Mining Data Records in Web Pages

Bing Liu
Department of Computer Science
University of Illinois at Chicago
851 S. Morgan Street
Chicago, IL 60607-7053
liub@cs.uic.edu

Robert Grossman
Dept. of Mathematics, Statistics, and Computer Science
University of Illinois at Chicago
851 S. Morgan Street, IL 60607
grossman@uic.edu

Yanhong Zhai
Department of Computer Science
University of Illinois at Chicago
851 S. Morgan Street
Chicago, IL 60607-7053
yzhai@cs.uic.edu

ABSTRACT
A large amount of information on the Web is contained in regularly structured objects, which we call *data records*. Such data records are important because they often present the essential information of their host pages, e.g., lists of products and services. It is useful to mine such data records in order to extract information from them to provide value-added services. Existing approaches to solving this problem mainly include the manual approach, supervised learning, and automatic techniques. The manual method is not scalable to a large number of pages. Supervised learning needs manually prepared positive and negative training data and thus also require substantial human effort. Current automatic techniques are still unsatisfactory because of their poor performances. In this paper, we propose a much more effective automatic technique to perform the task. This technique is based on two important observations about data records on the Web and a string matching algorithm. The proposed technique is able to mine both contiguous and non-contiguous data records. By non-contiguous data records, we mean that two or more data records intertwine in terms of their HTML codes. When they are displayed on a browser, each of them appears contiguous. Existing techniques are unable to mine such records. Our experimental results show that the proposed technique outperforms existing techniques substantially.

Categories and Subject Descriptors
I.5 [Pattern Recognition]: statistical and structural.
H.2.8 [Database Applications]: data mining

Keywords
Web data records, Web mining

1. INTRODUCTION
A large amount of information on the Web is presented in regularly structured objects. A list of such objects in a Web page often describes a list of similar items, e.g., a list of products or services. They can be regarded as database records displayed in Web pages with regular patterns. In this paper, we also call them *data records*. Mining data records in Web pages is useful because it allows us to extract and integrate information from multiple sources to provide value-added service, e.g., customizable Web information gathering, comparative-shopping, meta-search, etc. Figure 1 gives an example, which is a segment of a Web page that lists two Apple notebooks. The full description of each notebook is a data record. The objective of the proposed technique is to automatically mine all the data records in a given Web page.

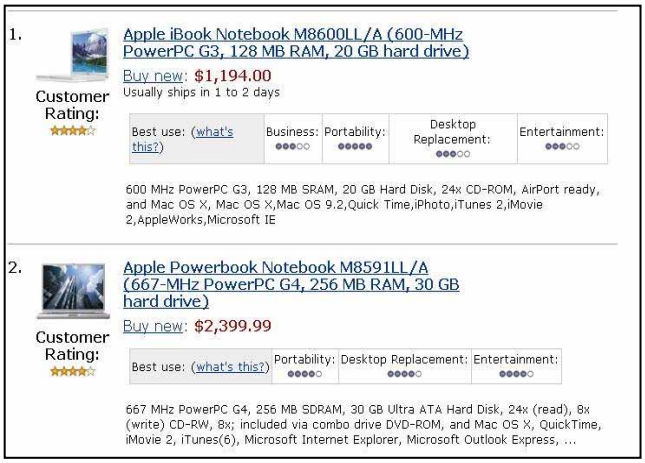

Figure 1. An example: two data records

Several approaches have been reported in the literature for mining data records (or their boundaries) from Web pages. The first approach is the manual approach. By observing a Web page and its source code, the programmer can find some patterns and then writes a program to identify each data record. This approach is not scalable to a large number of pages. Other approaches [2][4][6][7][8][9][10][12][14][16][17][18][19][21][22][23] all have some degree of automation. They rely on some specific HTML tags and/or machine learning techniques to separate objects. These methods either require prior syntactic knowledge or human labeling of specific regions in the Web page to mark them as interesting. [10] presents an automatic method which uses a set of heuristics and domain ontology to perform the task. Domain ontology is costly to build (about 2-man weeks for a given Web site) [10]. [2] extends this approach by designing some additional heuristics without using any domain knowledge. We will show in the experiment section that the performance of this approach is poor. [4] proposes another automatic method, which uses Patricia tree [11] and approximate sequence alignment to find patterns (which represent a set of data records) in a Web page. Due to the inherent limitation of Patricia tree and inexact sequence matching, it often produces many patterns and most of them are spurious. In many cases, none of the actual data records is found. Again, this method performs poorly. [19] proposes a method using clustering and grammar induction of regular languages. As shown in [19],

Conference '00, Month 1-2, 2000, City, State.

the results are not satisfactory. More detailed discussions on these and other related works will be given in the next section.

Another problem with existing automatic approaches is that they assume that the relevant information of a data record is contained in a contiguous segment of the HTML code. This model is insufficient because in some Web pages, the description of one object (or a data record) may intertwine with the descriptions of some other objects in the HTML source code. For example, the descriptions of two objects in the HTML source may follow this sequence, part1 of object1, part1 of object2, part2 of object1, part2 of object2. Thus, the descriptions of both object1 and object2 are not contiguous. However, when they are displayed on a Web browser, they appear contiguous to human viewers.

In this paper, we propose a novel and more effective method to mine data records in a Web page automatically. The algorithm is called MDR (Mining Data Records in Web pages). It currently finds all data records formed by table and form related tags, i.e., table, form, tr, td, etc. A large majority of Web data records are formed by them. Note that it is also capable of finding nested data records. Our method is based on two observations:

1. A group of data records that contains descriptions of a set of similar objects are typically presented in a contiguous region of a page and are formatted using similar HTML tags. Such a region is called a *data record region* (or *data region* in short). For example, in Figure 1 two notebooks are presented in one contiguous region. They are also formatted using almost the same sequence of HTML tags. If we regard the HTML formatting tags of a page as a long string, we can use string matching to compare different sub-strings to find those similar ones, which may represent similar objects or data records.

   The problem with this approach is that the computation is prohibitive because a data record can start from anywhere and end anywhere. A set of data records typically do not have the same length in terms of their tag strings because they may not contain exactly the same pieces of information (see Figure 1). The next observation helps us to deal with this problem.

2. The nested structure of HTML tags in a Web page naturally forms a *tag tree* [3]. Our second observation is that a group of similar data records being placed in a specific region is reflected in the tag tree by the fact that they are under one parent node, although we do not know which parent (our algorithm will find out). For example, the tag tree for the page in Figure 1 is given in Figure 2 (some details are omitted). Each notebook (a data record) in Figure 1 is wrapped in 5 TR nodes with their sub-trees under the same parent node TBODY (Figure 2). The two data records are in the two dash-lined boxes. In other words, a set of similar data records are formed by some child sub-trees of the same parent node.

