YANHONG ZHAI
B.S. (Xi’an Jiaotong University) 1998
M.S. (University of Illinois at Chicago) 2003

Submitted as partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2005

Chicago, Illinois
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>13</td>
</tr>
<tr>
<td>3.3.1</td>
<td>13</td>
</tr>
<tr>
<td>3.3.2</td>
<td>14</td>
</tr>
<tr>
<td>3.3.3</td>
<td>15</td>
</tr>
<tr>
<td>3.3.4</td>
<td>15</td>
</tr>
<tr>
<td>3.3.5</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>16</td>
</tr>
<tr>
<td>3.5</td>
<td>19</td>
</tr>
<tr>
<td>3.5.1</td>
<td>22</td>
</tr>
<tr>
<td>3.5.2</td>
<td>25</td>
</tr>
<tr>
<td>3.6</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>34</td>
</tr>
<tr>
<td>4.2.1</td>
<td>34</td>
</tr>
<tr>
<td>4.2.2</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>5.1</td>
<td>41</td>
</tr>
<tr>
<td>5.2</td>
<td>42</td>
</tr>
<tr>
<td>5.3</td>
<td>45</td>
</tr>
<tr>
<td>6</td>
<td>47</td>
</tr>
<tr>
<td>7</td>
<td>48</td>
</tr>
<tr>
<td>7.1</td>
<td>48</td>
</tr>
<tr>
<td>7.2</td>
<td>49</td>
</tr>
<tr>
<td>7.3</td>
<td>50</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CITED LITERATURE</td>
<td>52</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>39</td>
</tr>
<tr>
<td>II</td>
<td>41</td>
</tr>
<tr>
<td>III</td>
<td>43</td>
</tr>
<tr>
<td>IV</td>
<td>44</td>
</tr>
<tr>
<td>I FINAL DATA TABLE (”1” INDICATES A DATA ITEM)</td>
<td>39</td>
</tr>
<tr>
<td>II COMPARISON RESULT WITH OMINI AND IEPAD</td>
<td>41</td>
</tr>
<tr>
<td>III EXPERIMENT RESULTS</td>
<td>43</td>
</tr>
<tr>
<td>IV EXPERIMENT RESULTS (CONTINUED)</td>
<td>44</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Example page segment with data records</td>
</tr>
<tr>
<td>2</td>
<td>An example DOM-tree of a page segment</td>
</tr>
<tr>
<td>3</td>
<td>A HTML code segment and the boundary coordinates</td>
</tr>
<tr>
<td>4</td>
<td>DOM-tree of the HTML code in Figure 3</td>
</tr>
<tr>
<td>5</td>
<td>The optimal alignment of $T_1$ and $T_2$</td>
</tr>
<tr>
<td>6</td>
<td>The simple tree matching algorithm</td>
</tr>
<tr>
<td>7</td>
<td>Two trees with more than one possible match</td>
</tr>
<tr>
<td>8</td>
<td>Calculating the normalized tree distance between two adjacent generalized nodes</td>
</tr>
<tr>
<td>9</td>
<td>An illustration of generalized nodes and data regions</td>
</tr>
<tr>
<td>10</td>
<td>Combination and comparison</td>
</tr>
<tr>
<td>11</td>
<td>The algorithm for mining data regions</td>
</tr>
<tr>
<td>12</td>
<td>The structure comparison algorithm</td>
</tr>
<tr>
<td>13</td>
<td>A possible configuration of data records</td>
</tr>
<tr>
<td>14</td>
<td>Finding data regions under a given node</td>
</tr>
<tr>
<td>15</td>
<td>The UnCoveredDRs procedure</td>
</tr>
<tr>
<td>16</td>
<td>Each generalized node contains more than one data record</td>
</tr>
<tr>
<td>17</td>
<td>Four objects with non-contiguous description</td>
</tr>
<tr>
<td>18</td>
<td>Algorithm for Finding Data Records</td>
</tr>
<tr>
<td>FIGURE</td>
<td>DESCRIPTION</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>19</td>
<td>Expanding the seed tree: (A) and (B) Unique expansion; (C) Insertion ambiguity</td>
</tr>
<tr>
<td>20</td>
<td>The partial tree alignment algorithm</td>
</tr>
<tr>
<td>21</td>
<td>Iterative tree alignment with two iterations</td>
</tr>
<tr>
<td>22</td>
<td>Example page containing nested data</td>
</tr>
</tbody>
</table>
SUMMARY

