IJDAR (2002) 5: 17–27

International Journal on IJDAR
Document Analysis and Recognition

A hierarchical representation of form documents for identification and retrieval

Pınar Duygulu, Volkan Atalay

Department of Computer Engineering, Middle East Technical University, Ankara, 06531 Turkey;
e-mail: {duygulu, volkan}@ceng.metu.edu.tr

Received: August 21, 2001 / Accepted: November 5, 2001

Abstract. In this paper, we present a logical representation for form documents to be used for identification and retrieval. A hierarchical structure is proposed to represent the structure of a form by using lines and the XY-tree approach. The approach is top-down and no domain knowledge such as the preprinted data or filled-in data is used. Geometrical modifications and slight variations are handled by this representation. Logically identical forms are associated to the same or similar hierarchical structure. Identification and the retrieval of similar forms are performed by computing the edit distances between the generated trees.

Keywords: Form document processing – Logical layout extraction – Retrieval – Data processing

1 Introduction

With recent advances in multimedia technology, various types of data become available to be processed by computers as input to information systems. However, most of the time, data has to be manipulated and extracted manually. One of the challenges is to automate this information extraction process from the raw data. Paper is still one the most commonly used mediums for transferring information. Among paper documents, *forms* are structured documents used for information gathering, storage, retrieval, approval, and distribution which are very important processes in private and government business. Traditional manual key entering of data on forms is tedious, time-consuming, and error-prone. Consequently, it is highly desirable to automatize this task as much as possible.

There are several challenging problems in form processing such as layout extraction, retrieval, processing heterogeneous batches, etc. For example, the last problem requires identifying the particular type of input form, registering the input form with a reference model form, extracting information from the input form, and storing the extracted information.

In this section, we first give some of the basic definitions which are used in the frame of form document processing and in this study.

Definition: A *form* is a structured document which is composed of the following elements:

- horizontal and vertical layout lines: straight and continuous;
- preprinted data: machine printed characters, symbols, and pictures;
- user filled-in data: machine typed, hand-printed or handwritten characters.

Preprinted data and user filled-in data are located at predefined locations which are called *fields*. A field is enclosed by a rectangular box formed by two horizontal and two vertical lines. Fields frequently have interrelationships where the data entered in one field validates the data entered in another field. Thus, on a form the information is kept in tuples, the first element being preprinted and the second user filled-in.

The main interest in form processing systems is to extract user filled-in data and associate it with the corresponding preprinted field. However, in order to perform such a task, the structure of the form should be known in advance. The form structure can be obtained by processing a *reference model form*. A reference model form can be, for example, a blank form on which no user filled-in data exists. The reference model form can be used to specify the fields on the form. The reference model form may be stored in a form database and when an *input instance form* is presented to the system, its type can be identified by matching with one of the reference model forms in the database. Required data can then be automatically extracted following the specification given for the reference model form.

To store a reference model form in the database and also to match an input instance form with a stored reference model form, we need a representation. The most straightforward way to represent a form document is by its direct image. However, such physical information may not be appropriate, since the form or its image risks being modified geometrically (enlarged/shrunk, translated/rotated, etc.) or being distorted due to printing or digitization. On the other hand, the logical structure represents the semantics of the form and the same logical structure can be formatted in a variety of physical layouts. The geometrical structure should be mapped to a logical structure by considering the logical relations. Even multi-kinds of the same application-specific forms (logically the same but physically different forms) can then be represented by the same logical structure.

In this paper, we present a representation for the structure of form documents to be used for identification and retrieval purposes. We describe a heuristic algorithm based on the XY-tree method [1,2] in order to transform the geometric structure of a form document into a hierarchical structure by using horizontal and vertical layout lines which exist on the form. The structure of a form document is represented by a tree. The hierarchy of the tree corresponds to the hierarchy of the blocks in the form document. The proposed representation is close to the human point of view for the form document structure. Logically identical forms are expected to have the same or similar hierarchical tree structures. In addition, geometrical modifications and slight variations on a form are handled by the described representation. Identification and the retrieval of similar forms are performed by computing the edit distances between the generated trees.

2 Related work

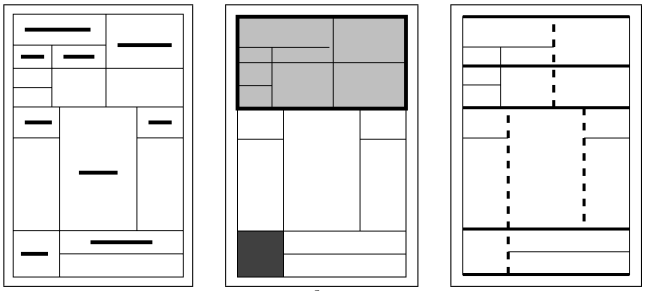
Rather than its direct image, a form document can be represented by the physical features of its components [15,17,16,18–26]. For example, length, width, and position of horizontal and vertical lines can be used [15]. However, this cannot handle variations on the physical structure of the logically same form. Another feature related to the horizontal-vertical lines are line crossings. Extracted line crossings on a given form can be classified into one of various types and the form can be represented by the set of type counts [16]. However, this approach will not work even for slight variations, because the types and counts of line crossings are prone to change. On a form document, a block is defined as a rectangular area which is surrounded by horizontal and vertical lines. In this sense, it is a higher-level feature than the lines. For form representation, the position and size of blocks or their relationships can be used [17–20,22,24]. However, multi-kinds of forms cannot be handled by such a scheme. In a similar way, relations of the blocks which share lines can be employed [19]. However, in such a scheme, hierarchical relationships of block groups are not represented. There are also studies in which the main aim is to classify the form documents [24,23]. Liu et.al. [21] describe a method using a very similar approach to ours in the sense that the horizontal and vertical lines are used to extract the blocks of a given form. However, they do not take into account the relationship among the blocks. Furthermore, they do not give a representation scheme for the extracted blocks such as a tree or a list. A more general method is to use a bottom-up approach to form blocks. In the work of Watanabe *et al.* [20], three binary trees are constructed. Two of them are for the global and local structure. The last one is for classification whose construction is performed through a block division process which is not given in detail in the paper. Registration is performed by searching the classification tree. In addition to the horizontal and vertical lines, some of the preprinted data are also used. Moreover, the described method is mostly for table-form documents. In the literature, the method is also found to be hard to apply to the analysis of filled-in forms, because it is considered to be limited to empty fields [18]. Another recent study by Ishitani [7] describes a hierarchical matching strategy based on sub-graph matching which consists of global matching stages by sub-form matching, local matching stages by line matching, and the interactions between them. Ishitani uses an association graph whose arc represents compatible correspondence between lines or between sub-blocks and an algorithm to obtain the best correspondence by searching for the largest clique in the association graph. The method seems to be rather theoretical though heuristics have been used for the solution of some of the problems and the proposed similarity measure totally depends on the number of vertical and horizontal lines. Furthermore, there is no explicit representation of a form document. A form document is directly represented by its lines rather than having a more abstract representation. Dengel [8] describes an algorithm which establishes weighted syntactic representations from detected layout features using position and dimension.