   A further note is that it is very unlikely that a data record starts inside of a child sub-tree and ends inside another child sub-tree. Instead, it starts from the beginning of a child sub-tree and ends at the same or a later child sub-tree. For example, it is unlikely that a data record starts from TD* and ends at TD# (Figure 2). This observation makes it possible to design a very efficient algorithm to mine data records.

Our experiments show that these observations are true. So far, we have not seen any Web page containing a list of data records that violates these observations. Note that we do not assume that a Web page has only one data region that contains data records. In fact, a Web page may contain a few data regions. Different regions may have different data records. Our method only requires that a data region to have two or more data records.

Given a Web page, the proposed technique works in three steps:

Step 1: Building a HTML tag tree of the page.

Step 2: Mining data regions in the page using the tag tree and string comparison. Note that instead of mining data records directly, which is hard, our method mines data regions first and then find the data records within them. For example, in Figure 2, we first find the single data region below node TBODY.

Step 3: Identifying data records from each data region. For example, in Figure 2, this step finds data record 1 and data record 2 in the data region below node TBODY.

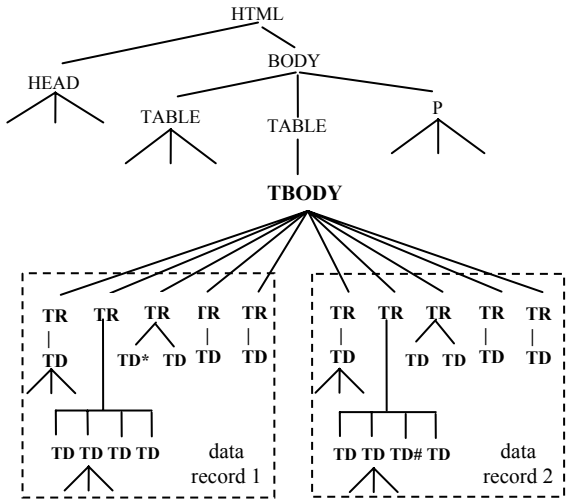

Figure 2. Tag tree of the page in Figure 1

Our contributions

1. A novel and effective technique is proposed to automatically mine data records in a Web page. Extensive evaluation using a large number of Web pages from diverse domains show that the proposed technique is able to produce dramatically better results than the state-of-the-art automatic systems in [2][4] (even without considering non-contiguous data records).
2. Our new method is able to discover non-contiguous data records, which cannot be handled by existing methods. Our technique is able to handle such cases because it explores the nested structure and presentation features of Web pages.

2. RELATED WORK

Web information extraction from regularly structured data records is an important problem. Identifying individual data records is often the first step. So far, several attempts have been made to deal with the problem. We discuss them below.

Related works to ours are mainly in the area of wrapper generation. A wrapper is a program that extracts data from a website and put them in a database [3][12][13][16]. There are several approaches to wrapper generation. The first approach is to manually write a wrapper for each Web page based on observed format patterns of the page. This approach is labor intensive and very time consuming. It cannot scale to a large number of pages.

The second approach is wrapper induction, which uses supervised learning to learn data extraction rules. A wrapper induction system is typically trained by using manually labeled positive and negative data. Thus, it still needs substantial manual effort as

labeling of data is also labor intensive and time consuming. Additionally, for different sites or even pages in the same site, the manual labeling process may need to be repeated. Example wrapper induction systems include WIEN [17], Softmealy [14], Stalker [21], WL [6], etc. Our technique requires no human involvement. It mines data records in a page automatically. [19] reports a unsupervised method based on clustering and grammar induction. However, the results are unsatisfactory due to the deficiencies of the techniques as noted in [19].

[9] reports a comparative shopping agent, which also tries to identify each product from search results. A number of heuristics rules are used to find individual products, e.g., price information, and required attributes from the application domain

In [10], a more general study is made to automatically identify data record boundaries. The method is based on 5 heuristic rules.

(1) Highest-count tags: This rule says that those highest-count tags are more likely to be record boundary tags.

(2) Identifiable "separate" tags: This rule says that tags, such as ht, td, tr, table, p, h1, etc., are more likely to be boundary tags.

(3) Standard deviation: This rule computes the number of characters between each occurrence of a candidate separator tag. Those with smallest standard deviation are assumed to be more likely boundary tags.

(4) Repeating-tag pattern: This rule says that boundaries often have consistent patterns of two or more adjacent tags.

(5) Ontology-matching: This rule uses domain knowledge to find those domain specific terms that only appear once and only once in data records.

A combination technique (based on certainty factor [10] in AI) is used to combine the heuristics to make the final decision.

[2] proposes some more heuristics to perform the task, e.g., sibling tag heuristic, which counts the pairs of tags that are immediate siblings in a tag tree, and partial path heuristic, which lists the paths from a node to all other reachable nodes and counts the number of occurrences of each path. It is shown in [2] that the new method (OMINI) performs better than the system in [10], even without using domain ontology. Our technique is very different from these tag based heuristic techniques. We will show that our method outperforms these existing methods dramatically.

[4] proposes a method (IEPAD) to find patterns from the HTML tag string, and then use the patterns to extract objects. The method uses a PAT tree (a Patricia tree [11]) to find patterns. The problem with the PAT tree is that it is only able to find exact match of patterns. In the context of the Web, data records are seldom exactly the same. Thus, [4] also proposes a heuristic method based on string alignment to find inexact matches. However, this method results in many patterns, most of which are spurious. For example, for the same segment of a tag string many patterns may be found and they intersect one another. In many cases the right patterns in the page are not found. As we will see in the experiment section, the result of this method is also poor.

Finally, all the above automatic methods assume that each record is contiguous in the HTML source. This is not true in some Web pages as we will see in Sections 3.3 and 4. Our method does not make this assumption. Thus, it is able to deal with the problem.

3. THE PROPOSED TECHNIQUE

As mentioned earlier, the proposed method has three main steps. This section presents them in turn.

3.1 Building the HTML Tag Tree

Web pages are hypertext documents written in HTML that consists of plain texts, tags and links to image, audio and video files, etc. In this work, we only use tags in string comparison to find data records. Most HTML tags work in pairs. Each pair consists of an *opening* tag and a *closing* tag (indicated by <> and </> respectively). Within each corresponding tag-pair, there can be other pairs of tags, resulting in nested blocks of HTML codes. Building a *tag tree* from a Web page using its HTML code is thus natural. In our tag tree, each pair of tags is considered as one *node*, and the nested pairs of tags within it are the *children* of the node. This step performs two tasks:

1. *Preprocessing of HTML codes*: Some tags do not require closing tags (e.g., <li> and <hr>). Hence, additional closing tags are inserted to ensure all tags are balanced. Next, useless or redundant tags are removed. Examples include tags related to HTML comments <!--, <script>, and others.
2. *Building a tag tree*: It follows the nested blocks of the HTML tags in the page to build a tag tree. This is fairly easy. We will not discuss it further. Figure 2 shows an example.

