Mining Web Pages for Data Records

Bing Liu, Robert Grossman, and Yanhong Zhai, *University of Illinois at Chicago*

Data mining to extract information from Web pages can help provide value-added services. The MDR (mining data records) system exploits Web page structure and uses a string-matching algorithm to mine contiguous and noncontiguous data records.

Much 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—for example, products or services. These objects can be regarded as database records displayed in Web pages with regular patterns. We also call them *data records*. By mining data records in Web pages, you can extract and integrate information from multiple sources to provide value-added services, such as customizable Web information gathering, comparative shopping, and metasearching.

Current approaches to mining Web data include supervised learning and automatic techniques. Supervised-learning systems use manually labeled training data and thus require substantial human effort. Current automatic techniques perform poorly (see the "Related Work" sidebar). In addition, current approaches assume that a data record's relevant items are contained in a contiguous segment of the HTML code, which might not be the case. For example, the descriptions of two objects in a page's HTML source can be noncontiguous—for instance, Part 1 of Object 1, Part 1 of Object 2, Part 2 of Object 1, Part 2 of Object 2—although they appear contiguous when displayed on a browser. Few researchers have exploited the nested structures of HTML tags and the layout of data records.

To solve this problem, we propose the MDR (mining data records) system. MDR is an automatic technique that finds all data records formed by table- and form-related HTML tags—that is, table, form, tr, td, and so on. Extensive evaluation using numerous Web pages from diverse domains shows that MDR produces dramatically better results than some existing systems.

Data records on the Web

We base our technique on two key observations of the layout of data records in Web pages.

First, a group of data records containing a set of similar objects typically appear in a contiguous region of a page—the *data record region* or *data region* for short—and are formatted with similar HTML tags. For example, Figure 1a shows two notebooks that appear in one contiguous region of a Web page and use almost the same sequence of tags. If we regard a page's tags as a long string, we can use string matching to find similar substrings, which might represent similar objects or data records.

With this approach, however, the computation is prohibitive because a data record can start from any tag and end at any other subsequent tag. Data records in a set typically vary in length in terms of their tag strings because they contain slightly different pieces of information.

The nested structure of HTML tags in a Web page naturally forms a *tag tree*. Our second observation is that a tag tree reflects a group of similar data records in a specific region by placing the records under one parent node, although we don't know which parent. For example, Figure 1b shows the tag tree for the page in Figure 1a (some details are omitted). Each notebook (a data record) in Figure 1a is wrapped in five tr nodes with their subtrees under the same parent node tbody (see Figure 1b). In other words, some child subtrees of the same parent node constitute a set of similar data records.

In addition, data records rarely start in the middle of a child subtree and end in the middle of another child subtree. Instead, a data record starts from the beginning of a child subtree and ends at the end of the same or a later child subtree. For example, in Figure