3 Definitions and approach

The physical structure of a form consists of horizontal and vertical layout lines in addition to the preprinted and filled-in data. We propose a hierarchical structure to represent the logical layout of a form by using lines which are considered as the separators among the subblocks of a form. First, some definitions are given and then we resume the approach.

3.1 Definitions

A *block* is a rectangular area on the form which is surrounded by the longest horizontal and vertical lines at any given instant. For example, the biggest block of a form is the form itself. A *cell* is defined as the smallest block which only consists of a block frame. A *block frame* is the horizontal and vertical lines that surround a block and the horizontal lines which have the same length as the borders of a block frame are defined as the *horizontal separator lines. Vertical reference lines* are the lines which are orthogonal to the horizontal separator lines, that start at any horizontal separator line and end at another horizontal separator line. For the overlapping vertical reference lines, the one with the maximum length



a **b** **c**

Fig. 1. An example form. **a** Thick black lines show the preprinted areas, graphics or images. **b** An example block is shown by the light gray area at the top along with the thick black lines which indicate the block frame. A cell is demonstrated with the dark gray area at the bottom left. **c** For the form, horizontal separator lines and vertical reference lines are given by the thick black lines and by the dashed lines, respectively

is taken as the vertical reference line. A block frame is defined to be *ambiguous*, if it includes only the lines of length equal to the borders of the block frame.

In order to visualize the execution of the algorithm, we proceed with the form shown in Fig. 1a where the thick black lines show the preprinted areas, graphics or images on the form. These areas are shown only for the visualization of the form structure. Notice that preprinted data is not used in the algorithm. In Fig. 1b, the light gray area shows an example block, the thick black lines show the block frame for this block, and the dark gray area at the bottom shows a cell. In Fig. 1c, horizontal separator lines for the form itself are shown by thick black lines and vertical reference lines are shown by dashed lines.

3.2 Approach

The main aim is to partition the form into blocks which can further be divided until cells are reached. The partitioning results in a tree where the root node is the form itself and the leaf nodes correspond to the cells. The heuristic behind the approach is that the blocks which contain similar information are grouped together and these group of blocks are separated by lines which are relatively longer than the others. Such lines by definition are called horizontal separator lines. Thus, the information about how to partition a block is given by the horizontal separator lines. However, not all horizontal separator lines provide this information, but the vertical reference lines give clues about which horizontal separator lines are separators.

In order to achieve a sequence of block partitioning, a switching of horizontal and vertical divisions is used. This idea, known as X-Y trees, has been proposed by Nagy [1]. In document analysis, the XY-tree is a popular decomposition method for page and layout analysis. In this method, the page is recursively split into rectangular blocks by alternating horizontal and vertical cuts along spaces [1] or lines [2]. The result of such a tree is represented by an XY-tree in which the root node corresponds to the whole page and the leaf nodes represent blocks of the page. Each level in the XY-tree alternatively represents the results of the horizontal and vertical segmentation. We use the XY-tree method along with the switching lines to partition a form document into its blocks. This exploits all of the information inherited by the logical structure of the form via horizontal and vertical lines. At a step, since only the information supplied by a certain type of line is exploited for further partitioning at the next step, the information provided by the other types of lines is used.

Using the given definitions, the proposed approach is implemented by an algorithm to extract a hierarchical representation of a given form document. The approach is top-down and no domain knowledge such as the preprinted data or filled-in data is used. At the end, two forms which have different physical structures but the same logical structure are associated to the same representation.

4 The heuristic algorithm

The algorithm consists of three phases : initialization, block finding (BlockFinding), and tree construction (Tree Construction).

```
ALGORITHM
{
  /* -- first phase-initialization -- */
  /* current block is initialized to be the form */
  block_curr = Form

  /* initial processing direction */
  block_curr.direction = horizontal

  /* queue to store the extracted blocks */
  BlockQ = CreateQueue

  /* -- second phase -- */
  /* partition the current block frame */
  BlockFinding(BlockQ, block_curr)

  /* -- third phase -- */
  /* insert the extracted blocks to the queue */
  TreeConstruction(BlockQ, block_curr)
}
```

