.

Bruno T. Messmer and Horst Bunke. A decision tree approach to graph and subgraph isomorphism detection. *Pattern Recognition*, 32(12):1979–1998, 1999.

Jean-Luc Meunier. Optimized XY-cut for determining a page reading order. In *ICDAR 2005: Proceedings of the 8th International Conference on Document Analysis and Recognition*, pages 347–351, 2005.

Ion Muslea, Steve Minton, and Craig Knoblock. STALKER: Learning extraction rules for semistructured, Web-based information sources. In *Proceedings of the AAAI-98 Workshop on Artificial Intelligence*, pages 74–81, 1998.

N

George Nagy. Twenty years of document image analysis in PAMI. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):38–62, January 2000.

George Nagy and Sharad Seth. Hierarchical representation of optically scanned documents. In *ICPR 1984: Proceedings of the 7th International Conference on Pattern Recognition*, 1984.

George Nagy, Sharad Seth, and Mahesh Viswanathan. A prototype document image analysis system for technical journals. *Computer*, 25(7):10–22, 1992.

R

Trygve Randen and John Håkon Husøy. Segmentation of text/image documents using texture approaches. In *In Proceedings of NOBIM*, pages 60–67, 1994.

Juan Raposo, Alberto Pan, Manuel Alvarez, Justo Hidalgo, and Angel Vina. The Wargo system: Semi-automatic wrapper generation in presence of complex data access modes. In *DEXA 2002: Proceedings of the 13th International Workshop on Database and Expert Systems Applications*, pages 313–320, 2002.

Maurizio Rigamonti, Oliver Hitz, and Rolf Ingold. A framework for cooperative and interactive analysis of technical documents. In *GREC 2003: Proceedings of the 5th IAPR Workshop on Graphics Recognition*, 2003.

Maurizio Rigamonti, Jean-Luc Bloechle, Karim Hadjar, Denis Lalanne, and Rolf Ingold. Towards a canonical and structured representation of pdf documents through reverse engineering. In *ICDAR 2005: Proceedings of the Eighth International Conference on Document Analysis and Recognition*, pages 1050–1055, 2005.

Massimo Ruffolo and Ermelinda Oro. PDF-TREX: An approach for recognizing and extracting tables from PDF documents. In *ICDAR 2009: Proceedings of the 10th International Conference on Document Analysis and Recognition*, pages 906–910, 2009.

Massimo Ruffolo and Ermelinda Oro. PDF-TREX dataset. `http://staff.icar.cnr.it/ruffolo/files/PDF-TREX-Dataset.zip`, September 2009 (Web).

Daniela Rus and Kristen Summers. Geometric algorithms and experiments for automated document structuring. In *Mathematical and Computer Modelling*, 1997.

S

Jürgen Schürmann, Norbert Bartneck, Thomas Bayer, Jürgen Franke, Eberhard Mandler, and Matthias Oberländer. Document analysis—from pixels to contents. *Proceedings of the IEEE*, 80(7):1101–1119, 1992.

Sargur N. Shari. Document image understanding. In *Proceedings of the 1986 ACM Fall Joint Computer Conference*, pages 87–96. ACM, IEEE Computer Society Press, 1986.

Statistik Austria. Leistungs- und Strukturstatistik 2007—Hauptergebnisse. `http://www.statistik.at/web_de/static/leistungs-_und_strukturstatistik_2007_-_hauptergebnisse_037117.pdf`, June 2009 (Web).

P. N. Suganthan and H. Yan. Recognition of handprinted Chinese characters by constrained graph matching. *Image and Vision Computing*, 16(3):191–201, 1998.

T

Nesime Tatbul, Olga Karpenko, Christian Convey, and Jue Yan. Data integration services. Technical report, Brown University, Computer Science Department, 2001.

TouchGraph LLC. Touchgraph library (free version). `http://touchgraph.sourceforge.net/`, September 2006 (Web).

Shuichi Tsujimoto and Haruo Asada. Understanding multi-articled documents. In *Proceedings of the 10th International Conference on Pattern Recognition*, 1990.

Zhuowen Tu and Song-Chun Zhu. Image segmentation by data-driven Markov chain Monte Carlo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5): 657–673, 2002.

U

J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the Association for Computing Machinery*, 23(1):31–42, January 1976.