Related Work

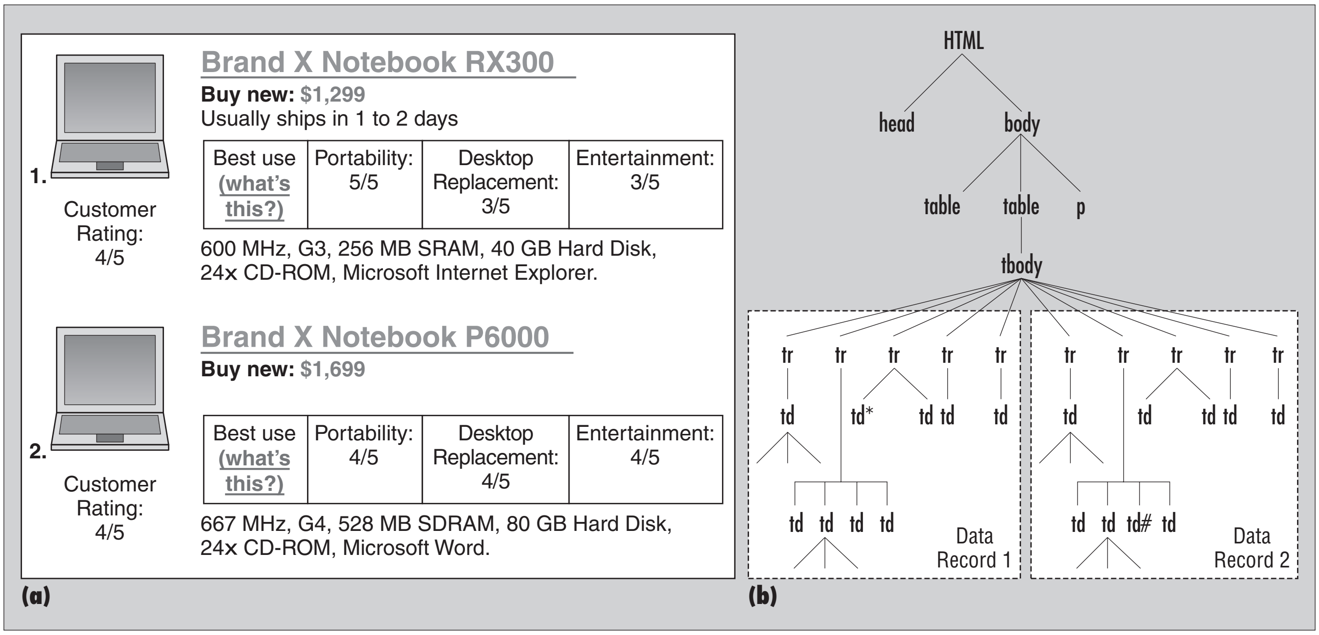
Researchers have developed several approaches for mining data records from Web pages. David Embley, Yuan Jiang, and Yiu-Kai Ng use a set of heuristics and a manually constructed domain ontology.[1] David Buttler, Ling Liu, and Calton Pu extend this approach in Omini (Object Mining and Extraction System, http://disl.cc.gatech.edu/Omini), which uses additional heuristics based on HTML tags but no domain knowledge.[2] This approach performs poorly, as our main article shows.

Chia-Hui Chang and Shao-Chen Lui propose IEPAD (*I*nformation *E*xtraction Based on *Pa*ttern *D*iscovery), an automatic method that uses sequence alignment to find patterns representing a set of data records.[3] Because this method's sequence matching is limited, it also performs poorly. Kristina Lerman, Craig Knoblock, and Steven Minton use clustering and grammar induction of regular languages but report unsatisfactory results.[4] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo compare two pages to find patterns for data extraction.[5] However, their method needs to start with a set of similar pages. By working on individual pages, our technique avoids this limitation.

Other related research is mainly in *wrapper induction*.[6] A wrapper is a program that extracts data from a Web site and puts it in a database. A wrapper induction system learns extraction rules using manually labeled training examples. Existing wrapper induction systems include WIEN (*W*rapper *I*nduction *En*vironment),[7] Softmealy,[8] Stalker,[9] and WL^2.[10] However, rather than mining data records, wrapper induction only extracts certain pieces of information on the basis of user-labeled items.

References

1. D. Embley, Y. Jiang, and Y. Ng, "Record-Boundary Discovery in Web Documents," *Proc. ACM Int'l Conf. Management of Data* (SIGMOD 99), ACM Press, 1999, pp. 467–478.
2. D. Buttler, L. Liu, and C. Pu, "A Fully Automated Extraction System for the World Wide Web," *Proc. 21st Int'l Conf. Distributed Computing Systems* (ICDCS 01), IEEE CS Press, 2001, pp. 361–370.
3. C.-H. Chang and S.-L. Lui, "IEPAD: Information Extraction Based on Pattern Discovery," *Proc. 10th Int'l Conf. World Wide Web* (WWW 01), ACM Press, 2001, pp. 681–688.
4. K. Lerman, C. Knoblock, and S. Minton, "Automatic Data Extraction from Lists and Tables in Web Sources," *Proc. IJCAI 2001 Workshop Adaptive Text Extraction and Mining*, 2001; www.isi.edu/info-agents/papers/lerman01-atem.pdf.
5. V. Crescenzi, G. Mecca, and P. Merialdo, "RoadRunner: Towards Automatic Data Extraction from Large Web Sites," *Proc. 27th Int'l Conf. Very Large Data Bases* (VLDB 01), Morgan Kaufmann, 2001, pp. 109–118.
6. C. Knoblock and A. Levy, eds., *Proc. 1998 Workshop AI and Information Integration*, AAAI Press, 1998; www.isi.edu/ariadne/aiii98-wkshp/proceedings.html.
7. N. Kushmerick, D. Weld, and R. Doorenbos, "Wrapper Induction for Information Extraction," *Proc. 15th Int'l Joint Conf. Artificial Intelligence* (IJCAI 97), Morgan Kaufmann, 1997, pp. 729–735.
8. C.-N. Hsu and M.-T. Dung, "Generating Finite-State Transducers for Semi-structured Data Extraction from the Web," *Information Systems*, vol. 23, no. 8, 1998, pp. 521–538.
9. I. Muslea, S. Minton, and C. Knoblock, "A Hierarchical Approach to Wrapper Induction," *Proc. 3rd Int'l Conf. Autonomous Agents* (Agents 99), ACM Press, 1999, pp. 190–197.
10. W. Cohen, M. Hurst, and L. Jensen, "A Flexible Learning System for Wrapping Tables and Lists in HTML Documents," *Proc. 11th Int'l Conf. World Wide Web* (WWW 02), ACM Press, 2002, pp. 232–241.



