Object-Level Document Analysis of PDF Files

Tamir Hassan
Database and Artificial Intelligence
Information Systems Institute
Technische Universität Wien
Favoritenstraße 9-11, A-1040 Wien, Austria
hassan@dbai.tuwien.ac.at

ABSTRACT

The PDF format is commonly used for the exchange of documents on the Web and there is a growing need to understand and extract or repurpose data held in PDF documents. Many systems for processing PDF files use algorithms designed for scanned documents, which analyse a page based on its bitmap representation. We believe this approach to be inefficient. Not only does the rasterization step cost processing time, information is also lost and errors can be introduced.

Inspired primarily by the need to facilitate machine extraction of data from PDF documents, we have developed methods to extract textual and graphic content directly from the PDF content stream and represent it as a list of "objects" at a level of granularity suitable for structural understanding of the document. These objects are then grouped into lines, paragraphs and higher-level logical structures using a novel bottom-up segmentation algorithm based on visual perception principles. Experimental results demonstrate the viability of our approach, which is currently used as a basis for HTML conversion and data extraction methods.

Categories and Subject Descriptors

I.7.5 [**Document and Text Processing**]: Document Capture—*document analysis*; H.3.3 [**Information Systems**]: Information Search and Retrieval

General Terms

Algorithms, Experimentation

1. INTRODUCTION

In recent years, PDF has become the *de facto* standard for exchanging print-oriented documents on the Web. Its popularity can be attributed to its roots as a page-description language. Any document can be converted to PDF as easily as sending it to the printer, with the confidence that the formatting and layout will be preserved when it is viewed or printed across different computing platforms. However, the print-oriented nature of PDF also provides a significant drawback: PDFs contain very little structural information about the content held within them, and extracting or repurposing this content is therefore a difficult task.

In the last few decades, there has been much work in the field of *document understanding* which aims to detect logical structure in unstructured representations of documents: usually scanned images. Many of these approaches have also been applied to PDF. Many of these methods simply make use of a bitmap rendition of each page of the PDF file at a given resolution and apply methods similar to those designed for scanned pages. Relatively little information, typically only the text, is used from the original PDF source. Other approaches do examine the PDF source code but make somewhat limited use of the information. Section 2 describes these approaches in more detail.

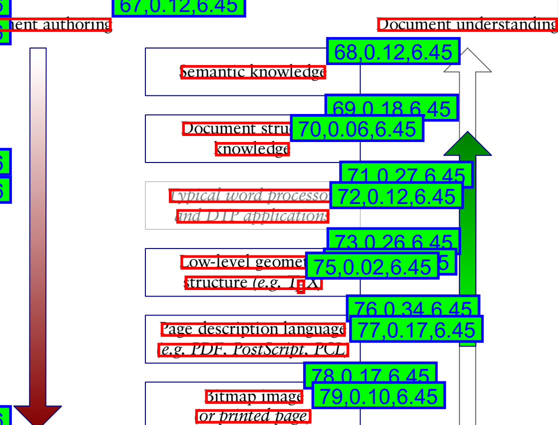

Figure 1: Document representation hierarchy

Figure 1 gives an overview of the document authoring process and the various levels of abstraction in which a document is represented during the document authoring process: from semantic concepts before any words have been written at the start to the printed image of the page at the end. Document understanding is essentially the opposite of document authoring. We believe the PDF representation to be a logical step *above* the printed page[1], and therefore that

DocEng'09, September 16–18, 2009, Munich, Germany.

[1] Please note that we are referring to PDF files which have been generated digitally, usually directly from a DTP or word-processing application, regardless of whether they are *tagged* or not. More information is given in Section 6.

performing document analysis on a bitmap rendition of a PDF page is taking a step "backwards", resulting in useful information being lost as well as additional processing overhead.