This report studies the problem of structured data extraction from Web pages. The objective of the proposed research is to automatically segment data records in a page, extract data items/fields from these records and store extracted data into a database. Existing methods addressing the problem of data extraction can be classified into two categories, each one having their own limitations. Methods in the first category use machine learning techniques to learn wrappers from human labeled examples. The learning process is often time-consuming due to the large number of sites and pages required for learning. Methods in the second category are based on the idea of automatic pattern discovery (APD). However, APD usually needs multiple pages that conform to a common schema as the input in order to initialize the extractor. This proposal proposes a new method to perform the task of Web data extraction automatically given a single page. The method consists of two steps, namely, (1) identifying individual records in a page, and (2) aligning and extracting items from identified records. For step 1, a method based on visual information (e.g., the physical location of HTML elements in a browsing window) is proposed to segment records. For step 2, a novel partial alignment technique based on tree matching is proposed, in which we align only those items in a pair of records that can be aligned (or matched) with certainty, and make no commitment on the rest of the items. We present some preliminary experimental result obtained using a large number of Web pages from diverse domains. The results show that visual information provides a very good cue for identifying data records and gives a better performance with higher precision and recall than merely using
DOM-trees. On the other hand, partial tree alignment enables very accurate alignment of multiple records.
CHAPTER 1

INTRODUCTION

Structured data in Web pages usually contain important information. Such data are often retrieved from underlying databases and displayed in Web pages by using fixed templates. In this proposal, we call these structured data data records. Extracting data records enables one to integrate data/information from multiple Web pages to provide value-added services, e.g., comparative shopping, meta-querying and search. Figure 1 shows a Web page segment containing a list of two products (books). The description of each book is a data record. Our objective is two fold: (1) automatically identify such data records in a page, and (2) automatically align and extract data items from the data records.

Several automatic approaches have been reported in the literature for mining data records from Web pages. In [1; 8], algorithms are proposed to find patterns and to infer grammars from multiple pages containing similar data records. Requiring an initial set of pages containing similar data records is, however, a major limitation of this type of approaches because such pages have to be found manually or by another system. In [18], a method is proposed to extract data records by exploring the detail pages (opposite to list pages, see section 7.3) that contain detailed information of each data record. The need for detail pages is also a serious limitation because many data records do not have such pages. Furthermore, the method assume that the detail pages are given, which is often not realistic in practice. In [5] an algorithm based on Patricia tree and approximate sequence alignment is proposed to find patterns in a Web page.
However, due to the inherent limitation of Patricia tree and inexact sequence matching, the algorithm often produces many patterns and most of them are spurious. Another assumption that most current systems make is that the relevant information of a data record is contained in a contiguous segment of the HTML code. However, in some Web pages, the description of one object may intertwine with the descriptions of some other objects. 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. In this case, the descriptions of both object1 and object2 are not contiguous. However, when they are displayed in a browser, they appear contiguous to human viewers. In Chapter 2, we discuss the use of visual information in identifying these non-contiguous data records and compare it with existing algorithms.

We propose a two-step strategy to solve the problem of data extraction from Web pages.
1. Given a page, the method first segments the page to identify each data record without extracting its data items. Visual information is used in this step in two ways:

- It helps improve the accuracy of data record segmentation by allowing the system to identify an valuable piece of information that is not available using mere DOM-trees.
- It enables the system to build DOM-trees more accurately. The proposed system identifies data records by analyzing HTML tag trees or DOM trees. A straightforward way to build a DOM-tree is to follow the nested tag structure in the HTML code. However, sophisticated analysis has to be incorporated to handle errors in the HTML code (e.g., missing or ill-formatted tags). Whereas the visual information can be obtained after the HTML code is rendered by a Web browser, it also contains information about the hierarchical structure of the tags. In this work, rather than analyzing the HTML code, visual information (i.e., physical locations on the screen at which tags are rendered) is utilized to infer the structural relationship among tags and to construct a DOM-tree. This method leads to more robust tree construction due to the high error tolerance of the rendering engines of Web browsers (e.g., Internet Explorer). As long as the browser is able to render a page correctly, its DOM-tree can be built correctly.