1. Customer Rating: 4/5

Brand X Notebook RX300

Buy new: $1,299

Usually ships in 1 to 2 days

Best use (what's this?)	Portability: 5/5	Desktop Replacement: 3/5	Entertainment: 3/5

600 MHz, G3, 256 MB SRAM, 40 GB Hard Disk, 24x CD-ROM, Microsoft Internet Explorer.

2. Customer Rating: 4/5

Brand X Notebook P6000

Buy new: $1,699

Best use (what's this?)	Portability: 4/5	Desktop Replacement: 4/5	Entertainment: 4/5

667 MHz, G4, 528 MB SDRAM, 80 GB Hard Disk, 24x CD-ROM, Microsoft Word.

(a)

(b)

Figure 1. A data record layout: (a) a typical product Web page showing two notebook computers and (b) the tag tree showing the layout of the HTML tags for the page.

1b, it's unlikely that a data record would start at td* and end at td#. This observation makes designing an efficient algorithm to mine data records possible.

Our proposed technique

Given a Web page, our technique has three steps:

1. We build an HTML tag tree of the page.
2. We mine all data regions in the page using our observations on data record layout and the *edit distance string-matching algorithm*[1] (which we describe later). For example, in Figure 1b, we first find the single data region below the node tbody.
3. We identify data records from each data region. For example, in Figure 1b, we find Data Records 1 and 2 in the data region below tbody.

The nested structure of HTML tags makes the first step fairly straightforward, so we don't discuss it further. Instead, we focus on the last two steps.

Mining data regions

This step mines every data region in a page containing similar data records. Instead of mining data records directly, which is difficult, we first mine *generalized nodes*. A sequence of adjacent generalized nodes forms a data region.

A generalized node (or *node combination*) of length r consists of r ($r \geq 1$) nodes in the HTML tag tree that are adjacent and have the same parent. We introduce this concept to demonstrate that an object (or data record) can be contained in a few sibling *tag nodes*—that is, nodes in the HTML tag tree—rather than one node. In Figure 1, for example, each notebook is contained in five table rows (or tr nodes).

A data region is a collection of two or more generalized nodes with these properties:

- They have the same parent.
- They have the same length (that is, the same number of child nodes in the tag tree).
- They're adjacent.
- The normalized *edit distance* between them is less than a fixed threshold. (The edit distance of two strings, s_1 and s_2, is the minimum number of *point mutations* required to change s_1 into s_2. A point mutation involves changing a letter, inserting a letter, or deleting a letter.)

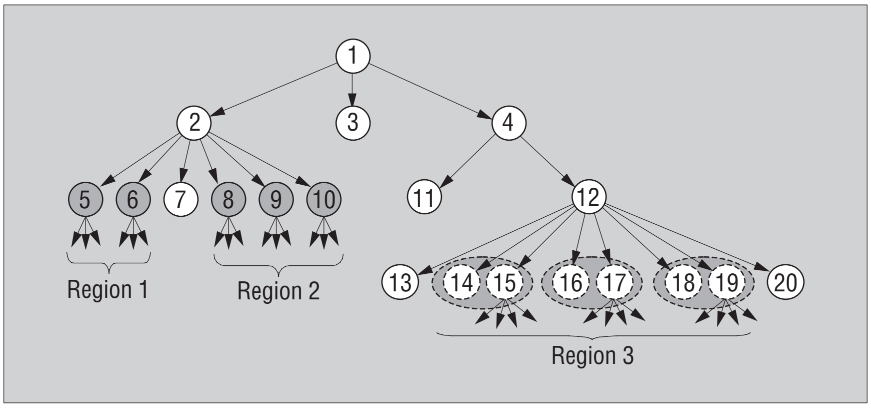


Figure 2. Generalized node and data region variations. Shaded areas in the tag tree are generalized nodes.