In this paper we present *PDF Analyser*, a system for processing, displaying and analysing PDF documents, which works exclusively on the object level; this means that items on the page are represented on various granular levels in sets of rectangular objects. Section 3 contains the two main contributions of this paper: we describe how these objects are obtained from the text and graphic instructions in the PDF source code and introduce the "best-first clustering" algorithm, which segments the page in a bottom-up fashion guided by principles based on visual cognition. Section 4 describes concisely how we use the results of our analysis in real-world data-extraction applications and refer the interested reader to our other relevant publications. The final two sections present experimental results and a concluding discussion.

2. RELATED WORK

There has been much research in analysing documents based on scanned images [2, 3], in which segmentation is primarily performed by carrying out pixel-based operations. To our knowledge, the first publication that deals with the analysis of PDF files is the paper by Lovegrove and Brailsford [10]. This paper focuses only on textual objects; the Adobe Acrobat SDK is used to obtain pre-merged lines in object form and bottom-up segmentation techniques are described.

Since then, the PDF format has gained popularity and there have been a number of research groups targeting PDF. Anjewierden [4] developed a method in which text and graphic objects are extracted from the PDF using methods based on top of `xpdf`. Further processing is performed by grammars, which results in the system being domain-specific. Hadjar et al. [7] also introduce a system for analysing PDF using text and graphic objects and analyse 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). Unfortunately, neither paper describes in detail how the low-level text and graphic instructions are processed to generate the resulting objects.

Futrelle et al. [6] describe a system for graphics recognition from PDF. Here, the *Etymon PJ Tools*[2] library is used to obtain the graphics primitives in object form. Of course, for this application, the extracted information is at a much finer level than what we require for document analysis.

Chao and Fan [5] have developed a 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 detail on how the initial objects are obtained from PDF.

We hope that our publication fills the gap in providing a detailed description of how the relevant PDF instructions can be processed to form a set of objects suitable for

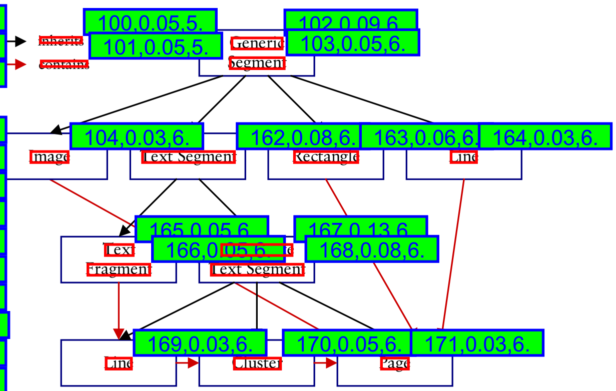

Figure 2: Element hierarchy

document analysis. The second main contribution of this paper is our novel *best-first clustering algorithm*, a robust bottom-up segmentation algorithm based on visual principles, which takes a list of rectangular-based objects as input and produces good results, even on complex layouts such as newsprint.

3. IMPLEMENTATION

3.1 Our PDF model

In order to perform our analysis on PDF documents and facilitate further processing, we have devised a model based on rectangular objects. The limitation of rectangular bounding boxes for objects allows our model to be relatively simple, yet it offers enough granularity to represent the document successfully for document analysis and understanding purposes. The object hierarchy is implemented in Java using inheritance and is shown in Fig. 2. The root item in the hierarchy is `GenericSegment`, which provides the four rectangular co-ordinates and associate methods. Nesting of objects is possible with the class `CompositeTextSegment`.

3.2 Object extraction from PDF

Our PDF parsing method is based on *PDFBox*[3], an open-source library for processing PDFs. We extend the `PDFStreamEngine` class, which is responsible for processing the stream of instructions on a page, to extract and store the objects for our representation as well as deal with page boundaries and rotation.

By extending the `OperatorProcessor` class, it is possible to define which actions are taken when a particular PDF instruction occurs. 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 have implemented are shown in Fig. 6. In particular, we aim to extract all text and bitmap image blocks, but only certain graphic items, such as ruling lines and rectangles, which are likely to help us understand the page better, and not logos or illustrations.