The first phase is used to initialize the current block frame (block_curr) to the form frame and also to create a queue for the blocks to be processed. Partitioning of the current block frame is performed at the second phase. Extracted horizontal separator lines for the current block frame are processed and a block is always defined between the first-in terms of position-horizontal separator line (s_curr) and the corresponding ending horizontal separator line (s_end). The ending horizontal separator line is sought through vertical reference lines that intersect with the other horizontal separator lines. The partitioned blocks are stored in a queue data structure to be inserted into the tree and in order to be further divided into smaller blocks, if possible. The processing order of the extracted blocks is maintained by the queue. We give definitions of the terms that are used in the algorithm as follows:

```
DEFINITIONS
  block_curr : current block
  S = s[i], i=1,...,n : extracted horizontal
      separator lines for block_curr
  n : number of horizontal separator lines
      for block_curr
  s_curr : first horizontal separator line
      in S which has not been processed yet
  s_end : ending horizontal separator line
  R = r[i], i=1,...,m : vertical reference lines
      starting at s_curr
  R'= r'[i],i=1,...,k : vertical reference lines
      starting at horizontal separator
      lines succeeding s_curr
  m : number of vertical reference lines
      for s_curr
  k : number of vertical reference lines for
      the succeeding horizontal separator lines
  BlockQ : queue of blocks
```

Defining a block, particularly determining the ending horizontal separator line, is not a very straightforward process. A basic algorithm for BlockFinding is given as follows:

```
BlockFinding(BlockQ, block_curr)
{
  /* horizontal separator lines of the
     current block */
  Construct S
  i = 1
  /* go through all of the horizontal
     separator lines */
  While (i<n) Do
  {
    s_curr = s[i]
    /* horizontal separator line at which
       the block terminates */
    s_end  = FindEndingHSL(s_curr)

    /* block instance to be enqueued */
    block = DefineBlock(s_curr,s_end)

    /* enqueue the block for eventual
       decomposition */
    AddQueue(BlockQ,block)

    /* if there are any skipped horizontal
       separator lines during the extraction
       of a block, do not consider them for
       processing */
    While (i<ord(s_end)) Do
     i = i+1
  }
}
```

Initially, the horizontal separator lines are found for the given block. The process proceeds until all of these horizontal separator lines are considered. The currently processed horizontal separator line is assigned to be s_curr which means that it indicates the beginning of the current block. Then, the corresponding ending horizontal separator line is determined by FindEndingHSL(). A block is simply defined by s_curr and s_end and the defined block is inserted into the queue to be inserted into the tree and for further processing, if possible. Then the horizontal separator lines in-between the beginning one (s_curr) and the ending one (s_end) are skipped and the process continues if there are any more horizontal separator lines.

After finding the blocks, each one is inserted into the tree which will represent the logical structure of the form. The TreeConstruction procedure performs the tree insertion operation in addition to the process of the whole algorithm until the cells are reached. If the processing continues, than the direction is switched.

```
TreeConstruction(BlockQ, block_curr)
{
  /* till all of blocks is processed */
  While Not EmptyQueue(BlockQ)
  {
    /* process the subsequent extracted block */
    block = DeQueue(BlockQ)

    /* insert the block to the tree */
    InsertTree(block_curr, block)

    /* if the current block is not a cell
       then continue */
    If Not Cell(block) Then
     {
       /* for subsequent processing
          change direction */
       block_curr.direction =
                  NOT(block_curr.direction)

       BlockFinding(BlockQ, block)

       /* recursive call */
       TreeConstruction(BlockQ, block)
     }
  }
}
```

4.1 A basic algorithm for finding the ending horizontal separator line

A basic algorithm to determine the ending horizontal separator line for a block is given below and is explained in this section. If there are only borders of the block frame orthogonal to the horizontal separator lines (i.e., m=2), then each horizontal separator line defines a block. Otherwise, the shortest vertical reference line that starts at the current horizontal separator line defines the ending horizontal separator line.

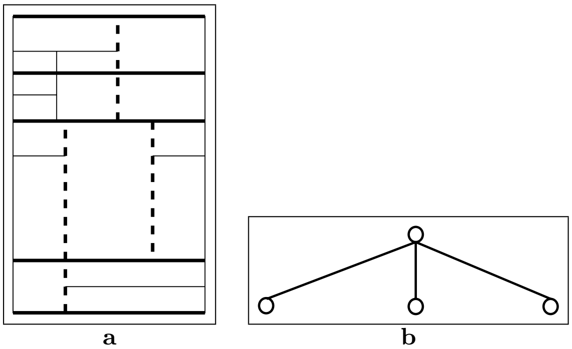


Fig. 2. a Initial horizontal separator lines (drawn as thick black lines) and vertical reference lines (drawn as dashed lines) of the form shown in Fig. 1a. **b** Initial tree when the current frame is the form frame

```
FindEndingBaseline(b_curr)
{
  /* if there are only borders of the
     block frame orthogonal to the
     horizontal separator lines then
     next horizontal separator line
     is the ending line */

  If (m=2) Then
     s_end = s_curr.next

  /* otherwise, the end of the shortest
     vertical reference line that starts
     at the current horizontal separator
     line defines the ending line */
  Else
   {
     v_shortest = FindShortestVRL
     s_end = v_shortest.end
   }
  return s_end
}
```

4.2 An example for the basic algorithm

In order to visualize the algorithm and particularly the block-finding part, the steps are demonstrated on the form shown in Fig. 2. At the first level where the current frame is the form frame, horizontal lines are defined as the separator lines and five separator lines, which are shown with thick black lines in Fig. 2a are found. The vertical reference lines for the form frame are also shown in Fig. 2a with dashed lines. Initially, the current separator line is the top border of the form. Since there is only one vertical reference line starting from the current separator line, the first block is defined till the third horizontal separator line.