For example, in Figure 1b, we can form two generalized nodes, the first consisting of the first five child tr nodes of tbody, and the second consisting of the next five child tr nodes. Although the generalized nodes in a data region have the same length, the lower-level subtree nodes can be quite different. Thus, they can capture a wide variety of regularly structured objects.

The artificial tag tree in Figure 2 further illustrates the different kinds of generalized nodes and data regions. For convenience, we use ID numbers to denote tag nodes. Nodes 5 and 6 are generalized nodes of length 1; together, they define Region 1 if the edit distance condition is satisfied. Nodes 8, 9, and 10 are also generalized nodes of length 1; together, they define Region 2. The node pairs (14, 15), (16, 17), and (18, 19) are generalized nodes of length 2. Together, they define Region 3.

Our definitions are robust in practice, as our experiments show. Our key assumption is that nodes forming a data region are from the same parent, which is realistic. In addition, a generalized node might not represent a final data record; rather, we use it to find the final data records.

Comparing generalized nodes. To find each data region, the mining algorithm must answer two questions:

- Where does the first generalized node of a data region start? For example, in Region 2 of Figure 2, the generalized node starts at Node 8.
- How many tag nodes (components) are in the generalized nodes in each data region? In Region 2 of Figure 2, for example, each generalized node has one tag node.

Let K be the maximum number of tag nodes in a generalized node. K is typically small (fewer than 10). To answer the first question, we try to find a data region starting from each node sequentially. To answer the second question, we try 1-node, 2-node, …, K-node combinations. That is, we start from each node and perform all one-node string comparisons, all two-node string comparisons, and so on. We then use the comparison results to identify each data region. The number of comparisons isn't large, for two reasons:

- We compare only the child nodes of a parent node. For example, in Figure 2, we don't compare Nodes 8 and 13.
- Some comparisons performed for earlier nodes are the same as for later nodes.

Figure 3 illustrates the comparison. The figure has 10 nodes below a parent node p. We start from each node and perform string comparisons for all possible combinations of component nodes. In this example, a generalized node can have a maximum of three components.

Starting from Node 1, we compute these 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)

The string comparison (1, 2) means we compare Node 1's tag string with Node 2's tag string. A node's tag string includes all the tags in the node's subtree. For example, in Figure 2, the tag string for the second tr node below tbody is <tr td td … td td>, where "…"

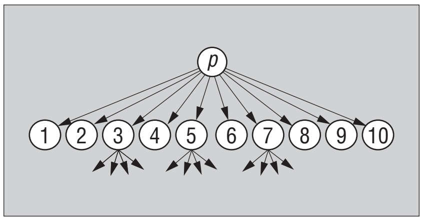


Figure 3. Comparison and combination. We perform different string comparisons (for example, one-node and two-node string comparisons) to get different combinations of tag nodes.

```
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);

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 + j;
```

Figure 4. The structure comparison algorithm (MDR). The CombComp procedure performs string comparisons using variations of the child subtrees.

denotes the substring of the subtree below the second td node. The tag string for the third tr node below tbody is <tr td td>.

The comparison (1-2, 3-4) means we compare the combined tag string of Nodes 1 and 2 with that of Nodes 3 and 4.

Starting from Node 2, we compute only

- (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 don't need to do one-node comparisons because we did them when we started from Node 1.

Starting from Node 3, we only need to compute one string comparison: (3-4-5, 6-7-8). Here, we don't need to do one- or two-node comparisons because we've already done them.

We don't need to start from any other nodes after Node 3 because we've performed all the computations. Proving that the process is complete is fairly easy. We omit the proof here because of space limitations.

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

Line 1 of CombComp starts from each node of *NodeList*. The procedure need only perform comparisons up to the Kth node. Line 2 compares different combinations of nodes, from i-component combinations to K-component combinations. Line 3 tests whether at least one pair of combinations exists. If not, no comparison is needed. Lines 4 to 8 perform string comparisons of various combinations by calling procedure EditDist, which uses the edit distance algorithm.[1]

Assume n number of elements in *NodeList*. Without considering the edit distance comparison, CombComp's time complexity is $O(nK)$. Because K is normally small, we consider the algorithm linear in n. If the total number of nodes in the tag tree is N, the complexity of MDR is $O(NK)$ without considering string comparison.