3.2.1 Text elements

The PDF specification [1] contains a number of operators for positioning text on the page. The positioning of characters is already handled by PDFBox's `PDFStreamEngine` class

[2]Etymon PJ Tools, `http://www.etymon.com/epub.html`

[3]PDFBox, `http://www.pdfbox.org`

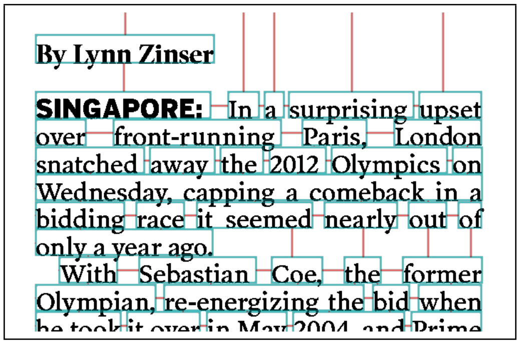

Figure 3: An example of a paragraph as represented by the *text fragments* which are obtained directly from the PDF source code and joined by edges representing adjacency. Here we can see the "brickwork effect": the entire paragraph could be built by joining just the vertical edges. The initial merging of successive pairs of blocks has resulted in th with tightly-spaced being merged completely

in the `showString` method, leaving the developer to conc trate on the actions to be taken when a string is shown.

Text can be placed on the page by two operators: `Tj` *(shou 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 provide their own kerning alg 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, 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. It is worth noting that this only certain tables and never with columns of text.

As we did not wish to risk overmerging the blocks, we kept 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 we could not completely avoid overmerging text fragments at this stage and would therefore need to split them later, as shown in the example in Fig. 9. This probl ed in the penultimate paragraph of section 3.4.

Finally, it is worth noting that characters (or complete strings) are sometimes *overprinted* with a slight offset to simulate boldface type. As long as these instructions follow another, they are automatically detected and represented a single text fragment with the boldface flag set to true.

Coordinate systems. 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 be easily placed in a PDF. Fortunately for us, the existing methods take care of all the translation operators.

3.2.2 Graphic elements

Bi map images are relatively straightforward. An image ed on the page either using the `Do` *(invoke)* instruc as an inline image using the `BI` *(begin inline image) age data)* and `EI` *(end inline image)* instructions, to with its rectangular co-ordinates before scaling and rmation. The only main pitfall is that of clipping we found it very common that the actual image would occupy a larger area than what was visible on the page, and that these extra parts of the image would be clipped using a rectangular clipping path (see below). We imagine that this is the result of the cropping functionality in common p publishing systems, which simply send printer in the most straightforward manner.

tor elements are a greater challenge for us, as we differentiate between objects which are parts of vec ages (such as illustrations and diagrams) and objects play a dominant role in conveying the logical struc the page to the reader, such as ruling lines and boxes. orth in the latter, curved segments are used.

PDF, vector graphics are drawn by defining a *path* which comprises one or more connected *subpaths*. A new h is begun by the `m` *(moveto)* operator. Straight line nts are drawn by the `l` *(lineto)* operator, curves by the *ve to)*, `v` *(curve to replicate initial point)* and `y` *(curve icate final point)* operators, and rectangles by the `re` *d rectangle to path)* operator. The operator `h` *(close* the subpath with a straight line back to the starting inate. A rectangle is equival ng three line nts and closing the subpath.

our simplified model only includes line and rectangle s, we approximate bezier curves with straight lines h their coordinate parameters. (In fact, we discard hs which include curves; we only need to store them stage in case they are later used to define a clipping boundary). Subpaths which include curves are flagged as We store all generated subpaths until they are either d by the `S` *(stroke path)* or `s` *(close and stroke path* ors, filled by the `f` *(fill non-zero rule)* or `f*` *(fill even- le)* operators or the path is ended. The `n` *(end path* or clears the path without stroking or filling; it is lly only used to clear the path af path en defined (see the section below).

en we come across a stroking or filling operator, we eck that the current colour's grey value lies below a certain threshold. If so, we represent each subpath which hs only vertical and horizontal lines and/or rectangles s respective objects in our simplified model. If a clip- rea is active, we first clip the objects. If the width or is above a minimum threshold (defined as 3 × modal t size of all text blocks on the page) and if, according to our heuristic, no other smaller or curved graphic objects

Figure 4: Page display without clipping (left) and with clipping (right) of an image

are nearby, the objects are repr[illegible]rwise they are assumed to be part of a graphic.