In defining the next block, there are two vertical reference lines and the block is then defined by the shorter one. The second block is thus between the third and fourth horizontal separator lines. The last block is defined between the fourth and the fifth horizontal separator lines since there is a single vertical reference line which have the same length as the borders of the current block frame. Whenever execution comes to the third phase, the initial tree which is shown in Fig. 2b is constructed. In the rest of the algorithm, all of the blocks are further partitioned since none of them is a cell. However, this time, divisions by vertical lines are performed. The algorithm is recursively applied on each of the three blocks extracted at the first pass of the algorithm. When the first block is taken as the current block, the horizontal separator lines and the vertical reference lines which are shown in Fig. 3a are obtained. At this time, separator lines are the vertical lines which are shown with thick black lines and the reference lines are the dashed horizontal lines. The first block is then defined between the first and the second separator lines according to the shortest reference line, while the second block is defined between the second and the third separator lines. Corresponding nodes for these blocks are shown in Fig. 3b. If the algorithm is again applied to the first block, the horizontal lines are taken as the separator lines as shown in Fig. 3c and since there is no reference line between the first and the second separator lines, these two separator lines define a new block. The other block is defined between the second and the fourth separator lines since the only vertical line lies between them. The first block is a cell and the execution of the algorithm stops for this block making it a leaf node in the tree. However, the execution continues for the other blocks recursively until the cells are reached.

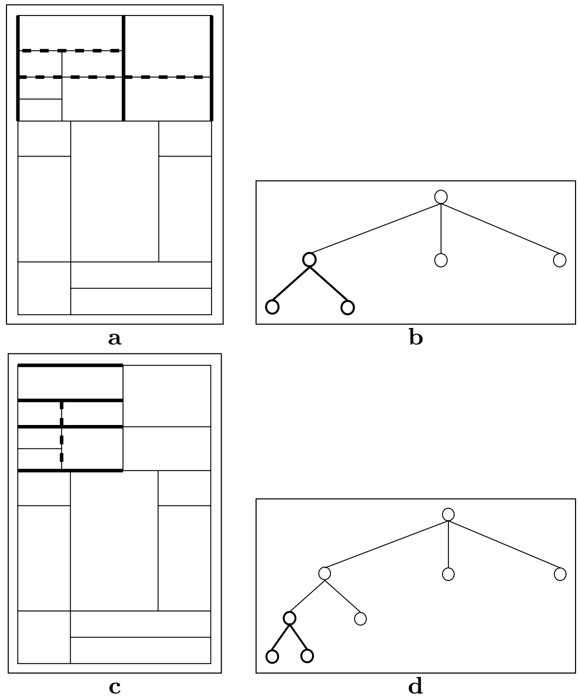


Fig. 3. a,b Horizontal separator lines (drawn as thick black lines) and vertical reference lines (drawn as dashed lines) for the current frames. **c,d** Hierarchical trees after the process of the current frames

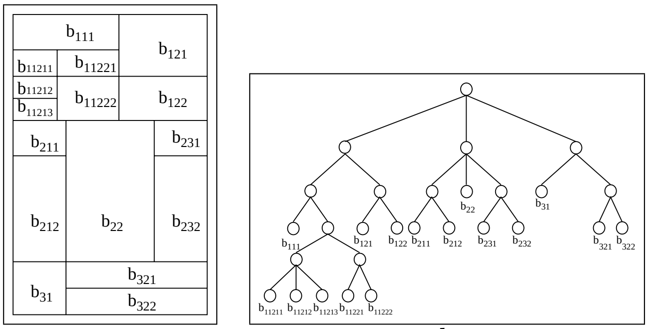


Fig. 4. a The example form shown in Fig. 1a. **b** Its representation as a tree

The other blocks are similarly processed and finally, when the algorithm terminates, the tree presented in Fig. 4b is constructed. The cells, which correspond to the preprinted or user filled-in areas on the form, are labeled to show the corresponding nodes in the tree (see Fig. 4a).

4.3 Improved algorithm for finding ending baseline

With the algorithm given in the previous two sections, blocks are defined by the shortest vertical reference line that starts at the current horizontal separator line. However, the other vertical lines become important when the forms are more complex: if there are horizontal separator lines shared among several vertical reference lines, then the ending separator line cannot be found by the information simply coming only from a single vertical reference line, since in this case the vertical reference lines are not independent from each other. Therefore, information from all of the vertical reference lines should be combined to solve the complex set of relations. For example, if there is no grouping mechanism, such as all of the vertical reference lines not starting or not ending at the same horizontal separator lines, then the block should be taken as a whole. For other cases, we have to check the ending of the vertical reference lines that are in the same current block but that start from horizontal separator line other than the current one. If at least one of these other existing vertical reference lines does not end at the same horizontal separator line as the shortest vertical reference line, then the ending horizontal separator line is taken as the one at which the longest vertical reference line ends. On the other hand, if there is any other vertical line which ends at the same horizontal separator line as the shortest vertical reference line, then this vertical line defines a block and its starting horizontal separator line should indicate the ending of the current block. Taking into account the above, we improve the algorithm to determine the ending baseline for a block as follows:

```
FindEndingBaseline(b_curr)   /* improved */
{
  /* if there are only borders of the block
     frame orthogonal to the horizontal
     separator lines */
  If (m=2) Then
    lines = 0
  Else
  {
    /* if there is no complex situation, the end
       of the shortest vertical reference line
       indicates the ending separating line */
    v_shortest = FindShortestVRL()
    lines = 1

    for i=1 to k do
    /* check all vertical lines to see if there
       is any complex situation */
    if (v_shortest.end > v'[i].start) and
       (v_shortest.end < v'[i].end)
     Then
     {
       v_longest = FindLongestVRL()
       lines = 2
       break
     }
     Else
     {
       tmp = v'[i]
       lines = 3
     }
  }

  case lines
  /* assign the ending separator line according
     to the above decisions */
  {
     0: s_end = s_curr.next
     1: s_end = v_shortest.end
     2: s_end = v_longest.end
     3: s_end = temp.start
  }
  return(s_end)
}
```

If the improved algorithm is applied to the form shown in Fig. 5a, then the ending horizontal separator line is assigned to the line where the longest vertical line ends (as shown in Fig. 5b) since the other vertical line shown by dashed line does not end at the same line where the shortest vertical line ends.

