Structural Extraction from Visual Layout of Documents

Binyamin Rosenfeld
ClearForest Corporation
15 East 26 St
New York, NY
212-432-1515
grur@clearfotrest.com

Ronen Feldman
ClearForest Corporation
15 East 26 St
New York, NY
212-432-1515
Department of Computer Science
Bar Ilan University
Ramat Gan, Israel
ronen@clearfotrest.com

Yonatan Aumann
ClearForest Corporation
15 East 26 St
New York, NY
212-432-1515
Department of Computer Science
Bar Ilan University
Ramat Gan, Israel
yonatan@clearfotrest.com

ABSTRACT

Most information extraction systems focus on the textual content of the documents. They treat documents as sequences of words, disregarding the physical and typographical layout of the information. While this strategy helps in focusing the extraction process on the key semantic content of the document, much valuable information can also be derived form the document physical appearance. Often, fonts, physical positioning and other graphical characteristics are used to provide additional context to the information. This information is lost with pure-text analysis. In this paper we describe a general procedure for structural extraction, which allows for automatic extraction of entities from the document based on their visual characteristics and relative position in the document layout. Our structural extraction procedure is a learning algorithm, which automatically generalizes from examples. The procedure is a general one, applicable to any document format with visual and typographical information. We also describe a specific implementation of the procedure to PDF documents, called PES (PDF Extraction System). PES works with PDF documents and is able to extract fields such as Author(s), Title, Date, etc. with very high accuracy.

Categories and Subject Descriptors

H.4. [**Information Systems Application**], H.3.3 [**Information Search and Retrieval**], H.3.1 [**Content Analysis and Indexing**], I.7.3 [**Index Generation**].

General Terms

Algorithms, Management, Experimentation, Documentation.

General Terms
Text Mining, Document Layout Analysis, Information Extraction

1. INTRODUCTION

Most text-processing systems [1] simplify the structure of the documents they process. The visual form and layout of the documents is ignored and only the text itself is processed, usually as linear sequences or even bags of words. This allows the algorithms to be simpler and cleaner, at the cost of a possible loss of valuable information. This paper is an attempt to restore the balance. We propose an approach that ignores the content of words, while focusing on their superficial features, such as size and position on the page. Such an approach is not aimed at replacing the semantic one, but rather to complement the conventional text extraction systems, and can also function as a preprocessor or a converter.

We implemented this approach in a system called PES (PDF Extraction System). The PES system accepts its input in the form of Acrobat PDF documents. A document page in PDF format is represented by a collection of primitive objects, which can be characters, simple graphic shapes, or embedded objects. Each primitive has properties, such as font size and type for characters, and position on the page, given as coordinates of the object's bounding rectangle. We are interested in an automatic process that accepts a formatted document as input, and returns a set of predefined set of elements of the document, each assigned to a corresponding field, e.g. "AUTHOR = …, TITLE = …, " etc. The set of field names and document elements that get assigned to them is problem-dependent, and may be different for different types of documents. Thus, we seek a system that learns how to extract the proper document elements based on examples provided by a domain expert. In PES system, described in this paper, a domain expert annotates a set of documents, marking the fields to be extracted. Each annotated document functions as a template, against which new documents are matched.

At the heart of the extraction system we have the following problem:

Given:

1. Document *A* (a *template*),
2. A set of primitives in *A* (annotated fields), denoted P_A,
3. Document *B* (a *query* document),

Find:

1. The degree of similarity between documents *A* and *B*.
2. The set of primitives in *B* that corresponds to P_A.

The first step in the process of finding the primitives of B that correspond to P_A, is to find similarities between the original document A and the new document B. The simplest way to match two documents is coordinate-wise: match objects of the that have identical bounding rectangles. The disadvantages of such an approach are obvious. If the same field has different visual sizes, e.g. a document title containing different number of words and text lines, or if the field is shifted a bit, the system will not identify the correct match. Nevertheless, the coordinates form a good basis for more refined heuristics, as the same fields tend to reside in more or less the same location across documents. However, the correspondence must be established between objects, not coordinates. In addition, the correspondence must be between higher-level groups and not only between primitive objects.

The PDF document representation does not contain any information about text lines, paragraphs, columns, tables, and other meaningful groups of primitives. The format is designed for human reading, where the human mind does the necessary grouping unconsciously. In order for information extraction systems to take advantage of the visual clues available in the PDF format, the system must perform perceptual grouping, as the first stage of processing a document. The approach we take is to take the physical/visual representation of the document, and transform it into a complex abstract representation – consisting of nested objects and relationships between them. We call this step *perceptual grouping*. Once perceptual grouping has been performed, the resulting structure is independent of the specific document type. This approach allows us to provide a general procedure applicable to diverse formats, and rapidly adaptable to new formats.