2. A novel partial tree alignment method is proposed to align and to extract corresponding data items from the discovered data records and put the data items in a database table. Using tree alignment is natural because of the nested (or tree structured) organization
of HTML code. Specifically, after all data records have been identified, the sub-trees of each data record are re-arranged into a single tree as each data record may be contained in more than one sub-tree in the original DOM-tree of the page, and each data record may not be contiguous. The DOM-trees of all the data records are then aligned using our partial tree alignment method. By partial tree alignment, we mean that for each pair of trees (or data records), we only align those nodes (or data items) that can be aligned with certainty and ignore those parts that cannot, i.e., making no commitment on the locations of the unaligned data items. Early uncertain commitments can result in undesirable effects for later alignment involving other data records. This method turns out to be very effective for multiple tree alignment. The resulting alignment enables us to extract data items from all data records in the page. It can also serve as an extraction pattern to be used to extract data items from other pages with data records generated using the same template.

The proposed two-step approach called DEPTA [32](Data Extraction based on Partial Tree Alignment), which is very different from all existing methods, does not make any unrealistic assumptions commonly found in existing methods, as will be explained in Chapter 2. As long as a page contains at least two data records, our system will automatically find them. We present some preliminary experimental results obtained using a large number of pages to show the effectiveness of the proposed technique.
CHAPTER 2

PREVIOUS WORK

One area of previous research related to this work is wrapper generation. A wrapper is a program that extracts data from a Web site or page and put them in a database. There are two main approaches to wrapper generation. The first approach is wrapper induction, which uses supervised learning to learn data extraction rules from a set of manually labeled positive and negative examples. Manual labeling of data is, however, labor intensive and time consuming. Additionally, for different sites or even pages in the same site, the manual labeling process needs to be repeated because they follow different templates/schemas. Example wrapper induction systems include WIEN [15], Softmealy [13], Stalker [20], WL2 [7], [22] and etc. The second approach is automatic extraction. In [9], a study is made to automatically identify data records’ boundaries based on a set of heuristic rules and domain ontology. Domain ontology is costly to build (about 2-man weeks for a given Web site)[9]. In [3], some additional heuristics are proposed to perform the task without using domain ontology. However, the experiment result in [19] shows that the performance of this approach is not satisfactory. In [5] a method (Iepad) is proposed to find patterns from the HTML tag string of a page, and then use the patterns to extract data items. The method uses the Patricia tree and sequence alignment to find inexact matches. The problem with Patricia tree is that it is only able to find exact matches of patterns. In the context of the Web, data records are seldom exactly the same. Thus, a heuristic method based on string alignment is also proposed in [5] to find inexact matches.
However, this results in many spurious patterns. Our new method uses tree alignment instead of string alignment, which exploits nested tree structures to perform much more accurate data extraction. Inspired by work on [5], a system for wrapper generation and label assignment is proposed in [30]. In their work, a candidate wrapper is generated based on a single page by finding repeated patterns in a HTML string, and then multiple similar pages are needed to construct a generalized wrapper based on multiple candidate wrappers. Therefore, their work needs multiple pages that conform to a common schema as the input. Besides, the method only deals with the result pages returned by a query to a complex search form, which also a limitation since a lot of Web sites only provide a simple keyword search. In [8], a matching algorithm is proposed to infer union-free regular expressions that represent page schemas given multiple pages. The algorithm itself has exponential time complexity. In order to limit the complexity a pruning technique is employed, which also compromise the expressive power of the inferred grammar. Furthermore, union-free regular grammar can not deal with disjunctions in the input schema. These problems are addressed in [1] and a polynomial time algorithm is proposed by using several heuristics. Both of them need multiple input pages that conform to a common schema and assume these pages are given, whereas, we are concerned with extracting data and discover schemas from a single page. Another method for data extraction from a list page (page containing a list of records) is proposed in [18]. Its main idea is to utilize the redundant information in list pages and detail pages (page containing detailed information of a single record) to aid information extraction. The method also needs multiple similar pages to infer a template to find tables in list pages, and then find records from the tables. It is
not applicable to the pages where data records have no links to detail pages. Furthermore, the method assumes that detail pages are given, which is not realistic. Due to a large number of links in a typical Web page, automatically identifying the correct links that point to detail pages is not a trivial task.