We find that the above treatment of PDF vector graphic instructions enables us to obtain a simplified representation of the most inportant 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 [illegible]y we require for performing document analysis.

Rectangles and lines. In many cases, we found that ruling lines on pages are 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 [illegible]out 5 pt), the object is re-classified 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 clip 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 give[illegible] results for our purposes, as shown in Figure 4.

To summarize, the table in Fig. 6 lists the PDF operators that we have implemented, the PDF operators which were already present in the PDFBox code and whose implementation was not altered by us, and th[illegible] which are [illegible]plemented at all in our system.

[illegible] The GUI

[illegible]rder to visually display our analysis and segmentation [illegible], we have built a GUI on top of the *XMIllum* framework[4], which allows a user to interactively open a document, select the desired level of granularity and show or hide the [illegible]ua[illegible]es. A screenshot of the GUI is shown [illegible] 5.

[illegible]n opening a document, the GUI makes a call to [illegible]stScript/PDF interpreter *Ghostscript*[5], to create a [illegible] version of the document, which we overlay as a background image. This makes it possible to easily compare the output of our anal[illegible]mentation methods with the original document.

[illegible] did have some initial problems in ensuring that [illegible]rocessing results were correctly aligned with the [illegible]script output. Whereas newer versions of `gs` automat[illegible] rotate the page, older versions do not. Also, because [illegible] page can have several bounding boxes defined (e.g. [illegible]p marks), Ghostscript was not always consistent in [illegible]ice of bounding box. Using [illegible]pBox switch [illegible] to have solved this problem.

[illegible] Best-first clustering algorithm

[illegible]r parsing through all the instructions on a PDF page, [illegible]ain a list of *text fragments*, which correspond loosely [illegible] individual (sub)instructions for displaying text. The [illegible]rst clustering algorithm merges these text fragments in a bottom-up fashion to represent single logical elements. [illegible]nsistency, we will use the term *cl*[illegible]r to text [illegible] at this final level of granularity.

[4] XMIllum, `http://xmillum.source`[illegible]

[5] Ghostscript, `www.ghostscript.com`

Figure 5: An example of the GUI, based ***m*, showing the results of our segmentation algorithm overlaid on a bitmap rendition of the page**

3.4.1 *Initial processing*

We take as input a list of text fragments, which may contain anything from one to several characters each, and are clearly oversegmented at this stage. As a complex page could contain as many as ten thousand of these segments, we first aim to reduce this to a more manageable number to keep processing time of ages 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. Certainly text fragments corresponding to a single TJ instruction are always returned together. Therefore, it makes sense to first process this list linearly (which costs relatively little processing time) and join neighbouring segments if they are on the same line. We use a threshold of 0.25 × font size; between 0.25 × font size and 1.0 × font size, we merge the blocks but assume th characters belong to separate words and a space is added.

After this initial step, we perform a merging procedure ge horizontally neighbouring blocks which were not n sequentially to the PDF. We sort the blocks in Y- order; this means that blocks with similar baselines urned together in left-to-right order, and that these ual lines of text are then sorted from top to bottom. n join any neighbouring blocks if they are on the same line and so close together that they could not conceivably to different columns we use a very tight old of 0.2 × font size. reason we allow for a greater threshold in the former because we are only comparing neighbouring items at age. As most text is written to the PDF in its reading the chances of overmerging are very low. Further- the threshold of 1.0 × font size is still low enough merge across neighbouring columns of text. Should rging occur, for example in tight tabular layouts as in the example in Fig. 9, the method described in nal paragraph of this section would take care of up to ermerged lines in a text block. In the latter merging process, we are comparing each block with every the likelihood of overmerging is therefore greater.