Determining data regions. When all string comparisons are complete, we identify each data region by finding its generalized nodes using the procedure FindDRs. For example, the page in Figure 5 contains eight data records. FindDRs reports each row as a generalized node, and the entire area (the dash-lined box) as a data region.

FindDRs uses the string comparison results at each parent node to find similar child node combinations. It uses these combinations to obtain candidate generalized nodes and data regions of the parent node. Two main issues affect the final decisions.

First, if a higher-level data region covers a lower-level data region—that is, if a lower-level data region is within a higher-level region—we report the higher-level data region and its generalized nodes. For example, in Figure 5, at a low level, Cells 1 and 2 are candidate generalized nodes, and together they form a candidate data region (Row 1). However, the data region that includes all four rows at a higher level covers them. In this case, we only report each row as a generalized node. We take this approach to avoid situations in which many very low-level nodes (with very small subtrees) are similar but don't represent true data records.

The second issue involves similar strings. If the strings in a set $s_1, s_2, s_3, \ldots, s_n$ are similar, a combination of any of the strings is similar to another combination of the same number. So, we report only the smallest generalized nodes covering a data region, which helps us find the final data records later. In Figure 5, we report each row as a generalized node rather than a combination of two rows (Rows 1 and 2 and Rows 3 and 4).

Figure 6 gives the FindDRs procedure. In FindDRs, T is the edit distance threshold, *Node* is any node, and K is the maximum number of tag nodes in a generalized node (we use 10 in our experiments). *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 of FindDRs is the same as line 1 in MDR (see Figure 4). FindDRs traverses the tag tree depth-first, performing one function at each node as it goes down the tree (line 2), and another when it backs up before going down another branch of the tree (line 6).

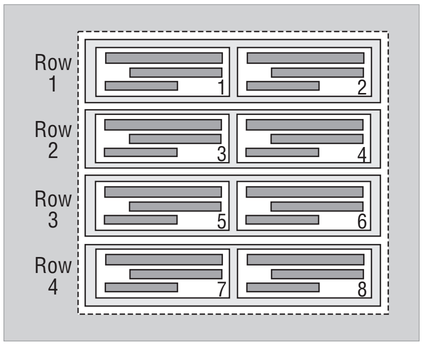


Figure 5. A possible configuration of data records. The FindDRs procedure reports each row as a generalized node; the dash-lined box represents a data region.

As FindDRs goes down, it identifies all the data regions at each node using the procedure IdentDRs. These aren't the final data regions of the Web page, but only candidates. As FindDRs goes back up, it checks whether the parent-level data regions in *Node.DRs* cover the child-level data regions and discards the covered child-level data regions. It stores child-level data regions that aren't in *Child.DRs* in *tempDRs* (line 6). After the algorithm processes all of *Node*'s child nodes, *Node.DRs* ∪ *tempDRs* gives the current data regions discovered from the subtree starting from *Node* (line 7).

The procedure IdentDRs (see Figure 6) uses the distance values for all possible child node combinations computed in the previous step as well as the threshold T to find *Node*'s data regions. That is, it must decide which combinations represent generalized nodes and where each data region begins and ends.

IdentDRs is recursive, as line 16 shows, and ensures that the algorithm identifies the smallest generalized nodes. In each recursion, IdentDRs extracts the next data region *maxDR* that covers the maximum number of children nodes. Three elements describe *maxDR* (line 1):

- The number of nodes in a combination
- The location of the data region's start child node
- The number of nodes involved in or covered by the data region

```
Algorithm FindDRs(Node, K, T)
1  if TreeDepth(Node) => 3 then
2      Node.DRs = IdentDRs(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

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 <= 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         if (maxDR[3] < curDR[3]) and (maxDR[2] = 0 or (curDR[2] <= maxDR[2]) then
13             maxDR = curDR;
14 if (maxDR[3] != 0 ) and
15     (maxDR[2] + maxDR[3] – 1 != size(Node.Children)) then
16     return {maxDR} ∪ IdentDRs(maxDR[2] + maxDR[3], Node, K, T)
17 return ∅;

UnCoveredDRs(Node, Child)
1  tempDiffDRs = ∅;
2  for each data region DR in Child.DRs do
3      if DR not covered by any region in Node.DRs then
4          tempDiffDRs = tempDiffDRs ∪ {DR}
5  return tempDiffDRs
```

Figure 6. The find all data regions (FindDRs) algorithm. FindDRS uses the procedure IdentDRs to identify data regions in the Web page, and UnCoveredDRs to identify data regions not covered by Node's data regions.