Once the document structures are generated, these structures can be used to extract information. A representative set of documents is annotated by a domain expert, with parts of the documents' structures being assigned to certain fields. These documents serve as templates to be matched against the new documents. In the process of structural mapping, a correspondence is created between two document structures, mapping the objects of a template document to the objects of the non-annotated query document.

In the following sections we describe our approach to the two problems outlined above, the perceptual grouping and the structural mapping. We then discuss a particular implementation suited for information extraction from PDF documents, and experimental results.

2. RACKS

The algorithms for both grouping and mapping use a special data structure, *the rack*, or *probabilistic priority queue*, for choosing objects to work upon. Racks are probabilistic queues, used in [7]. For both algorithms it makes sense that some of the objects are more important or more promising than others, and it is also possible to calculate their importance or promise level. Objects have weights that measure their importance and quality functions calculate their promise. A natural solution would be is to maintain a priority queue of objects, always choosing the best object to work upon. But this approach has two problems. First, if priority ranking of objects is very strict the algorithms may easily be thrown into a loop, constantly working on the same small set of objects and not making any progress. Secondly, even if the difference in priorities is very small the higher will always be chosen, which may cause "starvation" to some very promising objects. Racks provide the necessary solution to these problems, by chosening object randomly, rather than deterministically,but having the probability of choosing a particular object proportional to the object's priority.

Definition: A **rack** or **urgency queue** is a container data structure that supports two operations: *Add*(*Object*, *Urgency*) and *GetObject*(). The *GetObject*() returns a random object from among those added by *Add*(). The probability of choosing each object is proportional to its urgency.

Unique rack sums an object's urgencies if the object is added more than once.

Implementation: There is an efficient implementation of the rack data structure that allows adding and retrieving objects in *O*(log *N*), where *N* is the size of the container. The objects are stored in a balanced binary tree, and each node contains the sum of urgencies of all objects in a subtree rooted at the node. *Add*() inserts the object at the next available place and adds its urgency to all of the object's ancestors. *Get*() is more interesting:

```
Get(T : Tree) : returns object
    Let R := random number between 0 and T.Root.SumUrgencies
    S := GetAt(T, R);
    Return the object at S and erase the object from the tree; update ancestors' urgencies;

GetAt(T : Tree, R : number) : Tree
    if (T.LeftSubtree exists)
    if (T.LeftSubtree.SumUrgencies > R)
            return GetAt(T.LeftSubtree, R);
        end if
    R = R – T.LeftSubtree.SumUrgencies;
```

```
        end if
        if ($T.RightSubtree$ exists)
        if ($T.RightSubtree.SumUrgencies > R$)
                return $GetAt(T.RightSubtree, R)$;
            end if
        $R = R - T.RightSubtree.SumUrgencies$;
        end if
return $T$;
```

The unique rack is implemented in the same way using a red-black tree [4] as the storage structure, in order to allow rapid location of an object's location within the tree.

3. THE PERCEPTUAL GROUPING PROBLEM

An *Unstructured World Situation* is a set of primitive objects positioned in some coordinate system. The objects are primitive in the sense of having no internal structure, although they may be of different types and may have different properties. One property that is common to all object types is the "coordinates" property. The coordinate system allows defining a very crude relation between objects called *neighborhood*. In simple terms, two objects are neighbors if there is no other object between them. For coordinate systems of more than one dimension, the definition is somewhat more subtle , but can be based on the same idea. The neighborhood relation is the basis of all other semantic relations between objects.

The task of perceptual grouping is to discover meaningful groups – sets of neighboring objects that together define higher-level objects, which in turn participate in the neighborhood relation and may become parts of still higher-level objects. We define the notion of a hierarchical graph, or *H-Graph*, which mathematically captures the recursive building process of the neighborhood groups. Technically an H-graph is a forest of objects, with primitives for leaves and groups for internal nodes. Groups contain only objects that are connected by the neighborhood relation.

Formally, let P be a set of primitives and let $N \subseteq P \times P$ be a symmetric *neighboring relation.* An **H-Graph over (*P*,*N*)** is a rooted forest such that:

- At the leaves there are only elements of P.
- For any internal node x, if P_x denotes the set of all leaves in the subtree rooted in x, then $(P_x, N \cap (P_x \times P_x))$ is a connected graph.