	Operators
Implemented by us	B, BI,c, CS, cs, Do, f [illegible] K, k, l, m, n, q, Q, re, RG, rg, s, S, Tj, TJ, v, w, W, W*, y
Already impleme[illegible] Box	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, [illegible], EMC, EX, G, g, i, ID, j, J, M, MP, ri, SC, sc, SCN, scn, sh

Figure 6: A list of operators which are implemented in our system

3.4.2 Adjacency graph representation

Now that we have reduced the amount of text fragments to an acceptable number, we form an *adjacency graph* from these text objects. This graph structure is used as a basis for our best-first clustering algorithm and allows us to directly access the neighbours of each text block. In our graph, direct neighbours (regardless of distance) are joined by edges in each of the four directions; north, south, west and ea[illegible]. More precisely, the graph is formed in the following way:

- two lists are generated, `horiz` and `vert`, which contain all text blocks sorted in horizontal [illegible] order of midpoint coordinate respectively;
- each text block is examined in turn and its position located in both lists. Starting from these positions, the lists are examined in ascending and descending order, which corresponds to looking for the next neighbour[illegible] block in each of the four directions of the compass;
- as soon as a block is reached whose midpoint Y coordinate (if looking horizontally) or midpont X coordinate (if looking vertically) intersects that of the current block *and vice versa*, this block [illegible] its neighbour in that particular direction;
- after a neighbour is found in a particular [illegible] do not look any further in that direction.

```
- edges with vertical direction
  - edges where the font s[illegible] segments is
    approximately the same
    - smaller font sizes first
      - smaller line spacing (edge length) first
        - if the line spacing is approximately the
          edges with approx. identical width first,
          otherwise, o[illegible] spacing (edge length);
          lowest first
        - if line spacing approx. same but widths
          sort by width differenc[illegible]rst)
- edges with horizontal direction
  - shorter edges (edge length) first
```

Figure 7: Orderi[illegible]dges in the best-first clustering algorithm

The above method generates a list of neighbours within "line of sight" of each block. We then ensure that, for every neighbouring relation $A \rightarrow B$, a corresponding relation $B \rightarrow$ [illegible] A exists and remove any duplicate adjacency relations. An example of this graph structure is shown in Fi[illegible] graph structure is described in more detail in [8].

Each of the neighbourhood relations is represented as an `Edge` object with attributes such as `fontsize`, the average font size of the two nodes, and `length`, the closest distance [illegible]nts between the edges of both segments relative to [illegible]ize, as well as `nodeFrom` and `nodeTo`, the two text [illegible] which the edge connects. Non-textual segments are [illegible]d. After generation, any [illegible]cross a detected [illegible] *line* are also discounted.

[illegible] to its name, our *best-first clustering* algorithm first [illegible]s together edges where it is obvious that they belong [illegible]he same logical block. After most of these blocks have already been formed, the more problematic edges are then [illegible]hed, for which it is not possible to determine *a priori* [illegible]er they should be clustered together or left apart. At this stage, a better decision can be made, as the block structure is already partly present. As this process is based on [illegible]stalt laws of proximity and similarity, we believe it to [illegible]ilar to the way a human reader would analyse a page. [illegible] of these processes occur at a sub-conscious [illegible]

The first stage is to sort the edges into an appropriate [illegible] such that the most likely edges wi[illegible] first. [illegible]dering sequence is shown in Fig. 7. [illegible]edge lengths are always relative to font size.

$$\text{edge length} = \frac{\text{shortest length between t[illegible]}}{\text{average font size}}$$

As we are working only with text blocks, we ignore any edges which jo[illegible]s 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. This is because of the "brickwork effect": we [illegible]hat words in a paragraph rarely occur directly below each other, and that each word often has more than one neighbour above or below. This way, we can build most [illegible]s completely just from the vertical edges alone (See Fig. 3). In fact, in the rare case that words in a paragraph do line up vertically, this already begins to appear as tabulated data to the human reader, and this is why we need to exercise great care when joining horizontal edges. Therefore, horizontal edges are only visited after all vertical edges have been processed. Only at the very end of processing, any remaining unconnected horizontal neig[illegible]lly single lines) are joined together if necessary.