Dept. of Informatics, Univ. Fribourg. XMIllum—The Extensible Markup Illuminator. `http://xmillum.sourceforge.net/`, September 2002 (Web).

W

Xinxin Wang. *Tabular Abstraction, Editing and Formatting*. PhD thesis, University of Waterloo, 1996.

Yalin Wang. *Document Analysis: Table Structure Understanding and Zone Content Classification*. PhD thesis, University of Washington, 2002.

K.Y. Wong, R.G. Casey, and F.M. Wahl. Document analysis system. *IBM Journal of Research and Development*, 26:647–656, 1982.

Y

Burcu Yildiz, Katharina Kaiser, and Silvia Miksch. pdf2table: A method to extract table information from PDF files. In *IICAI 2005: Proceedings of the 2nd Indian International Conference on Artificial Intelligence*, pages 1773–1785, 2005.

Z

Oren Zamir and Oren Etzioni. Web document clustering: a feasibility demonstration. In *SIGIR 1998: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 46–54, 1998.

Yanhong Zhai and Bing Liu. Web data extraction based on partial tree alignment. In *WWW 2005: Proceedings of the 14th International Conference on the World Wide Web*, pages 76–85, 2005.

Konstantin Zuyev. Table image segmentation. In *ICDAR 1997: Proceedings of the 4th International Conference on Document Analysis and Recognition*, volume 2, pages 705–708, 1997.

Appendix A

Classification of errors in table recognition

This appendix lists the various recognition errors that were encountered during evaluation of our table structure recognition algorithm and PDF-TREX (see Section 4.4.2) and how they were evaluated, i.e. which cells were given which classifications. The totals for both systems are given in Section A.3.

A.1 Cell errors

A.1.1 Splitting errors

1. single column in table is detected as two separate columns (see the example in Figure A.1):

   (a) content of cell is not split; additional blank cell is introduced
      - the cell is classified as **split full**; the resultant blank cell as **extra blank**

   (b) content of cell is split into two (or more) cells
      - the original cell is classified as **split data**; the resultant additional cell(s) as **extra data**

2. single, multi-line row is erroneously split into its constituent lines:
   - the original cell is classified as **split full** or **split data**, depending on whether the textual data has been split; the resultant additional cell(s) as **extra data** or **extra empty**

135

3. cell spanning several rows is not detected and split into individual rows:

   - the original cell is classified as **split full** or **split data**, depending on whether the textual data has been split; the resultant additional cell(s) as **extra data** or **extra empty**

4. cell (e.g. a heading) spanning several columns is not detected and split into its individual columns

   - the original cell is classified as **split full** or **split data**, depending on whether the textual data has been split; the resultant additional cell(s) as **extra data** or **extra empty**

A.1.2 Merging errors

1. horizontal merging of cells in adjacent columns (e.g. due to insufficient whitespace between them, as in the example in Figure A.2):

   (a) one or more cells detected as spanning; column structure remains in other rows
   - the spanning cell is classified once as **merged**; the remaining cells within it as **not recognized**; the remaining cells in the column are detected correctly in this case

   (b) no cells detected as spanning, i.e. all cells in column are merged and the entire column disappears
   - the spanning cells are classified once as **merged**; the remaining cells within them as **not recognized**; all remaining data and blank cells in the missing column are also classified as **not recognized**

2. merging of adjacent rows (i.e. 2 rows are seen as one multi-line row)

   - each resulting (incorrect) cell is classified as **merged**

A.1.3 Other errors

1. cells are split in one direction and merged in another (see the example in Figure A.3)

   - the horizontal error takes priority; in this example, the cell is classified as a single **merged** cell. Void cells resulting from the split are still counted as normal

Descrizione	Saldo indiz.	Incrementi		Decrementi	Saldo finale
Ratei	1.669		0	1.269	400
RATEI ATTIVI	1.669		0	1.269	400
Risconti	26.676		0	26.079	597
RISC. ATTIVI	26.676		0	26.079	597
Ratei	49.374		0	14.467	35.267
RATEI PASSIVI	49.374		0	14.467	35.267

Figure A.1: Example of a table with a split column. The PDF Analyser view is shown above; the resulting HTML table below

Figure A.2: Example of a table with a merged column. This is the Statistik Austria document from Section 5.1 .