In the following, we may omit reference to the sets P and N, if obvious from the context. We now provide some formal definitions and notations.

Definition: Let $H=(O, E)$ be an H-Graph over (P, N). For $x \in O$:

- ***Ground*(*x*)** denotes the set of all leaves in the subtree rooted at x.
- ***DescendantsTree*(*x*)** denotes the sub-tree rooted at x.
- ***Neighbors*(*x*)** denotes all nodes y, which do not belong to $Descendants(x)$, for which there exists a $p_1 \in Ground(x)$ and $p_2 \in Ground(y)$ such that p_1 and p_2 are neighbors (i.e. $(p_1, p_2) \in N$) .
- H-Graph H' is an **extension** of H if H is a sub-forest of H'.

The definition of an H-graph is not concerned with the types and properties of objects and groups. While these are important to the specific implementation, it is possible to define the general grouping problem and the generic algorithm without reference to them. In section 4, when we refer to the PDF extraction problem, we show an example of a specific H-Graph over PDF documentsthat is based on specific object types relevant to the PDF format.

Problem Definition:

The input is a set of primitive elements P and a neighboring relation N, together representing an unstructured world situation. Intuitively, the goal is to find an H-Graph over (P, N) that "gives structure" to this representation. There are many ways to structure the same set of primitives and clearly some of the ways are better than others, from the point of view of the problem domain.

In order to be able to compare different possible ways of structuring the same situation, we assign a *quality score* to each H-graph, as follows. We assume that we are provided a value function $Q : \text{Groups}(P) \rightarrow \mathbf{R}$, which assigns a score to each group that can be built over P. Formally, Groups(P) is the set of all (up to isomorphism) trees, such that their set of leaves is a subset of P connected by N. Then, for an H-Graph H, we define:

$$Q(H) = \sum_{x \in H} Q(DescendantsTree(x)),$$

the sum of scores of all groups in H. Given the quality function Q, the problem of perceptual grouping becomes a problem of combinatorial optimization: find an H-Graph, H_{opt}, over (P, N), with the highest quality score $Q(H_{opt})$.

Naturally, the tractability of the problem depends on the properties of the neighborhood relation, N, and the quality function, Q. In most cases the right solution is obvious, and the quality function reflects this fact by giving it a much higher score than its rivals. However, there are some difficult or ambiguous cases where the right solution (from the standpoint of the domain problem) may not be found without additional information or further processing at stages other than the perceptual grouping. For those cases it is better if the system does not try to find the maximal solution, but instead choose randomly from the cluster of solutions of more-or-less the same score. Then if the solution is inadequate, the grouping may be rerun with hopefully better results.

The algorithm we present is a greedy algorithm, but it allows retracting choices at specific points if better alternatives come up. Rather then using simple backtracking, all changes to the structure are local and preserve whatever was done at other parts of the object collection. The algorithm includes a random element, so although the obvious groups are always built in the same way, the difficult ones could be rebuilt differently in each run. The algorithm, being greedy, assumes what can be called *accessibility* of the quality function, meaning that high-score groups can be built by gradual addition of objects, one by one, without significantly decreasing the score on the way. This condition does not always hold in real world situations. For instance, when building a table, it is necessary to add a whole row of objects at once in order not to decrease the quality. This is not a serious limitation, as the algorithm can be altered to check for 'minimal extensions' instead of 'extensions by one object'.

Grouping Algorithm:

Input: ***P*** (set of primitives)**,** ***N*** (Neighborhood relationship)**,** ***Q*** (quality function) **.**

Output: ***H*** (H-Graph over (P,N))

```
ObjectsToProcess := P;
while (ObjectsToProcess is nonempty)
    x := some object from ObjectsToProcess.
    if ($x \in H \setminus P$) // x is a group - try to extend it
        y := FindFittingExtension(x);
        if ($y \neq \emptyset$)
            ExtendGroup(x, y);
            continue;   // process next object
        end if
    else
  // x is primitive or x is a group but cannot be extended
        y := FindFittingParent(x);
        if ($y \neq \emptyset$)
            CreateGroup(y);
            continue;   // process next object
        end if
    end if
end while
```

The main loop of the algorithm revolves around the *ObjectsToProcess* rack. *ObjectsToProcess* initially contains every primitive, and in each iteration one object is taken off the rack and processed. If the object is a previously built group, an attempt is made to extend the group by adding one more object from its neighborhood. If the group cannot be extended, or if the object is primitive, an attempt is made to create a higher-level group containing the object.