3.2 Mining Data Regions

This step mines every data region in a Web page that contains similar data records. Instead of mining data records directly, which is hard, we first mine *generalized nodes* (defined below) in a page. A sequence of adjacent generalized nodes forms a data region. From each data region, we will identify the actual data records (discussed in Section 3.3). Below, we define generalized nodes and data regions using the HTML tag tree:

Definition: A *generalized node* (or a *node combination*) of length r consists of r ($r \geq 1$) nodes in the HTML tag tree with the following two properties:

1) the nodes all have the same parent.
2) the nodes are adjacent.

The reason that we introduce the generalized node is to capture the situation that an object (or a data record) may be contained in a few sibling tag nodes rather than one. For example, in Figures 1 and 2, we can see that each notebook is contained in five table rows (or 5 TR nodes). Note that we call each node in the HTML tag tree a *tag node* to distinguish it from a generalized node.

Definition: A *data region* is a collection of two or more generalized nodes with the following properties:

1) the generalized nodes all have the same parent.
2) the generalized nodes all have the same length.
3) the generalized nodes are all adjacent.
4) the normalized edit distance (string comparison) between adjacent generalized nodes is less than a fixed threshold.

For example, in Figure 2, we can form two generalized nodes, the first one consists of the first 5 children TR nodes of TBODY, and the second one consists of the next 5 children TR nodes of TBODY. It is important to notice that although the generalized nodes in a data region have the same length (the same number of children nodes of a parent node in the tag tree), their lower level nodes in their sub-trees can be quite different. Thus, they can capture a wide variety of regularly structured objects.

To further explain different kinds of generalized nodes and data regions, we make use of an artificial tag tree in Figure 3. For notational convenience, we do not use actual HTML tag names but ID numbers to denote tag nodes in a tag tree. The shaded areas are generalized nodes. Nodes 5 and 6 are generalized nodes

of length 1 and they together define the data region labeled 1 if the edit distance condition 4) is satisfied. Nodes 8, 9 and 10 are also generalized nodes of length 1 and they together define the data region labeled 2 if the edit distance condition 4) is satisfied. The pairs of nodes (14, 15), (16, 17) and (18, 19) are generalized nodes of length 2. They together define the data region labeled 3 if the edit distance condition 4) is satisfied.

It should be emphasized that a data region includes the sub-trees of the component nodes, not just the component nodes alone.

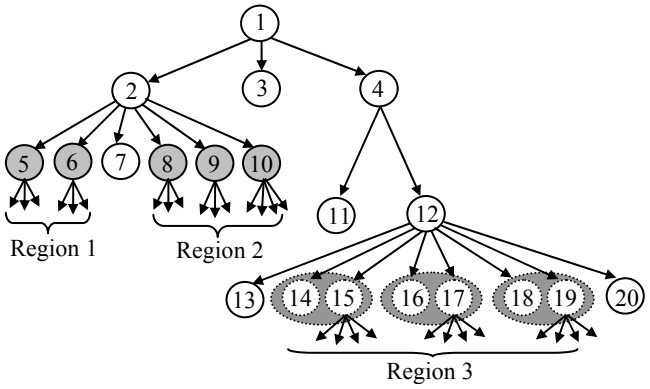

Figure 3: An illustration of generalized nodes and data regions

We end this part with some important notes:

1. In practice, the above definitions are very robust as our experiments show. The key assumption here is that nodes forming a data region are from the same parent, which is reasonable. For example, it is unlikely that a data region starts at node 7 and ends at node 14 (see also Figure 2).
2. A generalized node may not represent a final data record (see Section 3.3). It will be used to find the final data records.
3. It is possible for a single parent node to cover more than one data region, e.g., node 2 in Figure 3. Nodes adjacent to a data region may or may not be part of that region. For example, in Figure 3, nodes 7, 13, and 20 are not part of any data region.

3.2.1 Comparing Generalized Nodes

In order to find each data region in a Web page, the mining algorithm needs to find the following. (1) Where does the first generalized node of a data region start? For example, in Region 2 of Figure 3, it starts at node 8. (2) How many tag nodes or components does a generalized node in each data region have? For example, in Region 2 of Figure 3, each generalized node has one tag node (or one component).

Let the maximum number of tag nodes that a generalized node can have be *K*, which is normally a small number (< 10). In order to answer (1), we can try to find a data region starting from each node sequentially. To answer (2), we can try: one node, two node combination, …, *K* node combination. That is, we start from each node and perform all 1-node string comparisons, all 2-node string comparisons, and so on (see the example below). We then use the comparison results to identify each data region.

The number of comparisons is actually not very large because:

- Due to our assumption, we only perform comparisons among the children nodes of a parent node. For example, in Figure 3, we do not compare node 8 with node 13.
- Some comparisons done for earlier nodes are the same as for later nodes (see the example below).

We use Figure 4 to illustrate the comparison process. Figure 4 has 10 nodes, which are below a parent node, *p*. We start from each node and perform string comparison of all possible combinations of component nodes. Let the maximum number of components that a generalized node can have be 3 in this example.

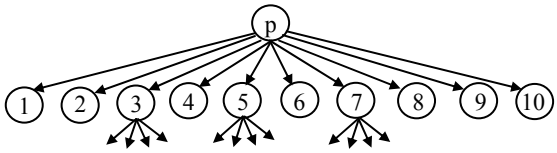

Figure 4: combination and comparison

Start from node 1: We compute the following string comparisons.

- (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8), (8, 9), (9, 10)
- (1-2, 3-4), (3-4, 5-6), (5-6, 7-8), (7-8, 9-10)
- (1-2-3, 4-5-6), (4-5-6, 7-8-9)

(1, 2) means that the tag string of node 1 is compared with the tag string of node 2. The tag string of a node includes all the tags of the sub-tree of the node. For example, in Figure 2, the tag string for the second TR node below TBODY is <TR TD TD … TD TD>, where "…"denotes the sub-string of sub-tree below the second TD node. The tag string for the third TR node below TBODY is <TR TD TD>.

(1-2, 3-4) means that the combined tag string of nodes 1 and 2 is compared with the combined tag string of nodes 3 and 4.

Start from node 2: We only compute:

- (2-3, 4-5), (4-5, 6-7), (6-7, 8-9)
- (2-3-4, 5-6-7), (5-6-7, 8-9-10)

We do not need to do 1-node comparisons because they have been done when we started from node 1 above.

Start from node 3: We only need to compute:

- (3-4-5, 6-7-8)

Again, we do not need to do 1-node comparisons. Here, we also do not need to do 2-node comparisons because they have been done when we started from node 1.

We do not need to start from any other nodes after node 3 because all the computations have been done. It is fairly easy to prove that the process is complete. It is omitted here due to space limitations.