curDR is the current candidate data region that IdentDRs is considering. A data structure attached to each node stores the string comparison results. We can get the value by calling procedure Distance(*Node, i, j*) (which is just a table lookup and thus isn't listed in the article), where i represents i-combinations, and j represents Node's jth child. Basically, IdentDRs checks each combination (line 2) and each starting point (line 3). For each possibility, the procedure finds the first continuous region with a set of generalized nodes (lines 5–11). Lines 12 and 13 update *maxDR*. The conditions (line 12) ensure that the algorithm uses smaller generalized nodes unless the larger nodes cover more (tag) nodes and start no later than the smaller nodes.

Finally, the procedure UnCoveredDRs (see Figure 6) identifies data regions not covered by Node's data regions. *tempDiffDRs* stores the data regions in *Child.DRs* that other data regions in Node don't cover.

If the total number of nodes in the tag tree is N, the complexity of FindDRs is $O(NK^2)$. Because K is normally small, the algorithm's computation requirement is low.

Identifying data records

As we noted earlier, a generalized node might not be a data record describing a single object because UnCoveredDRs reports higher-level data regions. The actual data records might be at a lower level. That is, a generalized node might contain one or more data records.

Figure 7a shows a data region containing two table rows—Rows 1 and 2—that are generalized nodes but aren't individual data records. Each row actually contains two data records in the two table cells, which describe two objects.

To find data records from each generalized node in a data region, we use the constraint, "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 similar objects or data records.

Identifying data records from each generalized node in a data region is relatively simple because CombComp has performed all string comparisons. This data record identification step, however, requires heuristic knowledge of how people present data objects. Given a data region DR, we have two cases.

First, 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 child nodes:

```
Procedure FindRecords-1(G)
1  If all child 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.
```

Line 1 means that the child node subtrees have similar tag strings. Figure 7a illustrates line 3, in which a table row contains multiple objects. Each row is a generalized node but not a data record. The data records are the cells (child nodes) in each row.

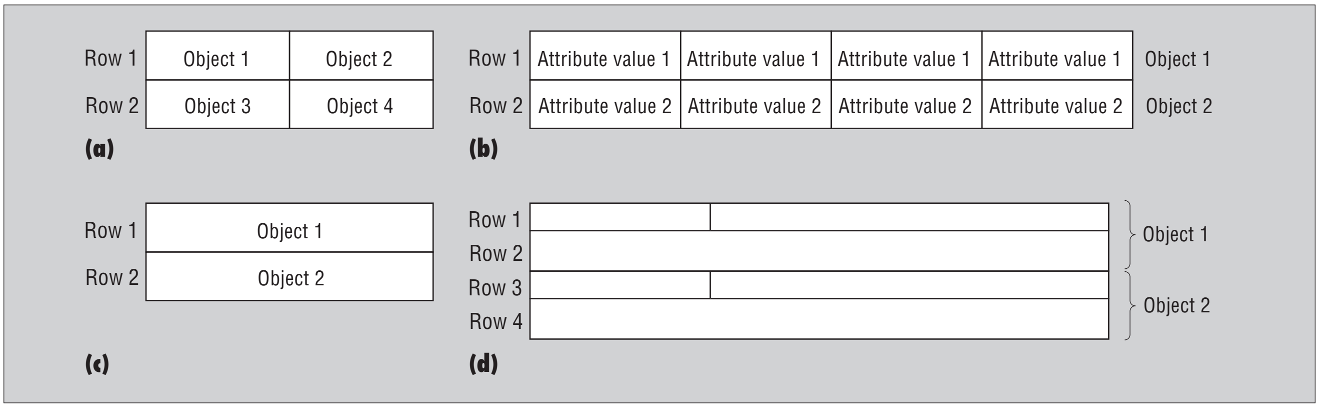


Figure 7. Example organizations of data records: (a) a table in which each row has more than one data record; (b) a data table; (c) a generalized node as a single data record; (d) an object in multiple rows.