An object y is *fit to extend a group x* if, (i) they are neighbors, (ii) the quality of x+y is better than the quality of x, and (iii) the ***Fitness*** of y in x is better than the *Fitness* of y in its previous parent group (if it exists). The *Fitness* function is defined via the quality function Q:

$$Fitness(y,x) = \begin{cases} 0, \text{if } y \notin Children(x) \\ Q(x) - Q(x-y), \text{if } Q(x-y) > 0 \\ Q(x) / |Children(x)|, \text{otherwise.} \end{cases}$$

ExtendGroup(x, y) disconnects both x and y from their previous parents, adds y to the children of x and places the new x+y object onto the *ObjectsToProcess* rack, allowing it to be further extended or to be used as a base for a higher-level group.

A node y is a *fitting parent of node x* if: (i) y has one or two children, one of which is x and another x's neighbor, (ii) the quality of y is positive, and (iii) the *Fitness* of all y's children in y is better than their *Fitness* in their previous parents.

CreateGroup(y) disconnects y's children from their previous parents, adds y to H, and places y onto the *ObjectsToProcess* rack.

DisconnectFromParent(x) checks whether $p = Parent(x)$ exists, and if it does, whether it can exist without x (it cannot if $Q(p - x) \leq 0$). If it can, x is removed from the children of p, and p is placed onto the *ObjectsToProcess* rack. Otherwise, p is destroyed (removed from the current H) and all its former children are placed onto the rack.

The main loop is executed so long as there are objects in the queue. The execution stops when all attempts to either extend any group or create a new one fail. Precautions may be taken to stop the loop after a fixed number of iterations, in order not to fall into an infinite loop because of a bad quality function. Our experience that it is usually safe to stop the processing after $5|P|$ iterations, since the mode of execution ensures that the most important and the best groups are constructed first.

4. STRUCTURAL MAPPING PROBLEM

Once we have a method to convert an unstructured object set into a structured H-Graph, the next goal is to map between two structured H-Graphs: that of the template document and that of the query document. We require that the mapping preserve the hierarchical structure of the two H-Graphs. Specifically, if a node x is under node y, then $Map(x)$ must be under $Map(y)$. We perform the mapping in a recursive fashion, mapping the highest-level objects to highest-level objects, and then recursively mapping their members. Thus, it remains to show how to perform one level of the mapping. On each level, there is a set of objects in each of the graphs, and neighboring relation defined between objects of each graph.

The problem definition:

The input is two undirected graphs, with nodes for objects and edges for the neighborhood relation between them. The desired output is a function that best maps one graph onto another. In order to define the quality of a mapping, two problem-dependent *similarity functions* are required, evaluating similarity of any two objects and of any two relations. The definition of the algorithm in terms of those functions allows avoiding the types and properties of objects and relations. There may also be a different weight assigned to each object and relation, if they have different importance. In this case, the problem is to optimize the weighted quality of the mapping.

Problem: Given two undirected connected graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, a weight function $W : V_1 \cup E_1 \rightarrow \mathbf{R}+$ on vertices and edges of the first graph, and two similarity functions, *SimObj* $: V_1 \times V_2 \rightarrow [0,1]$ and *SimRl* $: E_1 \times E_2 \rightarrow [0,1]$, find a mapping $M : V_1 \rightarrow V_2$, that maximizes *MapQuality,* where

$$MapQuality(M) = \sum_{v_1 \in V_1} w(v_1) \cdot SimObj(v_1, M(v_1)) \cdot RlQuality(v_1, M(v_1), M)$$

$$RlQuality(v_1, v_2, M) = \sum_{(v_1, u_1) \in E_1} w(v_1, u_1) \cdot SimObj(u_1, M(u_1)) \cdot SimRl((v_1, u_1), (v_2, M(u_1)))$$

(Note: If $(v2, u2) \notin E2$, $SimRl(*, (v2, u2))$ is considered to be zero.)

The algorithm we use to solve this problem is also a greedy algorithm. The algorithm requires a seed mapping to start working. A seed is a map of a single object. It doesn't have to be the correct or the best mapping, but if it is too far off, the algorithm will be slower and might not find a good mapping if the neighborhood relation is too sparse. In the specific problem of PDF document structures mapping, we take as the seed a mapping between two top-most and left-most objects that can be mapped onto each other.

The Mapping Algorithm

Input: G_1 (template graph), G_2 (query document graph), W (weigh function), *SimObj* (object similarity function), *SimRl* (relation similarity function).