The overall algorithm (MDR) for computing all the comparisons at each node of a tag tree is given in Figure 5. It traverses the tag tree from the root downward in a depth-first fashion (lines 3 and 4). At each internal node, procedure CombComp (Figure 6) performs string comparisons of various combinations of the children sub-trees. Line 1 says that the algorithm will not search for data regions if the depth of the sub-tree from *Node* is 2 or 1 as it is unlikely that a data region is formed with only a single level of tag(s) (data regions are formed by the children of *Node*).

```
Algorithm MDR(Node, K)
1   if TreeDepth(Node) >= 3 then
2       CombComp(Node.Children, K);
3       for each ChildNode ∈ Node.Children
4           MDR(ChildNode, K);
5   end
```

Figure 5: The overall algorithm

The main idea of CombComp has been discussed above. In line 1 of Figure 6, it starts from each node of *NodeList*. It only needs to try up to the *K*th node. In line 2, it compares different combinations of nodes, beginning from *i*-component combination

to *K*-components combination. Line 3 tests to see whether there is at least one pair of combinations. If not, no comparison is needed. Lines 4-8 perform string comparisons of various combinations by calling procedure EditDist, which compares two strings using edit distance [1][11].

```
CombComp(NodeList, K)
1  for (i = 1; i <= K; i++)        /* start from each node */
2     for (j = i; j <= K; j++)     /* comparing different combinations
3        if NodeList[i+2*j-1] exists then
4           St = i;
5           for (k = i+j; k < Size(NodeList); k+j)
6              if NodeList[k+j-1] exists then
7                 EditDist(NodeList[St..(k-1)],
                           NodeList[k..(k+j-1)]);
8                 St = k;
9  end
```

Figure 6: The structure comparison algorithm

Assume that the number of element in *NodeList* is *n*. Without considering the edit distance comparison, the time complexity of CombComp is $O(nK)$, which is the number of times that we need to run EditDist. To see this, let us do the following analysis:

Starting at node 1, we need at most the following number of comparisons (running EditDist) (we assume that *n* is much large than *K* or $n / K \geq 2$).

$$(n-1) + (\frac{n}{2} - 1) + (\frac{n}{3} - 1) + ... + (\frac{n}{K} - 1)$$

Starting from node 2, we have at most:

$$(\frac{n}{2} - 1) + (\frac{n}{3} - 1) + ... + (\frac{n}{K} - 1)$$

....

Starting from node *K*, we have at most: $(\frac{n}{K} - 1)$

Adding all together, we have $nK - \frac{K(K+1)}{2}$, which gives $O(nK)$ (we assume that *n* is much larger than *K*). Since *K* is normally small (< 10), the algorithm can be considered linear in *n*. Assume that the total number of nodes in the tag tree is *N*, the complexity of MDR is $O(NK)$ without considering string comparison.

3.2.2 String Comparison Using Edit Distance

The string comparison method that we use is based on *edit distance* (also known as Levenshtein distance) [1][11], which is a widely-used string similarity measure. In this work, we used a normalized version of edit distance to compare similarity between two strings. The edit distance of two strings, s_1 and s_2, is defined as the minimum number of *point mutations* required to change s_1 into s_2, where a point mutation is one of: (1) change a letter, (2) insert a letter and (3) delete a letter. As edit distance is a well known technique, we will not discuss it further in this paper.

The Normalized edit distance $ND(s_1, s_2)$ is obtained by dividing the edit distance by the mean length of the two strings s_1 and s_2:

$$ND(s_1, s_2) = \frac{d(s_1, s_2)}{(length(s_1) + length(s_2))/2}$$

The time-complexity of the algorithm is $O(|s_1||s_2|)$ [1]. In our application, the computation can be substantially reduced because we are only interested in very similar strings. The computation is only large when the strings are long. If we want the strings to have the similarity of more than 50%, we can use the following method to reduce the computation:

- If $|s_1| > 2|s_2|$ or $|s_2| > 2|s_1|$, no comparison is needed because they are obviously too dissimilar.

3.2.3 Determining Data Regions

After all string comparisons have been done, we are ready to identify each data region by finding its generalized nodes. We use Figure 7 to illustrate the main issues. There are 8 data records (1-8) in this page. Our algorithm reports each row as a generalized node, and the whole area (the dash-lined box) as a data region.

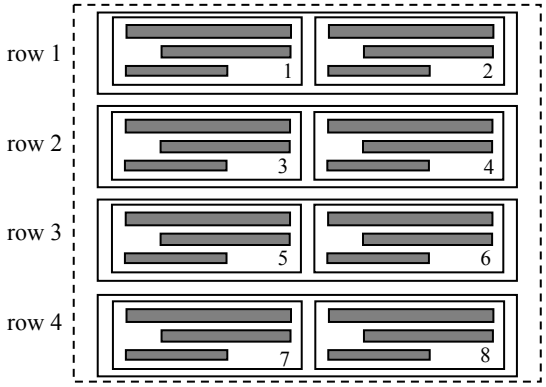

Figure 7. A possible configuration of data records

The algorithm basically uses the string comparison results at each parent node to find similar children node combinations to obtain candidate generalized nodes and data regions of the parent node. Three main issues are important for making the final decisions.

1. If a higher level data region covers a lower level data region, we report the higher level data region and its generalized nodes. *Cover* here means that a lower level data region is within a higher level data region. For example, in Figure 7, at a low level we find that cell 1 and cell 2 are candidate generalized nodes and they together form a candidate data region, row 1. However, they are covered by the data region including all the 4 rows at a higher level. In this case, we only report each row is a generalized node. The reason for taking this approach is to avoid the situations where many very low level nodes (with very small sub-trees) are very similar but do not represent true data records.
2. A property about similar strings is that if a set of strings s_1, s_2, s_3, …., s_n, are similar to one another, then a combination of any number of them is also similar to another combination of them of the same number. Thus, we only report generalized nodes of the smallest length that cover a data region, which helps us to find the final data records later. In Figure 7, we only report each row as a generalized node rather than a combination of two rows (rows 1-2, and rows 3-4).
3. An edit distance threshold is needed to decide whether two strings are similar. We used a set of training pages to decide it to be 0.3, which performs very well in general (see Section 4).

The algorithm for this step is given in Figure 8. It finds every data region and its generalized nodes in a page. *T* is the edit distance threshold. *Node* is any node. *K* is the maximum number of tag nodes in a generalized node (we use 10 in our experiments, which is sufficient). *Node.DRs* is the set of data regions under *Node*, and *tempDRs* is a temporal variable storing the data regions passed up from every *Child* of *Node*. Line 1 is the same as line 1 in Figure 5.

The basic idea of the algorithm is to traverse the tag tree top-down in a depth-first fashion. It performs one function at each node when it goes down (line 2), and performs another when it backs up before going down to another branch of the tree (line 6).