[illegible] pseudocode for our clustering algorithm is shown in [illegible] and refers to an external method `clusterTogether`. [illegible]ctice, the implementation is somewhat more co[illegible] as hash maps are used to improve performance. [illegible] method `clusterTogether` uses a number of heuristics to decide whether the clusters belong to [illegible] For [illegible]l edges, this method returns `true` if:

- [illegible]he new item(s) to be added are consi[illegible]he line spacing of the existing cluster; and

```
w[illegible] in list)
{
  get next edge
    i[illegible] rom and edge.nodeTo not yet in output
    {
      i[illegible] gether(edge.nodeFrom, edge.nodeTo, edge))
      {
        create new cluster [illegible] deFrom
          as single subitem
        create new cluster [illegible] deTo
          as single subitem
      [illegible] usters to output
    [illegible]
    }
    e[illegible] nodeFrom not yet in output
    {
      clust <- find cluster containing edge.nodeTo
      i[illegible] gether(edge.nodeFrom, clust, edge))
      {
      [illegible] deFrom to clust
    [illegible]
    }
    e[illegible] nodeTo not yet in output
    {
      clust <- find cluster containing edge.nodeFrom
      i[illegible] gether(edge.nodeTo, clust, edge))
      {
      [illegible] deFrom to clust
    [illegible]
    }
    e[illegible] nodes already in output
    {
      clust1 <- find cluster containing edge.nodeFrom
      clust2 <- find cluste[illegible] edge.nodeFrom
      i[illegible] clust2)
      {
        i[illegible] gether(clust1, clust2, edge))
        {
        [illegible] 1 with clust2
      [illegible]
    [illegible]
  [illegible]
}
```

Figure 8[illegible] ode of the best-first clustering algorithm

- the font sizes are approximately the same[6]

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 nor[illegible] but can be increased in the following situations:

- As blocks containing fewer lines of text are most likely to have been not fully clustered by the algorithm, the heuristic allo[illegible] reased 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 [illegible] nearest vertical neighbour distance is large.

`clusterTogether` then returns `true` if:

- the new item(s) to be added are c[illegible] h the font size of the existing cluster; and
- the edge width (i.e. the horizontal distance with respect to f[illegible] 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 [illegible] en headings of adjacent columns can, in special cases, even be less than the normal word spacing, as shown in the example[illegible] his can even occur if no ruling lines are present.

The human reader still recognizes the delineation between each individual column heading because of the clear column-[illegible] structure below, and because the headings are still [illegible] ently aligned with the data in these columns. We [illegible] ore check for such structures at every iteration of the [illegible] ntation process. After the columns have been clus-[illegible] ogether, our heuristic detects that the text block has [illegible] ped one or more "chasms" [illegible] the headings (maximum 2 lines) appropriately.

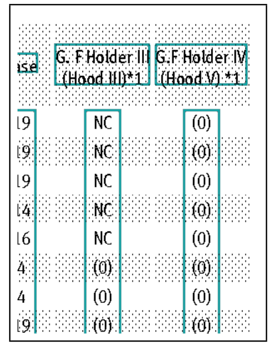

Figure 9: Example of tightly arranged column headings, which need to be accou[illegible] a later stage of the segmentation process

[illegible] FURTHER PROCESSING

[illegible] he motivation of our work stems from data extraction [illegible] PDF, we use the [illegible] r analysis algorithms in the following two ways:

- **Table detection:** We use the list of blocks as input to our table-detection algorithm [9]. As a later improvement, we have replaced the candidate column finding method with an updated version of the best-first algorithm: we search for candidate columns simultaneously with clusters; a separate `clusterTogether` method with increased thresholds is used to determine

---

[6]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 ad[illegible] respective paragraphs at the end of processing.