Output: M (mapping function))

```
$M(Seed_1) := Seed_2$ for some $Seed_1 \in V_1$ and $Seed_2 \in V_2$;
$ObjectsToProcess := \{ Seed_1 \}$;
while ($ObjectsToProcess$ is nonempty)
$v_1 :=$ some object from $ObjectsToProcess$;
for all $(v_1, u_1) \in E_1$
        find $(M(v_1), u_2) \in E_2$
            such that $RlQuality(u_1, u_2)$ is maximal;
        if ($M^{-1}(u_2) = \varnothing$ or $RlQuality(M^{-1}(u_2), u_2)$
            < $RelQuality(u_1, u_2)$ )
                $M^{-1}(u_2) := \varnothing$;
                $M^{-1}(u_1) := u_2$;
                $ObjectsToProcess := ObjectsToProcess \cup$
                        $\{ u_1 \}$;
                end if
        end for
end while
```

In a similar way to the grouping algorithm, the main loop is carried over the rack *ObjectsToProcess*. All objects in this rack are already mapped somewhere. At each iteration one object is taken off the rack and processed. An extension of the mapping is attempted for all neighbors of the object. All objects that were successfully mapped are placed in *ObjectsToProcess*. The process stops when there are no more objects to map.

5. STRUCTURAL EXTRACTION (IMPLEMENTATION FOR PDF)

The PES system contains several components, which instantiate for PDF documents the generic grouping and mapping algorithms described above. The specific instantiation defines the neighborhood relation and provides the various quality functions.

The exact definition of neighborhood can vary as long as it is consistent and provides good connectivity between adjacent objects. The definition chosen for our implementation considers two objects to be neighbors if no other object obscures them from each other's "view".

Grouping of PDF objects

There are three kinds of primitive objects, namely characters, simple graphics, and embedded pictures, and there could be many kinds of groups, of which three are currently implemented: 'text lines', 'paragraphs' and 'columns'. The qualities of the different kinds of groups are calculated differently.

Text lines are the simplest to spot. They are sequences of neighbor characters, whose vertical coordinates are more or less the same. The distinction between lines and non-lines is sharp enough for the definition to be exact (non-fuzzy). A group of neighborhood-connected characters is considered to be a line (with quality = 1.0) if their y-coordinates are not further apart than half their average height, and their x-coordinates are not too far from each other, so the x-distance between the bounding rectangles of any two adjacent characters is less than twice their average height. Any other group is considered to be unacceptable as a line (quality = 0).

'*Paragraphs'* are groups of neighboring lines that are bound to each other more closely than to anything else. A good paragraph has the following characteristics: the characters in different lines are of the same font; the distances between lines are constant; and the lines are aligned or centered. The exact scoring function for paragraphs is provided in the Appendix.

Columns are groups of neighboring paragraphs, residing vertically, with difference in font size less than twice minimal font size, and with the distance between paragraphs less than four times the average line height. The quality of an acceptable column is 1, and everything else 0.

The quality of paragraphs and columns includes a small positive bonus for the size. If the group contains only a single element, the quality is halved. Thus, bigger groups are preferred, and singleton groups will form only in the lack of any better option.

Mapping of PDF structures

In order to instantiate the generic algorithm to the specific instance of PDF extraction, we need to define the two similarity functions *SimObj*(*Obj*$_1$,*Obj*$_2$) and *SimRl*(*Rel*$_1$,*Rel*$_2$).

SimObj(*Obj*$_1$,*Obj*$_2$) measures the similarity between two objects. Our implementation of the function takes into account the following three parameters:

- Object types: only objects of the same type get a positive similarity value,
- Size: the closer in size the objects are to each other, the higher the score,
- Position: the closer the objects would be to each other, if placed on the same page, the higher the score.

SimRl(*Rel*$_1$,*Rel*$_2$) measures the similarity between two relations, i.e. pairs of objects. Our implementation of the function disregards all relations between two objects except for relations between positions of their bounding rectangles. Specifically, Rel_1=(Obj_{11},Obj_{12}) is considered similar to Rel_2 = (Obj_{21},Obj_{22}) if the corners of Obj_{12} lay in the same relations to Obj_{11} as the corners of Obj_{22} lay to Obj_{21}. A detailed description of the functions used in PES is provided in the Appendix.

Example

Figure 1 shows two portions of two different PDF documents (both Lehman Brothers research reports). Each primitive object is annotated with a number. These are the numbers 1 through 7. We ran our Grouping algorithm on both sets of objects, producing objects of higher order. These are the objects 8-11, indicated in Figure 1.