1. When it goes down, at each node it identifies all the data regions of the node using procedure IdentDRs (line 2). Note that these are not the final data regions of the Web page, but only the candidate ones of this node (see below).
2. When it backs up, it checks to see whether the parent level data regions in *Node.DRs* cover the child level data regions. Those covered child level data regions are discarded. We take the parent level data regions as we believe they are more likely to be true data regions. Those uncovered data regions in *Child.DRs* are returned and stored in *tempDRs* (line 6). After all the children nodes of *Node* are processed, *Node.DRs* ∪ *tempDRs* gives the current data regions discovered from the sub-tree starting from *Node* (line 7).

```
Algorithm FindDRs(Node, K, T)
1  if TreeDepth(Node) => 3 then
2     Node.DRs = IdenDRs(1, Node, K, T);
3     tempDRs = ∅;
4     for each Child ∈ Node.Children do
5        FindDRs(Child, K, T);
6        tempDRs = tempDRs∪UnCoveredDRs(Node, Child);
7     Node.DRs = Node.DRs ∪ tempDRs
8  end
```

Figure 8: Finding all data regions in the tag tree

We now discuss procedure IdentDRs. Recall that the previous step has computed the distance values of all possible child node combinations. This procedure uses these values and the threshold *T* to find data regions of *Node*. That is, it needs to decide which combinations represent generalized nodes, where the beginning is and where the end is for each data region.

Procedure IdentDRs is given in Figure 9, which ensures the smallest generalized nodes are identified.

```
Procedure IdentDRs(start, Node, K, T)
1  maxDR = [0, 0, 0];
2  for (i = 1; i <= K; i++)     /* compute for each i-combination */
3     for (f = start; f <= start+i; f++) /* start from each node */
4        flag = true;
5        for (j = f; j < size(Node.Children); j+i)
6           if Distance(Node, i, j) <= T then
7              if flag=true then
8                 curDR = [i, j, 2*i];
9                 flag = false;
10             else curDR[3] = curDR[3] + i;
11          elseif flag = false then Exit-inner-loop;
12       end;
13       if (maxDR[3] < curDR[3]) and
            (maxDR[2] = 0 or (curDR[2]<= maxDR[2]) then
14          maxDR = curDR;
15 end
16 if ( maxDR[3] != 0 ) then
17    if (maxDR[2]+maxDR[3]-1 != size(Node.Children)) then
18       return {maxDR}∪
                IdentDRs(maxDR[2]+maxDR[3], Node, K, T)
19    else return {maxDR}
20 end;
21 return ∅;
```

Figure 9. Identifying data regions below a node.

The IdentDRs procedure is recursive (line 18). In each recursion, it extracts the next data region *maxDR* that covers the maximum number of children nodes. *maxDR* is described by three members (line 1), (1) the number of nodes in a combination, (2) the location of the start child node of the data region, and (3) the number of nodes involved in or covered by the data region. *curDR* is the current candidate data region being considered. String comparison results are stored in a data structure attached with each node. The value can be obtained by calling procedure Distance(*Node*, *i*, *j*) (which is just a table lookup, and thus it is not listed in the paper), where *i* represents *i*-combination, and *j* represents the *j*th child of *Node*. IdentDRs basically checks each combination (line 2) and each starting point (line 3). For each possibility, it finds the first continuous region with a set of generalized nodes (line 5 - line 12). Lines 13 and 14 update the maximum data region *maxDR*. The conditions (line 13) ensure that smaller generalized nodes are used unless the larger ones cover more (tag) nodes and starts no later than the smaller ones.

Finally, procedure UnCoveredDRs is given in Figure 10..

```
Procedure UnCoveredDRs(Node, Child)
1  for each data region DR in Node.DRs do
2     if Child in range DR[2] .. (DR[2] + DR[3] - 1) then
3        return null
4  return Child.DRs
```

Figure 10: The UnCoveredDRs procedure

Assume that the total number of nodes in the tag tree is *N*, the complexity of FindDRs is $O(NK^2)$. Since *K* is normally very small. Thus, the computation requirement of the algorithm is low.

Finally, it is important to note that the algorithm is able to find nested data records due to the bottom up data region evaluation.

3.3 Identify Data Records

After all data regions and their generalized nodes are found from a page, we are ready to identify the data records in each region. As noted earlier, a generalized node (a combination of tag nodes) may not be a data record containing the description of a single object because procedure UnCoveredDR reports higher level data regions. The actual data records may be at a lower level. That is, a generalized node may contain one or more data records.

Let us see an example (Figure 11). Figure 11 shows a data region that contains two table rows, row 1 and row 2. Row 1 and row 2 are generalized nodes as identified in Section 3.2. However, they are not individual data records. Each row actually contains two data records in the two table cells, which are descriptions of two objects. It is important that we report each cell (an object description) as a data record rather than each row.

row 1	Object 1	Object 2
row 2	Object 3	Object 4

Figure 11: Each row with more than one data record

To find data records from each generalized node in a data region, the following constraint is useful:

If a generalized node contains two or more data records, these data records must be similar in terms of their tag strings.

This constraint is clear because we assume that a data region contains descriptions of similar objects (i.e., similar data records).

Identifying data records from each generalized node in a data region is relatively easy because they are nodes (together with

their sub-trees) at the same level as the generalized node, or nodes at a lower level of the tag tree. Our experiments show that we only need to go down one level to check if data records are there. If not, the data records are at the same level as the generalized node. This is easily done based on the above constraint because all string comparisons have been done by procedure CombComp.

This step, however, needs heuristic knowledge of how people present data objects. Let a data region be *DR*. We have two cases:

1. Each generalized node *G* in *DR* consists of only one tag node (or component) in the tag tree. We go one level down the tag tree from *G* to check its children nodes (Figure 12).

   Procedure FindRecords-1(*G*)
   1 **If** all children nodes of *G* are similar
   2 AND *G* is not a data table row **then**
   3 each child node of *R* is a data record
   4 **else** *G* itself is a data record.

   Figure 12. Finding data records in a one-component generalized node

   Line 1 means that the tag strings of the sub-trees of the children nodes are similar. Figure 11 shows an example for line 3 (Figure 12), where a table row contains multiple objects. Each row is a generalized node, but not a data record. The data records are the cells (children nodes) of each row.