137

2. an entire non-spanning column of a table is seen as spanning several columns, except for a few individual cells, which do not span the entire width of the column

   - in this case, the spanning cells are classified as having been **correctly detected**; the non-spanning cells are classified as **split full**, and the resulting empty cells as **extra blank**

3. cells, which fall within the rectangular boundary of the table, are not recognized as belonging to the table (e.g. in Figure A.4, they lie on the edge of the table and, due to the text being in a different font size, have been detected as surrounding text)

   - these cells are classified as **not recognized**; any empty cells in their place are **incorrect empty**

A.2 Table boundary errors

1. additional lines or columns are detected, outside of the table's actual boundary:

   - table is classified as **merged into surroundings**; the additional cells as **incorrect data** or **incorrect empty**

2. extra lines or other data is added to a cell along the edge of a table (but no extra cells are added to the table outside its boundary)

   - these cells are classified as **merged**; other cells in the row or column are unaffected; the table area classification is also unaffected (i.e., if no other errors are present, it is classified as **found correctly**)

3. lines or columns, which are part of the table, are not detected:

   - these are classified as **not recognized**

4. single table is split up into two or more tables across the data cells (see Figure A.5)

   - the first table is classified as **split**; the resulting additional tables as **extra table**

Figure A.3: Example of cells which are split in one direction and merged in another

Figure A.4: Example of missing cells within a table boundary

Figure A.5: Example of a partially recognized table being split up into two tables. Blank cells between the two tables are not classified as having been recognized

139

Figure A.6: Example of two horizontally neighbouring tables (or *sub-tables*) merged together.

- all cells within the split tables belonging to the original full table are classified as normal, even if their respective columns or rows have been split across several tables; cells that have not been detected (e.g. between two split tables) are classified as **not recognized**

5. single table is split, but only across the heading/access cells (i.e. the heading cells are separated from the data; all data cells remain together)

  - the table containing all data cells is classified as **data cells found**; the resulting additional tables as **extra table**

  - all cells within the split tables belonging to the original full table are classified as normal, even if their respective columns or rows have been split across several tables; cells that have not been detected (e.g. between two split tables) are classified as **not recognized**

6. neighbouring tables are merged; rows and/or columns align with each other

  - the first table is classified as **merged**; the resulting additional tables as **not recognized**. Cells from both original tables are classified normally

7. two horizontally neighbouring tables (or sub-tables) are merged and rows do not align (see Figure A.6)

140

- these tables may appear to be part of one large table. But, as their rows do not align with each other, it was decided to interpret these tables as separate tables. Therefore, in this example, the first is classified as **merged**; the second as **not recognized**. Cells in both tables are classified normally, although it is worth noting that a large number of **incorrect blank** rows and **merged cells** result as a result of the merge

A.3 Classification totals for both systems

Based on the classifications described in this Appendix, the results are shown in Tables A.1 and A.2.

Table areas		Our system	PDF-TREX
Total areas		126	
Found correctly	TP	67	53
Data cells found	TP	22	17
Partially found	TP/FP	13	3
Split table	TP/FP	8	24
Extra table (from split)	FP	12	49
Incorrect table	FP	19	22
Merged into surroundings	TP	8	9
Merged	TP/FP	1	4
Not recognized	TN	9	16

Table A.1: Totals of table area classifications in the dataset

Table cells		Our system	PDF-TREX
Total cells		10120 (9185 data; 935 blank)	
Found correctly data	TP	7489	8217
Found correctly blank	TP	718	806
Split full	TP	431	411
Split data	TP/FP	159	204
Split blank	TP	13	27
Extra data	FP	228	266
Extra blank	FP	935	1260
Incorrect data	FP	105	291
Incorrect blank	FP	83	534
Merged	TP/FP	28	104
Not recognized data	TN	1004	162
Not recognized blank	TN	202	83

Table A.2: Totals of table cell classifications in the dataset

C U R R I C U L U M   V I T A E

Tamir Hassan, MEng

Kolingasse 3/18, 1090 Wien, Austria

Tel: +43 676 721 6288 • **Email:** tamir@tamirhassan.com • **Web:** http://www.tamirhassan.com

Summary of research interests

In my doctoral thesis, I have dealt with the topic of document understanding of PDF files, with the goal of making them amenable to *wrapping*, or user-guided mass data extraction. In the first part of this work, I have worked on page segmentation and table detection, with the goal of converting PDF files into HTML so that they can be wrapped using existing methods. More recently, I have developed the *GraphWrap* system, a novel approach to wrapping, which uses methods based on subgraph isomorphism to locate wrapping instances on a graph-based representation of the document.