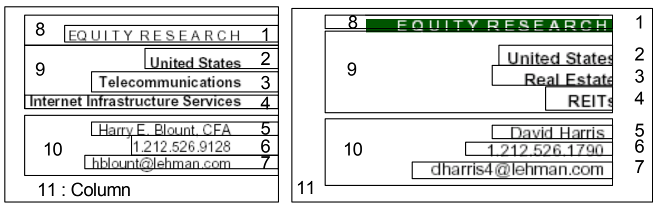

Figure 1 - two portions of 2 different PDF documents A (left) and B(Right)

Figure 2 depicts the sequence of steps taken by the Grouping algorithm, for the two PDF portions; in forming the H-Graph (both portions produced the same H-Graph). The final H-Graph is shown in Figure 3.

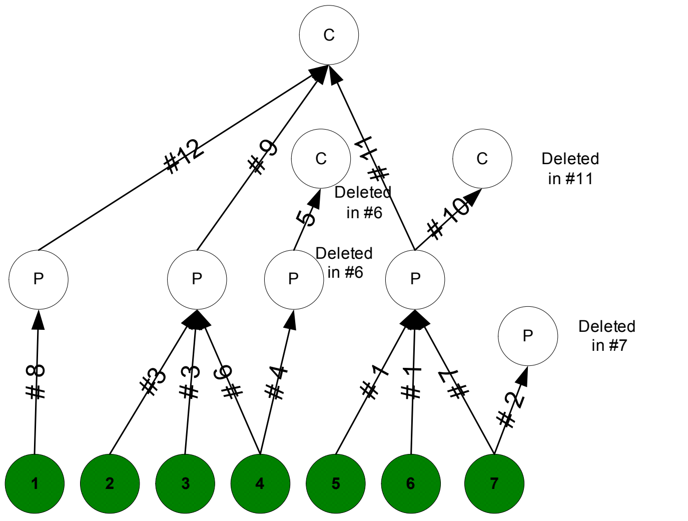

Figure 2 - The Grouping algorithm operating on each of the PDF portions of Figure 1

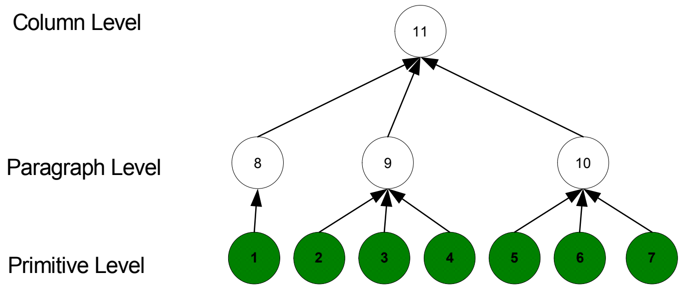

Figure 3 - The Final H-Graph of the PDF portions

The Mapping Algorithm

Once the H-Graphs for both PDF portions where constructed, we applied the Mapping algorithm, to map the two structures. Note that while the resulting H-Graphs were identical, the objects themselves are different (different font, location, etc.). Thus, there is still a need to find the best mapping.

The mapping algorithm in this case runs in three levels, according to the three levels of the associated H-Graphs. Following is a trace of the operations of the algorithm (we denote by A.*i*, and B.*i*, the *i*-th object of PDF A and B, respectively).

Level 1. *Mapping of the roots.* In the given example, there is only single root in each H-Graph. Both are of the same type (column) and thus can be mapped.

M(Object A.11) := Object B.11

Level2. *Mapping of the roots' children.* The topmost and leftmost compatible objects are matched. The seed is placed in *ObjectsToProcess*.

M(A.8) := B.8

Loop: Already mapped object is taken from *ObjectsToProcess* and its neighbors are mapped and placed into *ObjectsToProcess*.

Neighbors(A.8) = { A.9 }.

M(A.9) := B.9

Neighbors(A.9) = { A.8, A.10 }.

M(A.8) is unchanged

M(A.10) := B.10

Neighbors(A.10) = { A.9 }.

M(A.9) is unchanged

Level3. *Mapping of the children of the matched paragraphs.*

Children(A.8) = { A.1 }
Children(B.9) = { B.1 }
M(A.1) := B.1

Children(A.9) = { A.2, A.3, A.4 }

Children(2.9) = { B.2, B.3, B.4 }

M(A.2) := B.2

M(A.3) := B.3

M(A.4) := B.4

Children(A.10) = { A.5, A.6, A.7 }

Children(B.10) = { B.5, B.6, B.7 }

M(A.5) := B.5

M(A.6) := B.6

M(A.7) := B.7

At this point we have a perfect match between the two H-graphs, and the mapping is complete.