   In line 2, "*G* is not a data table" means that if *G* is a data table row, and its data region *DR* actually represents a *data table*. A data table is like a spreadsheet or a relational table, where all the cells in a row are similar (since we only consider tags) and each cell has only one piece of text or a number. Instead of reporting each cell as a data record, *G* should be reported as a data record. Figure 13 shows a data table example. Each cell is an attribute value of an object description.

row 1	attr1-v	attr1-v	attr1-v	attr1-v	Object 1
row 2	attr2-v	attr2-v	attr2-v	attr2-v	Object 2

   Figure 13: A data table example

   Figure 14 gives an example for the case in line 4 of Figure 12. Each row (*G*) is a data record in *DR*.

row 1	Object 1
row 2	Object 2

   Figure 14: A generalized node being a single data record

2. A generalized node *G* in *DR* consists of *n* tag nodes ($n > 1$) or components. We identify data records as follows:

   Procedure FindRecords-n(*G*)
   1 **If** the children nodes of each node in *G* are similar AND each node also has the same number of children **then**
   2 The corresponding children nodes of every node in *G* form a non-contiguous object description
   3 **else** *G* itself is a data record.

   Figure 15. Finding data records in an n-component generalized node

   The case in line 2 is discussed below. An example of the case in line 3 is given in Figure 16. Here, every two rows (a generalized node) form a data record. The cells (children nodes) of the first row in *G* are different and the second row in *G* has only one child (or cell).

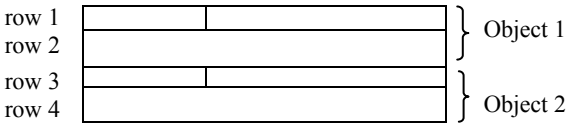

Figure 16: One object in multiple rows

Non-contiguous object descriptions

In some Web pages, the description of an object (a data record) is not in a contiguous segment of the HTML code. There are two main cases. Figure 17 shows an example of the first case. Objects are listed in two columns (can be more). Each object is also described in two table rows. One row lists the names of the two objects in two cells, and the next row lists the other pieces of information of the objects also in two cells. This results in the following sequence in the HTML code,

name 1, name 2, description 1, description 2, name 3, name 4, description 3, description 4.

We can see that different pieces of information of an object are not contiguous in the HTML source code.

row 1	name 1	name 2
row 2	description 1	description 2
row 3	name 3	name 4
row 4	description 3	description 4

Figure 17: Four objects with non-contiguous descriptions

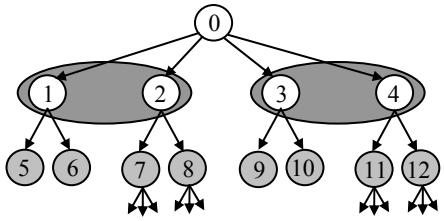

Figure 18: Tag tree of the objects in Figure 17.

Figure 18 shows the part of the tag tree. This data region has two generalized nodes. Each generalized node has two tag nodes, i.e., two rows (1 and 2, or 3 and 4). Since in line 1 of Figure 15, we already know that node 1 (row 1) has two similar children nodes, 5 and 6, and node 2 (row 2) also has two similar children nodes, 7 and 8. We simply group the corresponding children of nodes 1 and 2, i.e., to join nodes 5 and 7 to form one data record and join nodes 6 and 8 to form another data record. Likewise, we can find the data records under node 3 and node 4.

We now turn to the second case. In this case, we do not have row 3 and row 4, but only row 1 and row 2 in Figure 17. Row 1 will form a data region and row 2 will form another data region. We handle this case by detecting adjacent data regions and then check to see whether these data regions are similar. If they are not, we can join the corresponding nodes to give the final data records. We realize that this heuristic may not be safe because the two data regions may contain unrelated objects and should not be merged. However, the heuristic seems to work because people seldom put two types of objects right next to each other with no separators.

3.4 Data Records not in Data Regions

In some situations, some data records are not covered by any data region. Yet, they describe similar objects as data records in some

data regions. These situations occur due to various reasons. Figure 19 shows one example. Due to an odd number of objects, Object 5 will not be covered in a data region, which only includes row 1 and row 2, since row 3 is not sufficiently similar to row 2. Row 3 also will not form a data region itself because our algorithm in Section 3.2 requires that at least two nodes (with their sub-trees) with the same parent are similar.

row 1	Object 1	Object 2
row 2	Object 3	Object 4
row 3	Object 5	

Figure 19: An odd number of objects

Figure 20 gives this part of the HTML tag tree. Rows 1, 2, and 3 (r1, r2, r3) are at the same level. Rows 1 and 2 (two generalized nodes) form a data region, which does not include row 3. The data records are Objects 1 to 5 (i.e., O1, O2, O3, O4, and O5).

It is easy to find object O5 after finding O1 to O4. We can simply use the tag string of O4 (or any of the 4 objects or data records) to match each tag string of the children of the sibling nodes of r1 and r2. In this case, r3 is the only sibling node of r1 and r2. We will find that r3's only child O5 matches O4 well.

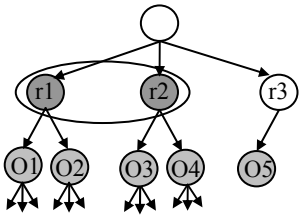

Figure 20: The tag tree for the objects in Figure 19

4. EXPERIMENT RESULTS

This section evaluates our system (MDR) that implements the proposed technique. We also compare it with two state-of-the-art existing systems, OMINI [2] (which is an improvement of [10]), and IEPAD [4]. Both systems are available on the Web (OMINI: http://disl.cc.gatech.edu/Omini/, IEPAD: http://140.115.155.99). In running both OMINI and IEPAD, we used their default settings. The experimental results are given in Table 1. Below, we first describe some experimental settings and then discuss the results.

Experiment Web pages: Our experimental Web pages include:

1. The set of 18 pages from the OMINI system at OMINI's Web site. Since most Web pages change frequently, three pages in OMINI did not contain any regularly structured data record when we performed our experiments. Thus, these 3 pages are not included in our experiments.
2. A large number of other pages from a wide range of domains, books, travel, software, auctions, jobs, electronic products, shopping, and search engine results (IEPAD is only evaluated using search engine results). These pages were provided by a colleague who was not involved in the project. We used all the pages provided by him except four (4) that do not use table tags to form data records. Note that we did not perform evaluation using many pages from a single site as was done in [2][4] because the pages in a single site are often similar. It is more useful to test pages from a large number of diverse sites.

Edit distance threshold: We used a number of training pages in building our systems and in selecting our default edit distance threshold. These pages are:

- Some notebook and book pages from www.amazon.com.
- Some notebook pages from www.hp.com.
- Some news pages and shopping pages from www.yahoo.com.
- First 4 pages in the OMINI site (the first 4 pages in Table 1).

These training pages were not used in testing our system except the four pages from OMINI, which are included for completeness of OMINI pages (the first 4 pages in Table 1).

Using these training pages, we select an edit distance threshold value of 0.3, which is set as the default value for our system. It is used in all our experiments.

Evaluation measures: We use the standard measure of precision and recall to evaluate the results from different systems. In order to give readers a clearer idea of the experiment results, we also describe the errors made by OMINI and IEPAD.