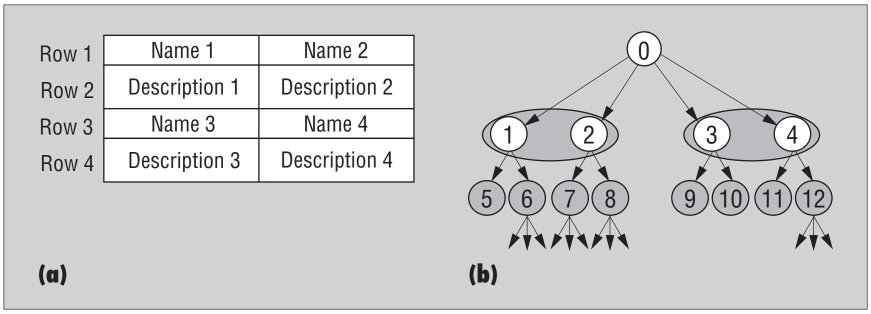


Figure 8. Four objects with noncontiguous descriptions: (a) a data table that lists data records by column and by row and (b) the tag tree for (a) in which the data region has two generalized nodes.

In line 2, "*G* is not a data table" means that if *G* is a data table row, its data region *DR* actually represents a data table. A data table, illustrated in Figure 7b, is like a spreadsheet or a relational table, in which all the cells in a row are similar (because 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, we report *G* as a data record. Each cell is an attribute value of an object description. Figures 7b and 7c are examples of the case in line 4. Each row (*G*) is a data record in *DR*.