The overall system architecture

The system contains several components: Annotator, Grouper, Mapper, and Extractor. Annotator is a GUI tool that allows the user to mark fields in a PDF document and store their names and positions in a separate file. Grouper takes a PDF document as input, does the grouping and saves the document structure. Mapper's input is a template (document structure + fields data) and a document structure for a query document. The template is mapped onto the query document, and the elements assigned to the various fields are produced as output, together with the overall quality of the mapping. Extractor takes a document structure and the selected elements and outputs the elements' text.

The Grouper and the Mapper may sometimes produce different results for the same document(s), because the algorithms have a random element to them. This feature can actually be an advantage and can be used to improve the precision of the results, as follows. The Mapper outputs the mapped elements for a new document. This allows the document to be used as a template, so the Mapper can map it back to the original template, producing alternative elements. If the elements are different from the original elements, a different grouping may be tried. Experiments suggest that this technique (called 'MapBack') does significantly improve precision. The architecture of the PES system is show in Figure 4.

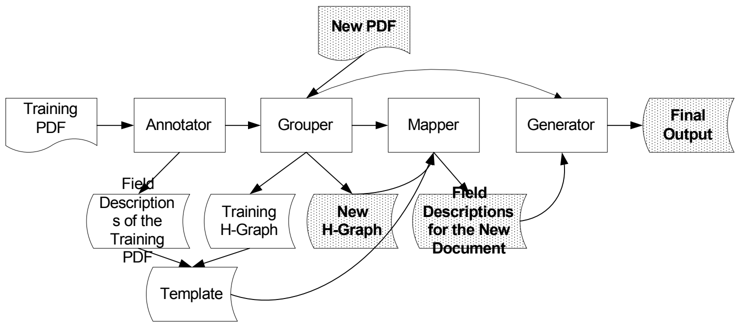

Figure 4 - Architecture of the PES System

6. EXPERIMENTAL EVALUATION

The experiments were performed using a dataset of 500 annotated company reports, 100 from each of the five brokerage firms: Lehman Brothers, Merrill Lynch, Morgan Stanley Dean Witter, SalomonSmithBarney and HSBC. For each of the documents, 4 fields were marked: Author (first author), Date, Title, and Document Source. The experiments were conducted to find out how the results depend on the number of template documents and on the use of randomness and 'mapping back and regrouping' technique. The results are summarized in the Table 1. Each entry contains the recall and the precision (recall/precision) for each of the 4 elements (Author, Date, Title and Source) and their average values.

Table 1. Recall/Precision figures for experiments done with PES

Map Back?	#temp -lates	Author	Date	Title	Source	All Fields
No	1	40.0/58.0	48.0/72.8	47.3/67.0	70.0/70.1	51.3/67.0
No	3	50.0/67.3	45.3/70.7	46.0/52.7	67.3/70.7	52.2/65.4
No	6	65.3/86.4	84.0/94.4	77.3/83.7	82.0/91.9	77.2/89.1
No	10	66.0/87.0	83.3/91.9	83.3/89.9	78.0/93.5	77.7/90.6
Yes	1	52.7/70.1	46.7/74.7	46.0/68.7	68.7/99.3	53.5/78.2
Yes	3	58.0/65.9	46.7/72.7	46.7/68.1	70.0/88.2	55.3/73.7
Yes	6	90.7/90.7	90.7/91.8	83.3/83.8	94.7/94.7	89.8/90.3
Yes	10	93.3/93.3	94.0/94.0	94.7/94.7	96.7/96.7	94.7/94.7

Figure 5 shows the dependence of the overall performance on the number of training documents (templates) per document type, for both the regular and MapBack approaches.