Outputs of OMINI, IEPAD and MDR: Given a page, OMINI first identifies the region that contains main objects, it then extracts these objects. IEPAD produces a set of extraction rules. One needs to try each rule to find the rules that extract those objects that one is interested in. This step is used because a page typically has a number of data regions, and the system will not know which one is of interest to the user. Our MDR system outputs each data region and its data records. The results of the systems are fairly easy to judge because one can visually compare their results with those data records in the actual Web pages.

Experimental results: We now discuss the results in Table 1.

Columns 1 and 2: Column 1 gives the id number of each experiment or Web page. The first 15 pages are from OMINI. Column 2 gives the URL of each page.

Column 3: It gives the number of data records contained in each page. These are those obvious data records of the page (e.g., product list, and search results). They do not include navigation areas, which can have regular patterns as well. Since OMINI tries to identify the main objects in the page, it does not give navigation areas or other smaller regions. However, both IEPAD and MDR are able to report such regions if they exist. Although it may be possible to remove them by using some heuristics, we choose to keep them because they may be useful to some users. We do not use these areas in our comparison in Table 1 because OMINI does not find them.

Column 4: It shows the number of data records found by our system MDR. It gives perfect results for all pages except one, page 44. For this page, MDR lost one data record.

Columns 5, 6 and 7: Column 5 shows the number of correct data records found by OMINI. Column 6 gives the total number of data records (which may not be correct) found by OMINI. Column 7 gives some remarks about the problems with OMINI.

Columns 8, 9 and 10: These three columns give the three corresponding results of IEPAD for each page. IEPAD often produces a large number of extraction rules (see below), which extract information from the data records. We tried every rule and present here the best result in column 8 for each page (the results may come from more than one rule).

The last three rows of the table give the total number of data records in each column, the recall and the precision of each system. The precision and the recall are computed based on the total number of data records found in all the pages by each system and the total number of data records that exist in all the pages.

Before further discussing the experimental results, we first

	URL	Obj.	MDR	OMINI			IEPAD		
				corr.	found	remark	corr.	found	remark
1	http://www.bookbuyer.com	4	4	2	4	all-miss-info	4	5	all-miss-info
2	http://www.powells.com	4	4	4	5	1 err.	0	0	none-found
3	http://www.barnesandnoble.com	4	4	0	5	all-in-1 with noise	0	7	none-found
4	http://www.codysbooks.com	6	6	0	3	all-in-1 with noise	6	7	all-miss-info
5	http://www.bookpool.com	25	25	25	26	1 err.	0	12	none-found
6	http://www.borders.com	25	25	25	25		14	14	miss 9 objects
7	http://www.alphabetstreet.infront.co.uk	10	10	0	8	none-found	10	10	
8	http://www.ebay.com	7	7	0	1	all-in-1 with noise	7	7	
9	http://auctions.yahoo.com	6	6	0	3	all-in-1 with noise	6	7	1 err, all-miss-info
10	http://www.drugstore.com	8	8	0	0	none-found	7	7	miss 1 object
11	http://www.epicurious.com	3	3	0	0	none-found	0	12	none-found
12	http://www.mymenus.com	6	6	0	2	none-found	0	6	none-found
13	http://www.cooking.com	11	11	0	3	none-found	9	14	2 split into 5
14	http://www.eve.com/	9	9	0	2	all-in-1 with noise	9	9	
15	http://www.etoys.com	5	5	4	4	miss 1 object	5	5	all-miss-info
16	http://www.tourvacationstogo.com	70	70	70	70		0	5	none-found
17	http://www.tourturkey.com	6	6	5	6	1 err.	0	0	none-found
18	http://www.asiatravel.com	18	18	0	4	all-in-1 with noise	15	15	all-miss-info
19	http://www.mapquest.com	2	2	0	4	all-in-1 with noise	0	5	none-found
20	http://www.travelocity.com	5	5	0	7	all-in-1 with noise	5	5	
21	http://www.ubid.com	22	22	0	2	all-in-1 with noise	13	27	extra 14 wrong
22	http://www.grijns.net/	62	62	0	14	all-in-12 with noise	5	5	miss 57 object
23	http://journeys.20m.com	8	8	3	5	miss 5 objects	8	10	extra 2 wrong
24	http://www.softwareoutlet.com	9	9	0	3	all-in-1 with noise	8	9	extra 1 wrong
25	http://qualityinks.com/index.php	66	66	0	22	all-in-22 with noise	0	0	none-found
26	http://www.nothingbutsoftware.com	17	17	14	17	3-in-1 with noise	14	14	
27	http://www.newegg.com	12	12	0	5	6-in-3 with noise	6	6	
28	http://chemstore.cambridgesoft.com	5	5	5	5		5	5	
29	http://www.godaddy.com	4	4	0	2	all-in-1 with noise	4	11	7 err, all-miss-info
30	http://www.compusa.com	8	8	0	3	all-in-1 with noise	8	8	
31	http://www.radioshack.com	9	9	3	4	miss 6 objects	9	9	
32	http://www.earlemu.com	4	4	4	5		0	0	none-found
33	http://www.kadybooks.com	20	20	0	50	split into 50	10	10	
34	http://www.kidsfootlocker.com	9	9	0	1	all-in-1 with noise	9	9	2-miss-info
35	http://shop.lycos.com	13	13	0	2	all-in-1 with noise	5	5	
36	http://thenew.hp.com	4	4	0	5	all-in-1 with noise	0	10	split into 10
37	http://www.dell.com	5	5	5	5		5	5	
38	http://www.circuitcity.com	4	4	0	0	none-found	0	6	none-found
39	http://www.overstock.com	3	3	3	3		0	8	split into 8
40	http://www.kodak.com	3	3	0	6	none-found	0	6	none-found
41	http://www.flipdog.com	25	25	25	28	3 err.	0	0	none-found
42	http://www.summerjobs.com	20	20	0	3	none-found	0	7	none-found
43	http://search.lycos.com	10	10	0	8	all-in-2 with noise	10	10	
44	http://www.northernlight.com	10	9	10	11	1 err.	10	10	all-miss-info
45	http://www.coolhits.com	20	20	20	21	1 err.	0	0	none-found
46	http://www.mamma.com	15	15	15	20	5 err.	15	15	
	Total	621	620	242	432		241	357	
	Recall		99.8%	39%			39%		
	Precision		100%	56%			67%		

Table 1: Experimental results

explain the problem descriptions used in the table:

all-in-n (*m-in-n*) *with noise*: It means that all (or *m*) data records are identified as *n* data records (*m* > *n*). For example, "all-in-1" means that the system is unable to separate the data records but identified them together as one. "with noise" means that some items in the data records that are not part of the data records.

n-err.: It means that *n* (extra) incorrect data records are found.

miss n objects: *n* correct data records are not identified.

split into n: This means that the correct data records are split into *n* smaller ones, i.e., a data record is not found as one but a few.

none-found: None of the correct data records is found.

all-miss-info (*n-miss-info*): This means that all (or *n*) data records found by the system with some parts missing.