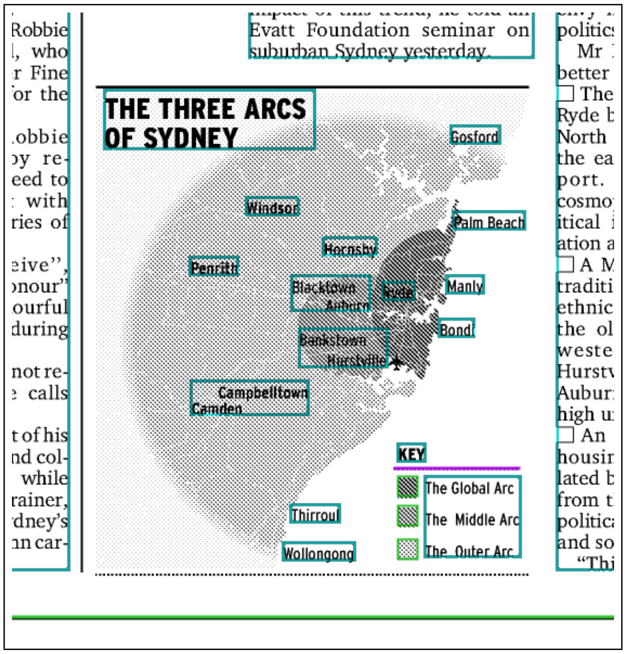

Figure 10: An example of segmentation errors in text inside a diagram

which blocks should be merged together to form these columns. The resulting tables are then represented in HTML, where they can be wrapped using [illegible] HTML wrapping systems such as *Lixto*[7].

- **Wrapping using graph matching techniques:** As described in [8], we use our graph-based representation as a basis to perform data extraction in an interactive fashion using an algorithm based on subgraph isomorphism. After the segmentation is complete, a further adjacency graph is created to join neighbouring clusters in the result, which is used to represent the document and perform data extraction. The user can also choose to perform wrapping on the line level; in that case, the blocks are [illegible] up into lines before the graph is generated.

5. EXPERIMENTAL RESULTS

We have tested our object extraction and segmentation algorithms on the front pages of 50 different issues of the Sydney Morning Herald[8] and visually compared the found objects with what a human reader would deem to be the correct result or [illegible]uth". The results are shown in the table below.

Item ty[illegible]	[illegible]otal	Detected	False pos[illegible]
Clusters	[illegible]	[illegible].3%)	[illegible])
Ruling lines	[illegible]	[illegible]4%)	[illegible]
Bitmap im[illegible]	[illegible]	[illegible]8%)	[illegible])
Rectangles	568	536 (94.4%)	45 (7.9%)

The experimental evaluation raised two important issues:

Firstly, in our case, the ground truth was very open t[illegible] interpretation, as exemplified in the following questions:

- Which lines on the page are materially important in gaining an understa[illegible] document's structure and which are not?
- Should [illegible] paragraphs 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 algorithms at a later stage. It was therefore very difficult to generate quantitatively measured results, as the evaluation process is 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 [illegible]ations (this is a[illegible]ned for a later stage in the processing pipeline).

In general, we found our best-first segmentation algorithm to produce very good results, as objects were rarely split or [illegible]erged. Because our dataset included a large number [illegible]rams with text labels, the ratio of correctly detected [illegible]s 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. 10. [illegible] alternative interpretation would be to class these la[illegible] parts of images and therefore as false positives, which [illegible] lead to a significantly higher recall value. In practice, [illegible] not interested in text in diagrams, which we ignore [illegible]n in the processing pipeline. Unfortunately, we found [illegible]ur evaluation strategy did not discriminate between [illegible]ortant errors in diagrams and catastrophic segmenta[illegible]rors, for example when two columns of an article are [illegible]d together. F[illegible]he latter type of error was a [illegible] occurrence.