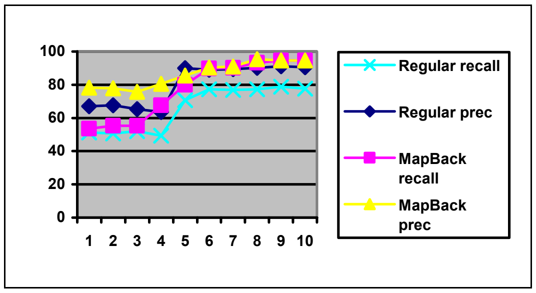

Figure 5 - Performance per number of templates

Several things are worth noticing in this chart. First, both precision and recall of the algorithm with MapBack are significantly better than their regular counterparts. Also, note that the MapBack curves show more stable improvement as more templates are added, in contrast to the regular experiments, where adding a template sometimes makes results worse. This shows that the MapBack technique can filter out bad templates. , Finally, all curves show a jump at 4-5 templates. The reason is that the templates were chosen randomly. Thus, the document types (formats) are not represented equally or uniformly among the templates. Apparently, the fifth template happened to belong to a format that was not previously represented.

We also compared the performance of our algorithm to a fully greedy algorithm. The difference is that our algorithm allows reshuffling of objects between groups, while a greedy algorithm would not. The results are shown in Figure 6.

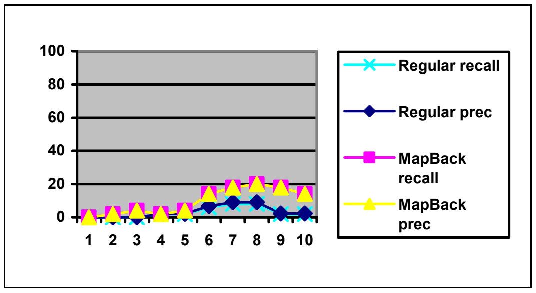

Figure 6 - Performance of fully greedy algorithm

As can be seen, the results are very bad, which shows that for the perceptual grouping problem our algorithm is indeed stronger than purely greedy algorithm.

7. CONCLUSIONS

The experiments suggest that the proposed approach is indeed viable, although not without shortcomings. The significant improvement of the precision with MapBack tests shows that the feedback between grouping and mapping is important. The technique can be thought of as a way to influence the grouping process with the results of the mapping process, a way to blend them together, however weakly. The improvement of the performance confirms the point of view of [7] that the two basic problems, perception and analogy making, are interdependent, even in such a limited domain as structuring a PDF document. Further research should focus on blending them altogether, within the same urgency-based architecture.

Another essential shortcoming is the complexity of group quality functions. It is interesting to note in contrast that the similarity functions are very simple and still adequate. This suggests that another level of abstraction may be required, between the quality functions and the types of groups they represent, abstraction that allows the group quality to be calculated from a set of features of the group's components and the group as a whole.

The method of learning new document types is close to learning concepts from prototypical examples. It is assumed that simpler concepts, which define parts of the document type, are already learned and can be found and identified by the process of perceptual grouping. So, what is learned is the particular configuration of simpler concepts that make a complex concept, or a set of such configurations. Yet another direction of further research would be to allow some of the intermediate concepts, less than the whole document, to be learned in this way. Other researchers [2,3,5,8] have dealt with modeling the document structure. However the main purpose of their systems was to generate the structure of the document and not to extract specific elements. In addition these systems have no learning component that is able to learn the extraction patterns based on a set of training examples.

8. REFERENCES

[1] Appelt D. E., Hobbs J., Bear J., Israel D. and Tyson M., 1993. "FASTUS: A Finite-State Processor for Information Extraction from Real-World Text", Proceedings. IJCAI-93, Chambery, France, August 1993.

[2] Cattoni R., Coianiz T., Messelodi S., and Modena C. M. ITC-IRST, Via Sommarive, I-38050 Povo, Trento, Italy January 1998 .

[3] Cerny, V., "Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm", J. Opt. Theory Appl., 45, 1, 41-51, 1985

[4] Cormen Thomas H., (Editor), Leiserson, Charles E., and Rivest, Ronald L. "Introduction to Algorithms", Second Edition. MIT Press, September 2001.

[5] Dori D., Doermann D., Shin, C., Haralick R., Phillips I., Buchman M., and Ross D. The Representation of Document Structure: A Generic Object-Process. Handbook on Optical Character Recognition and Document Image Analysis, World Scientific Publishing Company, 1996

[6] Feldman R., Rosenfeld B., Stoppi J., Liberzon Y. and Schler, J., 2000. "A Framework for Specifying Explicit Bias for Revision of Approximate Information Extraction Rules". KDD 2000: 189-199.

[7] Hofstadter, D., Fluid Concepts & Creative Analogies : Computer Models of the Fundamental Mechanisms of Thought, Basic Books, 1995.

[8] Kirkpatrick, S., C. D. Gelatt Jr., M. P. Vecchi, "Optimization by Simulated Annealing",Science, 220, 4598, 671-680, 1983.