The following summarizes the experimental results in Table 1.

1. Our system MDR is able to give perfect results for every page except for page 44. For this page, one data record is not found

because it is too dissimilar to its neighbors. From the last two rows, we can see that MDR has a 99.8% recall and 100% precision, which are remarkable. Both OMINI and IEPAD only have a recall of 39%. In the recall computation, those data records with missing information are considered correct for both OMINI and IEPAD (MDR does not lose any information). If such data records are not considered correct, the recall of OMINI is 38.3%, and the recall of IEPAD is reduced to only 29%. The precision values of both systems are also low, 56% for OMINI and 67% for IEPAD.

2. In columns 7 and 10, those cells that do not contain remarks show that the system finds all data records correctly. We can see that OMINI only gives perfect results for 6 pages out of 46, while IEPAD gives perfect results for only 14 pages. Our MDR system is able to give perfect results for all the pages except page 44.
3. We observed in our experiments that IEPAD and OMINI tend to work well only when the page is simple in the sense that there are a large number of similar data records, and there is little other information in the page. However, most Web pages, e.g., product pages, are often more complex, and they may not contain a large number of products.
4. In column 7, we see that in 20 pages, many data records are identified together as one by OMINI and also include some noise items. This clearly shows the serious weakness of OMINI's tag based heuristic approach.
5. IEPAD does poorly when data records follow similar patterns but not exactly the same pattern. We believe that this problem with IEPAD is due to its use of Patricia tree, which only finds exact repetitive patterns. Its sequence alignment algorithm could not remedy the situation in many cases. Even when the alignment algorithm works, it tends to lose information.

   Another problem with IEPAD is that it often generates a large number of extraction rules. For our 46 pages in Table 1, IEPAD generates more than 20 rules per page for 5 pages, and more than 10 rules per page for 11 pages. There are often many rules extracting the same piece of information with minor variations. This shows that IEPAD is unable to decide data record boundaries, but provides some possible ones. The user needs to try every rule to see which gives the best result. This is time consuming. In contrast, MDR does not have this problem. It identifies the boundaries of the objects very well.
6. Both OMINI and IEPAD are unable to find non-contiguous structures. Such cases occur in pages 1, 4 and 36.

From our experiments, we can conclude that MDR is very accurate and is dramatically better than OMINI and IEPAD.

Execution time of MDR: Section 3.2 analyzed the complexity of our algorithm, which indicated that the algorithm is efficient. Our system is implemented in Visual C++. All the experiments were conducted on a Pentium 4 1.4GHz PC with 512 MB RAM. The execution time for every page is always less than 0.5 second.

5. CONCLUSIONS

In this paper, we proposed a novel and effective technique to mine data records in a Web page. Our algorithm is based on two important observations about data records on the Web and a string matching algorithm. It is automatic and thus requires no manual effort. In addition, our algorithm is able to discover non-contiguous data records, which cannot be handled by existing techniques. Experimental results show that our new method is extremely accurate. It outperforms the two existing state-of-the-art systems dramatically. In our future work, we plan to find data records that are not formed by HTML table related tags.

Acknowledgement: We thank Chris Livadas for identifying some errors in the original pseudo-code.

6. REFERENCES

[1] Baeza-Yates, R. "Algorithms for string matching: A survey." *ACM SIGIR Forum*, 23(3-4):34--58, 1989

[2] Buttler, D., Liu, L., Pu, C. "A fully automated extraction system for the World Wide Web." *IEEE ICDCS-21*, 2001.

[3] Chakrabarti, S . *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan Kaufmann Publishers, 2002.

[4] Chang, C-H., Lui, S-L. "IEPAD: Information extraction based on pattern discovery." *WWW-10*, 2001.

[5] Chen, H.-H., Tsai, S.-C., and Tsai, J.-H. "Mining tables from large scale html texts." *COLING-00*, 2000.

[6] Cohen, W., Hurst, M., and Jensen, L. "A flexible learning system for wrapping tables and lists in HTML documents. *WWW-2002*, 2002.

[7] Cohen, W., McCallum, A., Quass, D. Learning to Understand the Web. *IEEE Data Engineering Bulletin*, Vol. 23, No. 3. pp. 17-24, 2000.

[8] Craven, M., DiPasquo, D., Freitag, D., McCallum, A., Mitchell, T., Nigam, K. & Slattery, S. "Learning to construct knowledge base from the World Wide Web" *Artificial Intelligence*, 118(1-2), 2000.

[9] Doorenbos, R., Etzioni, O., Weld, D. "A scalable comparison shopping agent for the World Wide Web." *Agents-97*, 1997.

[10] Embley, D., Jiang, Y and Ng, Y. "Record-boundary discovery in Web documents." *SIGMOD-99*, 1999.

[11] Gusfield, D. *Algorithms on strings, tree, and sequence*, Cambridge. 1997.

[12] Hammer, J. Garcia-Molina, H, and Cho, J. Aranha. R and Crespo, A. "Extracting semi-structured information from the Web." *Proceedings of the Workshop on Management of Semi-structured Data*, 1997.

[13] Han, J. and Chang, K. C.-C. "Data mining for Web intelligence." *IEEE Computer*, Nov. 2002.

[14] Hsu, C.-N. and Dung, M.-T. "Generating finite-state transducers for semi-structured data extraction from the Web." *Information Systems*. 23(8): 521-538, 1998.

[15] Kleinberg, J. "Authoritative sources in a hyperlinked environment." *ACM-SIAM Sym. on Discrete Algo.*, 1998.

[16] Knoblock, A. et al., Eds. *Proceedings of the 1998 Workshop on AI and Information Integration*. 1998.

[17] Kushmerick, N.; Weld, D. and Doorenbos, R. "Wrapper induction for information extraction." *IJCAI-97*, 1997.

[18] Kushmerick, N. "Wrapper induction: efficiency and expressiveness." *Artificial Intelligence*, 118:15-68, 2000.

[19] Lerman, K. Knoblock, C., and Minton, S. "Automatic data extraction from lists and tables in web sources." *IJCAI-01 Workshop on Adaptive Text Extraction and Mining*, 2001.

[20] Liu, L., Pu, C. and Han, W. "XWrap: an XML-enabled wrapper construction system for Web information sources." *ICDE-2000*, 2000.

[21] Muslea, I., Minton, S. and Knoblock, C. "A hierarchical approach to wrapper induction." *Agents-99*, 1999.

[22] Sahuget, A. and Azavant, F. "WysiWyg Web wrapper Factory (W4F)." *WWW8*, 1999.

[23] Soderland, S. "Learning to extract text-based information from the World Wide Web." *KDD-1997*, 1997.

[24] Wang, Y., Hu, J. "A machine learning based approach for table detection on the Web." *WWW-2002*, 2002.