In the second case, 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 child nodes of each node in G are similar
   AND each node has the same number of children then
2      The corresponding child nodes of every node in G
       form a noncontiguous object description
3  else G itself is a data record.
```

We discuss the case in line 2 in the next section. Figure 7d is also an example of the case in line 3. Here, every two rows (a generalized node) form a data record. The cells (child nodes) of the first row in *G* are different, and the second row in *G* has only one child (or cell).

Noncontiguous object descriptions. In some Web pages, an object's description (a data record) isn't in a contiguous segment of the HTML code. There are two main cases. In Figure 8a, for example, objects are listed in two columns (although there can be more). Two table rows also describe each object. 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.

Different pieces of information about an object might be noncontiguous in the HTML source code. Figure 8b shows part of a tag tree in which the data region has two generalized nodes. Each generalized node has two tag nodes—that is, two rows (1 and 2 or 3 and 4). Line 1 of procedure FindRecords-1 tells us that node 1 (Row 1) has two similar child nodes (5 and 6) and that Node 2 (Row 2) also has two similar children nodes (7 and 8). We therefore simply group the corresponding children of Nodes 1 and 2—that is, we join Nodes 5 and 7 to form one data record and Nodes 6 and 8 to form another. Likewise, we can find the data records under Nodes 3 and 4.

In the second case, we don't have Rows 3 and 4 (Figure 8a), but only Rows 1 and 2. Row 1 forms one data region and Row 2 forms another. We handle this case by detecting adjacent data regions and determining whether they're similar. If they aren't, we can join the corresponding nodes to give the final data records.

Data records not in any data regions. In some situations, some data records aren't covered by any data region, yet they are similar to data records in some data regions. In Figure 9a, for example, where there's an odd number of objects, Object 5 won't be covered by a data region because Row 3 isn't sufficiently similar to Row 2.

Figure 9b shows part of an HTML tag tree in which Rows 1, 2, and 3 (r1, r2, r3) are at the same level, and Rows 1 and 2 (two generalized nodes) form a data region that doesn't include Row 3. The data records are Objects 1 to 5 (O1, O2, O3, O4, and O5). Object 5 (O5) is not in any generalized node.

Finding Object 5 after finding Objects 1 to 4 is easy. We simply use the O4 tag string (or any of the four 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 find that r3's only child O5 matches O4 well.

Experiment results

We evaluated MDR and compared it with

two current systems: Omini and IEPAD. For more on these systems, see the sidebar.

We used pages from Amazon.com, Yahoo.com, and the Hewlett-Packard Web site (www.hp.com) to build our system—that is, to implement and debug our MDR algorithm and to determine the default edit distance threshold. We determined a default value of 0.3, which we used in all our experiments. This fixed threshold requires no tuning for new pages or Web sites.

Our experiments used 46 Web pages (18 pages from Omini's Web site and 28 pages from a wide range of domains, such as travel, software, auctions, shopping, and search engine results) containing 621 data records. These data records include product lists and search results but not navigation areas, which can also have regular patterns. Because Omini tries to identify a page's main objects, it doesn't include navigation areas or other smaller regions. Both IEPAD and MDR report such regions if they exist. IEPAD generates many rules, all of which we tried. The best result is used for each page (the results for a page might come from more than one rule).

We used standard precision and recall measures to evaluate the results, which Table 1 summarizes. As the table shows, MDR had dramatically better results than either system. In fact, MDR had perfect results for all but one page. For this page, MDR missed one data record that was too dissimilar from its neighbors.

In our experiments, Omini and IEPAD worked well only with simple pages—that is, pages with many similar data records and little noise. However, most Web pages are complex with a lot of noise, although they might contain only a few data records (for example, many product pages list just a few products). In addition, Omini and IEPAD couldn't find noncontiguous data records in the three test pages that contained them.

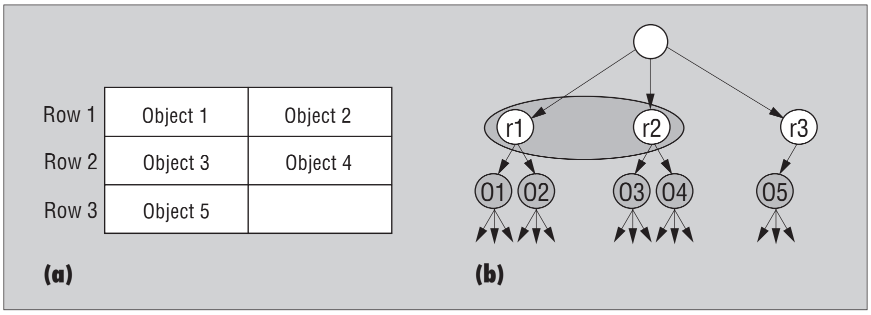


Figure 9. A data record layout for an odd number of objects: (a) an odd number of objects in a table and (b) HTML tag tree.

Our current work on MDR goes in two directions. First, we're customizing MDR for two practical applications: extracting consumer product reviews from online merchant and dedicated review sites, and extracting postings from Internet forum pages. The purpose of these tasks is to collect data for the subsequent analysis of reviews and forum postings to produce marketing intelligence information. Second, we're working on a more effective technique for extracting individual data fields from data records, which the current MDR system does not do. In our future work, we also plan to study the problem of extracting information from text documents that are much less structured than data records on the Web.

The MDR system is available at www.cs.uic.edu/~liub/MDR/MDR-download.html.

Reference

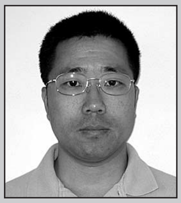
1. R. Baeza-Yates, "Algorithms for String Matching: A Survey," *ACM SIGIR Forum*, vol. 23, nos. 3–4, 1989, pp. 34–58.

Table 1. Summary of experimental results using 621 data records from 46 Web pages.

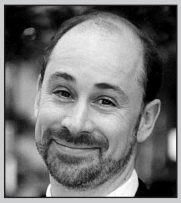
Test	MDR	Omini	IEPAD
Recall (%)*	99.8	39	39
Precision (%)*	100	56	67
Pages correctly extracted (every data record was found)	45	6	14
Pages in which no record was found	0	28	17
Pages with extraction errors (missing data records or items)	1	40	32

* Recall and precision are based on the correctly extracted data records, not Web pages.

The Authors



Bing Liu is an associate professor at the University of Illinois at Chicago. His research interests include data mining, Web and text mining, and machine learning. He received his PhD in artificial intelligence from the University of Edinburgh. He's a member of the IEEE Computer Society, the AAAI, and SIGKDD. Contact him at the Dept. of Computer Science, Univ. of Illinois at Chicago, 851 S. Morgan St., Chicago, IL, 60607-7053; liub@cs.uic.edu.



Robert Grossman is a professor of mathematics, statistics, and computer sciences and director of the National Center for Data Mining at the University of Illinois at Chicago. His research interests include data mining, distributed computing, and performance computing. He received his PhD in applied mathematics from Princeton University. He's a member of the IEEE, the ACM, the American Mathematical Society, and the Society for Industrial and Applied Mathematics. Contact him at the Dept. of Mathematics, Statistics, and Computer Science, Univ. of Illinois at Chicago, 851 S. Morgan St., Chicago, IL, 60607-7053; grossman@uic.edu.

Yanhong Zhai is a PhD student in computer science at the University of Illinois at Chicago. Her research interests include information retrieval and machine learning. She received her MS in computer science from the University of Illinois at Chicago. Contact her at the Dept. of Computer Science, Univ. of Illinois at Chicago, 851 S. Morgan St., Chicago, IL, 60607-7053; yzhai@cs.uic.edu.