4.4 Ambiguity

As a special case, the current block may consist of only the vertical lines with the same length equal to the length of the borders of the block frame and there may be no other line parallel to the horizontal separator lines. In such a case, there is an ambiguity in defining blocks in the sense that only the vertical and horizontal lines do not give enough information about the type of division. If we consider the example in Fig. 6a, without preprinted data, it may represent both blocks shown in Fig. 6b and shown

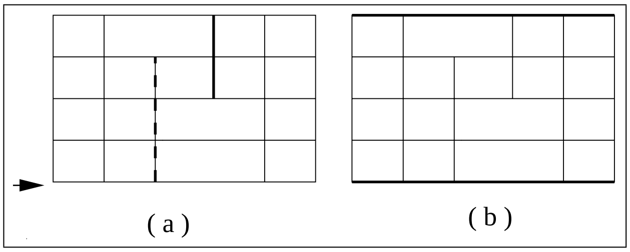


Fig. 5. **a** A complex situation in which a vertical line (shown by the dashed line) in the current block does not end at the same line where the shortest vertical line (shown by thick black line) ends. **b** In this case, the ending horizontal separator line is assigned to the line where the longest vertical line ends (thick black lines indicate the borders of the current block)

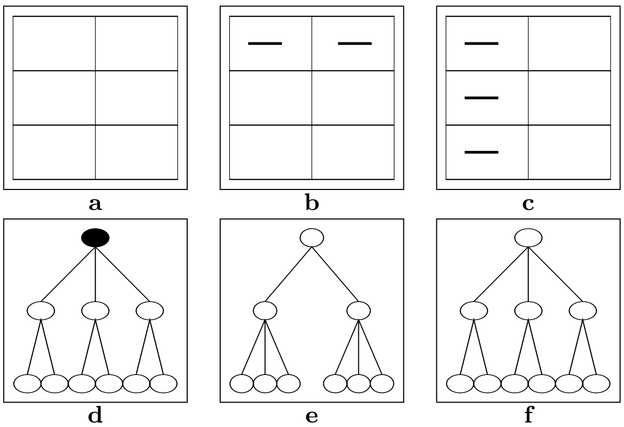


Fig. 6. **a** A block without preprinted data. **b**,**c** Alternatives for preprinted data. **d** Corresponding subtree **black node shows the ambiguity**. **e**,**f** Two subtrees **interpretations** represented by a single ambiguous subtree

in Fig. 6c. This ambiguity can be handled by defining the current block as *ambiguous* and forming new blocks between each horizontal line. In the tree construction part, a flag is used to define the ambiguity for the node which represents the current block. The rest of the tree is constructed in a usual manner. The tree for Fig. 6a is shown in Fig. 6d. An ambiguous branch corresponds to two different subtrees. Possible subtrees for Fig. 6a are shown in Fig. 6e and Fig. 6f. The ambiguity flag in the tree brings flexibility to the registration process. Without knowing anything about preprinted or user filled-in data, by using only horizontal and vertical lines, the forms can be registered even if they have slight variations.

5 Retrieval of form documents via hierarchical representation

The described hierarchical structure may be used for *identification*, i.e., search for the reference model form corresponding to an input instance form and for *retrieval*, i.e., retrieving similar forms. Both identification and retrieval can be realized by comparing hierarchical structures.

5.1 Retrieval of a reference model form for registration and Parsing

In order to extract the user filled-in data, the input instance form should be registered with one of the reference model forms in the database. An input instance form is registered with one of the model forms in the form database if their structures are identical. In our approach, the identification of forms are simplified by identification of trees. However, since there may be ambiguous branches in one of the trees, sometimes unification is performed instead of direct identification. Unifiability is possible, if one of the trees has an ambiguous subtree and the other has the corresponding non-ambiguous subtree. If the representations of hierarchical trees can directly be matched, then the trees are defined as identical. Otherwise, if there exists an ambiguity in any of the trees, the number of the nodes of two subtrees should be equal in order to be defined as unifiable.

Registration of trees (identification or unification) is performed by level order. Since the hierarchy of the tree corresponds to the hierarchy of the blocks in the form, lower levels carry more information than the higher levels (note that the root corresponds to level 0). Thus, if the trees are not matched at lower levels, then the registration process stops.

After the registration of forms, the information is obtained from the input instance form by using a parser as shown in Fig. 7. Leaf nodes of the reference model form correspond to the locations that carry the information, preprinted data or user filled-in data. Since the input instance form and the reference model form should have the same hierarchical structure, corresponding leaf nodes represent the same fields in the forms. The parser is used to extract the corresponding fields. The information about the field is taken from the reference model form and the user filled-in data is taken from the input instance form to process further. Both the reference model form and input instance form keep the information about the lines that surround cells at the leaf nodes. Since the approach is top-down, there is no need for an extra process to get the locations of the lines. The information is carried from the form frame to the cell frames. Each tree can keep its corresponding physical coordinates and this handles problems due to scaling or translation.