Our algorithms did also return some false positives, in particular for ruling lines, which were found, on 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. [illegible]r purposes, this is not a big problem at all, as in our [illegible]rocessing steps, vector objects not in the vicinity of [illegible]e ignored anyway. Although the result is more than [illegible]ate for our purposes, further development on our vec[illegible]gram/image recognition he[illegible]d result in this number being significantly lower.

[illegible]ven with digitally generated PDFs, certain graphic ele[illegible]nts on the page (in particular advertisements) would have [illegible]r text included in bitmap or vector form, rather than as [illegible]t instructions. The same applies to logos[illegible]hese [illegible]be generally text items of little interest.

[illegible]inally, a number of errors occurred where major ruling [illegible]ere 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 transfo[illegible]rix, rather than a problem with our approach.

[7] Lixto, `www.lixto.com`

[8] The Sydney Morning Herald, `www.smh.com.au`

6. CONCLUSION AND DISCUSSION

In this paper we have presented in detail an efficient approach for extracting textual and graphical data from PDF documents at a level of granularity suitable for document analysis purposes. We have also presented a bottom-up segmentation algorithm, which copes well even with complex layouts, to group these segments into blocks representing logical elements on the page. We have described two use-cases in which this extracted data is processed further and used in real-world applications. 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 believe that the resulting data is of more than sufficient quality for these purposes.

In developing the extraction algorithms from PDF, we noticed that the structure of the PDF and the ordering of the operators usually represents 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 file usually resembles the reading order of the page;
- text in subinstructions within a single `Tj` instruction almost always belongs to the same 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 would make for an interesting research project.

Further areas in which this work could be extended include: using tagging information (where present) to improve the result, further development of the methods within the PDFBox library to improve robustness and extending the system to cope with non-Manhattan (rectangular) layouts.

7. ACKNOWLEDGEMENT

This work is supported by the Austrian Federal Ministry for Transport, Innovation and Technology under programme 815128/9306.

8. REFERENCES

[1] Adobe Systems Incorporated. *PDF Reference*.

[2] M. Aiello, C. Monz, L. Todoran, and M. Worring. Document understanding for a broad class of documents. *International Journal of Document Analysis and Recognition*, 5(1):1–16, 2002.

[3] O. Altamura, F. Esposito, and D. Malerba. Transforming paper documents into xml format with wisdom++. *International Journal of Document Analysis and Recognition*, 4(1):2–17, 8 2001.

[4] A. Anjewierden. Aidas: incremental logical structure discovery in pdf documents. In *International Conference on Document Analysis and Recognition 2001, Proceedings*, 2001.

[5] H. Chao and J. Fan. Layout and content extraction for pdf documents. *Document Analysis Systems VI*, 3163/2004:213–224, 2004.

[6] R. P. Futrelle, M. Shao, C. Cieslik, and A. E. Grimes. Extraction, layout analysis and classification of diagrams in pdf documents. In *International Conference on Document Analysis and Recognition 2003, Proceedings*, volume 2, page 1007, 2003.

[7] K. Hadjar, M. Rigamonti, D. Lalanne, and R. Ingold. Xed: a new tool for extracting hidden structures from electronic documents. In *Document Analysis for Libraries, 2004. Proceedings*, 2004.

[8] T. Hassan. User-guided wrapping of pdf documents using graph matching techniques. In *International Conference on Document Analysis and Recognition 2009, Proceedings*, 2009.

[9] T. Hassan and R. Baumgartner. Table recognition and understanding from pdf files. In *International Conference on Document Analysis and Recognition 2007, Proceedings*, volume 2, pages 1143–1147, 2007.

[10] W. Lovegrove and D. Brailsford. Document analysis of pdf files: methods, results and implications. *Electronic Publishing – Origination, Dissemination and Design*, 8(3):207–220, 1995.