A further interest of mine is in document authoring, and I believe that there is still much room for improvement in current tools and file formats to enable efficient creation, storage and repurposing of documents. I also have a great passion for digital typography, a field which I believe still provides opportunities for further advancement, and in which further research could provide a great contribution.

My doctoral work has been published in international conferences such as ICDAR and DocEng, and a prototype of *GraphWrap* has also been demonstrated at CeBIT 2009.

Selected publications

- Hassan, T.: Object-Level Document Analysis of PDF Files, DocEng 2009, Munich, Germany
- Hassan, T.: GraphWrap – A System for Interactive Wrapping of PDF Documents Using Graph Matching Techniques (demonstration), DocEng 2009, Munich, Germany
- Hassan, T.: User-Guided Wrapping of PDF Documents Using Graph Matching Techniques, ICDAR 2009, Barcelona, Spain
- Hassan, T., Baumgartner, R.: Table Recognition and Understanding from PDF Files, ICDAR 2007, Curitiba, Brazil
- Carme, J., Ceresna, M., Frölich, O., Gottlob, G., Hassan, T., Herzog, M., Holzinger, W., Krüpl, B.: The Lixto Project: Exploring New Frontiers of Web Data Extraction. BNCOD 2006, Belfast, UK
- Hassan, T., Baumgartner, R.: Using Graph Matching Techniques to Wrap Data from PDF Documents, WWW 2006 (Poster track), Edinburgh, UK
- Hassan, T., Baumgartner, R.: Intelligent Text Extraction from PDF Documents, short paper, IAWTIC 2005, Vienna, Austria

Awards and Achievements

- FIT-IT Dissertation Fellowship for the *GraphWrap* project; a one-year, one-man project which I initiated, and resulted in a novel approach to supervised data extraction being developed to prototype stage
- Mobility scholarship from the *Akademisch-soziale Arbeitsgemeinschaft Österreichs* to work for three months (March – May 2010) with Prof. Roger Hersch, Ecole Polytechnique Fédérale de Lausanne, Switzerland, on parameterized representations of fonts based on shape components

Education

2005 – 2010	Technische Universität Wien, Vienna, Austria
	Doctorate in Natural Sciences (Dr.rer.nat.)
	    at the Database and Artificial Intelligence research group
	Supervisor: Prof. Georg Gottlob
	Thesis topic: User-guided data extraction from print-oriented
	    documents

2000 – 2004	University of Warwick, Coventry, West Midlands, UK
	Master of Engineering (MEng) in Computer Science
	    with Honours (upper second class)
	Projects undertaken: PDF to HTML Conversion;
	    A Visual Editor for TV Programme Guide Information

1998 – 2000	Christ the King Sixth Form College, Lewisham, London, UK
1994 – 1998	Bexley & Erith Technical High School, Bexley, Kent, UK

Employment

Jan 2008 to date	Research Assistant, Database and Artificial Intelligence and
	    Pattern Recognition and Image Processing research groups,
	    Technische Universität Wien, Vienna, Austria
	DBAI: worked on research project *GraphWrap* on the use of graph
	    matching methods to extract data from PDF files
	PRIP: working on comparison of vector graphics in PDF;
	    supervising undergraduate thesis in watermarking of
	    images in print media

Apr – Aug 2007	Solution Consultant, Professional Services Team, Lixto Software
	    GmbH, Vienna, Austria
	Tasks included consulting with customers, and specifying and
	    building complete data extraction solutions for them using
	    the *Lixto* suite of products originally developed at TU Wien

Jan 2005 – Mar 2007	Research Assistant, Database and Artificial Intelligence research
	    group, Technische Universität Wien, Vienna, Austria
	Tasks included researching document analysis methods and their
	    use in data extraction from PDF files

Jul 2004 – Dec 2004	IT Technician, Northbrook School, Lee, London, UK
	Tasks included server management, training, problem solving and
	    playing a key role in the design, specification and
	    implementation of the school's plan to migrate from paper-
	    based records for attendance and assessment (examinations)
	    to an online database system

Nationality:	British
Languages spoken:	English (native); German (fluent); Polish (intermediate); French (basic)
Driving licence:	Full (manual and automatic); class B