After parsing the tree, only leaf nodes that correspond to the locations where the user filled-in data exists are extracted. The extracted areas are processed further in order to be given as an input to an *Optical Character Recognizer (OCR).*

5.2 Retrieval of similar forms

The retrieval problem can be stated as follows [24]: "Given a form in image database and a query image, how do we retrieve form images in the database with the same or similar layout structure as the query?". There may be several motivations for retrieval: the same forms at different scales (resolutions), minor physical variations which

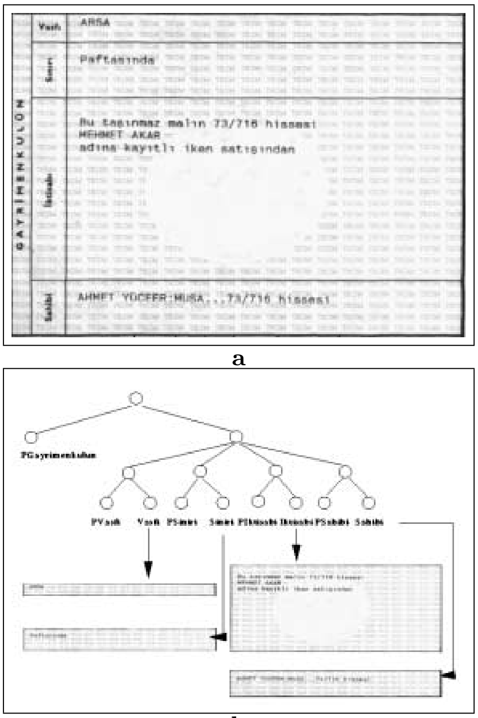


Fig. 7. a Part of a form. **b** Its hierarchical structure with field names and the corresponding extracted fields

occur over the years, etc. In the case of scale change, the hierarchical structure remains the same. However, minor variations are reflected as additional subtrees.

In this study, the extracted tree hierarchy is used for retrieval of similar forms. The similarity of two forms is computed by the distance between the corresponding two trees. The computation of distance of two trees is known as the *tree editing* problem [9,13] and is a generalization of the problem of computing the distance between two strings to labeled trees [10–12]. If the trees under consideration are unordered, then the tree editing problem becomes NP-complete [13]. The operations in the tree editing problem are changing, deleting, and inserting a node. Since, the generated trees are unordered and not labeled in our case, we have only delete and insert operations and no label change operation. The problem is to find a sequence of such operations transforming a tree T_1 into a tree T_2 with minimum cost. The distance between the trees T_1 and T_2 is then defined to be the cost of such a sequence. The insert operation is weighted with respect to the level of the node. A generated tree corresponding to a particular for document is stored as a matrix M. The indices M_{ij} of the matrix M represent the number of nodes having i children at level j. For example, the form in Fig. 4 can be represented in matrix form as shown below. In this study, we choose the maximum possible number of children of a node as 20 and the maximum possible level as 10.

	1	2	3	4	5	6	7	8	9	10
2	0	2	5	1	1	0	0	0	0	0
3	1	1	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0

This matrix representation shows that, the form has 1 node having 3 children at level 1, 2 nodes having 2 children and 1 node having 3 children at level 2, 5 nodes having 2 children at level 3, 1 node having 2 children at level 4, and 1 node having 2 children and 1 node having 3 children at level 5. Then, the similar forms are found by defining a cost for the node deletion and node insertion. The hierarchy of the form is reflected by assigning higher priorities to the higher levels in the tree (the highest priority is given to the root node). Thus, the distance between two matrices M_m and M_n is found as follows:

$$d(M_m, M_n) = \sum_{1\leq j\leq 10}(L_j * \sum_{1\leq i\leq 20}(M_{m_{ij}} - M_{n_{ij}}))$$

where L_j is the number showing the priority of level j.

For each ambiguous branch in the tree, a different matrix representation is needed, since an ambiguous branch can be represented in two ways: M nodes at level i and M*N nodes at level i+1 or N nodes at level i and N*M nodes at level i+1. Thus, for each form, more than one matrix representation can be needed. Then, the similarity between two forms F_m and F_n is found as follows:

$$D(F_m, F_n) = argmin_{k,l} d(F_m^k, F_n^l)$$

where F_m^k represents the kth matrix of the form F_m and F_n^l represents lth matrix of the form F_n. Then, the similarity of the forms are found by computing D(Fi,Fj) for each form pair.

6 Experimental results

The database that is used to assess the performance of the described algorithm for identification and retrieval contains 33 different types of form documents. These

form documents are collected either from local institutions or from the studies already published in the literature [20]. The details of our work, form documents used, and an interactive form document browser are presented on a website at:
`http://www.ceng.metu.edu.tr/~duygulu/form/index.html`

Subjective evaluation of the described algorithm can be performed via the results presented at the given url address and by the use of the interactive browser at the same address. On the other hand, for the quantitative analysis, we have used ten different types of form documents selected among our database. For each type of form document, we have generated three new subtypes :

1. Exactly the same document, but the generated tree is different because of ambiguity.
2. A new form document which is logically the same but has some geometrical modifications.
3. A new form document which is similar but has important geometrical modifications.

Figure 8a shows a sample form document which is one of those ten form types selected to be used for quantitative assessment. Its subtypes are given in Fig. 8b, Fig. 8c,d, and Fig. 8e–g as examples to the first, second, and third subtypes, respectively. For each form document type, we manually derive new form documents of the same type (logically) which may have different geometrical structures. The physical difference is obtained by slight changes in the position and length of the lines. For example, in the form document given in Fig. 8c, the first vertical line is no more a long vertical line as it is in the original form document shown in Fig. 8a, but it has been divided into three smaller vertical lines. The other form documents given in Fig. 8d also have similar modifications. In addition to these, there are also new form documents which are expected to be similar to the form type, i.e., the original form document. For example, in the form document demonstrated in Fig. 8e the first vertical line has totally disappeared with respect to the form type.

We have conducted three types of experiments to assess the performance of the proposed method for retrieval. The results are given in terms of the retrieval rate in each experiment. In the first experiment, the set of form documents contains the ten form documents and their ambiguous versions. Therefore, we have a total of 20 form documents which means two examples per form type. The purpose in this experiment is to assess the effect of different interpretations of the same physical form document on the retrieval rate. Table 1 shows the retrieval rate versus the number of retrieved documents for this experiment. The results demonstrate that if the ambiguity is at a lower level, i.e., in the sub-blocks, then the form documents can be retrieved among the first few candidates. The decrease in the retrieval rate is because of the fact that the ambiguity is at a level close to the root node.

The second experiment is performed with the form documents of ambiguous subtypes and of the same logical structure but having geometrical modifications. In this experiment, we have a total of 51 form documents and at least two examples per form type. As can be observed from the sample form documents shown in Fig. 8c,d, even if there is a slight geometrical change, the human interpretation varies drastically. Table 2 shows the retrieval rate versus the number of retrieved documents for this experiment. The retrieval rate is around 70%.

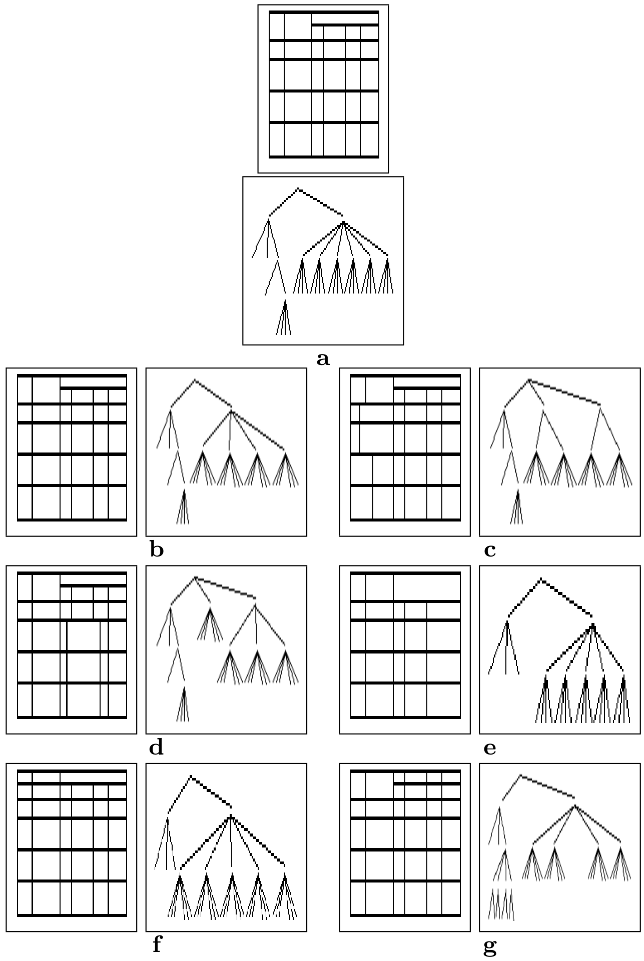


Fig. 8. **a** One of the ten form document types (original form document) and its representation. **b** The same form document, but the tree representation in the ambiguous part is different. **c,d** New form documents (subtypes) which are logically the same but have some geometrical modifications and their representations. **e–g** New form documents (subtypes) which are similar but have important geometrical modifications and their representations

Table 1. Retrieval rate when the form documents with ambiguous blocks are considered

No. of retrieved documents	Retrieval rate (%)
2	77.5
3	90.0
4	100

Table 2. Retrieval rate versus the number of retrieved documents when the set contains form documents of ambiguous subtypes and of the same logical structure but with geometrical modifications

No. of retrieved documents	Retrieval rate (%)
2	86.3
4	71.5
5	63.7
8	72.9
10	80.5
15	87.7
20	91.3
25	92.4

Table 3. Retrieval rate versus the number of retrieved documents when all of the form documents are used, but especially the subtypes which are similar but which have important geometrical modifications

No. of retrieved documents	Retrieval rate (%)
2	75.4
4	53.6
5	48.2
8	38.4
10	35.4
15	41.7
20	51.3
25	59.4

In the third and final experiment, we have used all the different subtypes of form documents which seems to be a rather difficult retrieval problem. One hundred and twelve example form documents are employed in this experiment and there are at least ten examples for each form document type. Table 3 shows the retrieval rate versus the number of retrieved documents for this experiment.

Since higher nodes are given higher priorities during the coding scheme, the level of detail is an important criteria regarding similarity. Forms having a complex structure are closer to each other, rather than the simpler forms, and vice versa. Forms designed for the same application, but having some slight differences, are found to be the most similar ones among the others. The results show that the retrieval of form documents is rather a difficult problem in the sense that we still need a better representation scheme for a form document.

7 Conclusion

Identification and retrieval is an essential process even for a single type of form, since there may be modifications on the physical structure of the form or there may be more than one design for the same application-specific form. Most of the current studies use physical features and can handle only some translation or scaling problems. However, there is a need for more logical features if the forms have several different physical structures and if the number of forms to be compared is large.

In this study, the logical structure of the forms is represented by an XY-tree which corresponds to the hierarchy of the blocks on the form. Since the forms are designed in a hierarchical manner – that is, the blocks which contain similar information are grouped together and separated from the other groups – this representation is also close to the human point of view. The described hierarchical structure is used both for identification and retrieval. A coding scheme along with a well-known tree editing method is used for the comparison of hierarchical structures. Several experiments are conducted to assess the performance of the described method. However, it is still necessary to perform extensive experimental work to further test the ideas described in this study in a comparative manner with respect to the other existing algorithms in the literature.

One of the most similar studies in the literature is that of Watanabe et al. However, their method is more suitable for tabular documents rather than form documents. Furthermore, regarding the relations among the document elements, i.e., horizontal and vertical lines, blocks are represented in two different trees and the registration or identification is performed in a rather complicated manner. On the other hand, the method described in this paper is particularly useful for form documents and the output is a single tree from which identification and retrieval is directly possible through popular existing algorithms. The second closest study is that of Ishitani [7] in which the ideas and the results are very impressive. However, all of the proposed processes and, particularly, the representation depend on the horizontal and vertical lines. There is no abstract representation of the form document. A form document is represented by its physical features-horizontal and vertical lines. On the other hand, we generate a tree whose hierarchy corresponds to the layout in a form document. The tree is an abstract representation of the input form document. One advantage of representing a form document by a tree is that in the literature, there are several available algorithms to compare trees; therefore we do not need to include heuristics to determine similarities among the form documents. However, the disadvantage of using an abstraction is that we lose some of the low-level details of the structures of the form documents and this causes a certain degradation in the retrieval rate. In this context, Ishitani's method may give better results.

Acknowledgements. We thank Ali İnce for his help with programming.

References

1. G.Nagy, S.Seth Hierarchical representation of optically scanned documents. Proc. ICPR, pp. 347–349 (1984)

2. F.Cesarini, M.Gori, S.Marinai, G.Soda: Structured document segmentation and representation by the modified X-Y tree. Proc. 5th ICDAR, pp. 563–566 (1999)
3. Y. Y. Tang, C. D.Yan, M.Cheriet, C. Y.Suen: Automatic analysis and understanding of documents. In: Handbook of pattern recognition and computer vision, pp. 625–654 (1993)
4. D.Niyogi, S.Srihar: Using domain knowledge to derive the logical structure of documents. SPIE, pp. 114–125 (1996)
5. B.Yu, A. K.Jain: A form dropout system. Proc. 13th Int. Conf. on Pattern Recognition, ICPR'96, Vienna, Austria, August, 1996, pp. 701–705
6. D. Wang, S. N.Srihari: Analysis of form images. Proc. 1st Int. Conf. on Document Analysis and Recognition, ICDAR'91, Saint-Malo, France, Sept., 1991, pp. 181–191
7. Y. Ishitani: Flexible and robust model matching based on association graph for form image understanding. Pattern Anal Appl 3:104-119 (2000)
8. F. Dubiel, A. Dengel: FormClass: a System for OCR free identification of forms. Document Analysis Systems II, World Scientific, pp. 189–209 (1998)
9. T. Richter: A new measure of the distance between ordered trees and its applications. Technical Report 85166, Dept. of Computer Science, Uni. of Bonn (1997)
10. V.I. Levenshtein: Binary codes capable of correcting deletions. Soviet Phys Dokl 6:707–710 (1966)
11. P.H. Sellers: On the theory and computation of evolutionary distances. SIAM J Appl Math 26:787–793 (1974)
12. R.A. Wagner, M.J. Fischer: The string-to-string correction problem. JACM 21:168–173 (1974)
13. K. Zhang, R. Statman, D.Shasha: On the editing distance between unordored labeled trees. IPL 42:133–139 (1992)
14. R. Casey, D. Ferguson, K. Mohiuddin, E. Walach: Intelligent forms processing system. Mach Vision Appl 5(3):143–155 (1992)
15. J. Mao, M. Abayan, K. Mohiuddin: A Model-based form processing sub-system. Proc. 13th Int. Conf. on Pattern Recognition, ICPR'96, Vienna, Austria, August, 1996, pp. 691–695
16. S. Taylor, R. Fritzson, J. Pastor: Extraction of data from preprinted forms. Mach Vision Appl 5:211–222 (1992)
17. J. Lin, C. Lee, Z. Chen: Identification of business forms using relationships between adjacent frames. Mach Vision Appl 9:56–64 (1996)
18. S. Shimotsuji, M. Asano: Form identification based on cell structure. Proc. 13th Int. Conf. on Pattern Recognition, ICPR'96, Vienna, Austria, August, 1996, pp. 793–797
19. Y. Hirayama: Analyzing form images by using line-shared-adjacent cell relations. Proc. 13th Int. Conf. on Pattern Recognition, ICPR'96, Vienna, Austria, August, 1996, pp. 768–772
20. T. Watanabe, Q. Luo, N. Sugie: Layout recognition of multi-kinds of table-form documents. IEEE Trans Pattern Anal Mach Intell 17(4):432–445 (1995)
21. J. Liu, X. Ding, Y. Wu: Description and recognition of form and automated form data entry. Proc. Third Int. Conf. on Document Analysis and Recognition, ICDAR'95, 1995, pp. 579–582
22. O. Hori, D.S. Doermann: Robust table-form structure analysis based on box-driven reasoning. Proc. Third Int. Conf. on Document Analysis and Recognition, ICDAR'95, 1995, pp. 218–221
23. P. Heroux, S. Diana, A. Ribert, E. Trupin: Classification method study for automatic form class identification. Proc. 14th Int. Conf. on Pattern Recognition, ICPR'98, Brisbane, Australia, August (1998)
24. J. Liu, A.K. Jain: Image-based form document retrieval. Proc. 14th Int. Conf. on Pattern Recognition, ICPR'98, Brisbane, Australia, August (1998)
25. P. Duygulu, V. Atalay, E. Dincel: A heuristic algorithm for hierarchical representation of form documents. Proc. 14th Int. Conf. on Pattern Recognition, ICPR'98, Brisbane, Australia, August (1998)
26. P. Duygulu, V. Atalay: A hierarchical representation of form documents for identification and retrieval. SPIE, Electronic Imaging 2000, Document Recognition and Retrieval VII, San Jose, USA, January (2000)