Tree matching is used for Web data extraction in [23; 12]. In [23], in order to extract Web news, all the input pages are clustered and the outcome of tree matching is used as the distance measure of the hierarchical clustering. For the trees being grouped into one cluster, tree patterns are generated based on the matching result. The time complexity of the clustering is $O(n^2m^2)$ (where $n$ is the number of input pages and $m$ is the average length of input pages) and for every new page, it needs to compare with all the existing tree patterns and then choose the one with the highest matching score. This makes the algorithm unable to scale to a large number of Web pages. In [12], users are required to label the data of interest. Tree matching is applied based on the labeled data in order to generate the wrapper. The generated tree patterns have less expressive power than the one presented in [8]. In their algorithm, any DOM element that exists only in one tree is discarded. Therefore, their technique can not deal with the optional case in the regular pattern.

Visual information of Web pages has been used in the area of web mining in the last few years. Using visual layout of documents for data extraction is first proposed in [24] and later on used for segmenting Web pages into different blocks and assigning different importance values for each block [26]. The latest technique which also uses visual information for wrapper
generation is proposed in [34], but it only deals with the result pages returned by search engines, which have limited and domain-specific rendering characteristics.
CHAPTER 3

DATA RECORD IDENTIFICATION

We now start to present our proposed technique. This section focuses on the first step: segmenting the Web page to identify individual data records. It does not align or extract data items in the data records, which will be the topic of the next section.

3.1 The Basic Idea of Identifying Data Records

This step is based on two observations about data records in Web pages:

1. A group of data records that contains descriptions of a set of similar objects are typically rendered 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 books are presented in one contiguous region. They are also formatted using similar sub-trees. We can use restricted tree mapping approaches (introduced in section 3.3) to compare different sub-trees to find those similar ones, which may represent similar data records. The problem with this approach is that the computation is prohibitive because a data record can start from any tag and end at any tag. The next observation helps to deal with this problem.

2. The structure of a Web page can be represented by a DOM-tree. For example, Figure 2 shows an example DOM-tree. In this tree, each data record is wrapped in 3 TR nodes together with their sub-trees. The two data records are in the two dash-lined boxes. Our
second observation is that a set of similar data records are formed by some child sub-trees of the same parent node. This observation makes it possible to design a very efficient algorithm based on restricted tree mapping to identify data records because it limits the tags from which a data record may start and end in a DOM-tree.

Experiments show that these observations work very well. By no means, we 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 format for data records. Given a Web page, the algorithm works in three steps:

**Step1:** Building a DOM-tree of the page based on visual (rendering) information.

**Step2:** Mining data regions in the page using the DOM-tree. A data region is an area in the page that contains a list of similar data records. Instead of mining data records directly, which is hard, the algorithm mines data regions first and then finds data records within them.

For example, in Figure 2, we first find the single data region below node TBODY. Both
visual information and structure information of the page are used in this step to produce accurate results.

Step3: 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.

3.2 Building DOM-Trees based on Visual Information

In a Web browser, each HTML element (consisting of a start tag, optional attributes, optional embedded HTML content, and an end tag that may be omitted) is rendered as a rectangle spatially. A DOM-tree can be constructed based on the nested rectangles. The details are as follows:

1. Find the 4 boundaries of the rectangle of each HTML element by calling the embedded parsing and rendering engine of a browser, e.g., Internet explorer.

2. Detect the containment relationship among the rectangles, i.e., whether one rectangle is contained inside another rectangle. A tree can be built based on the containment check.

Let us use an example to illustrate the process. Assume we have the HTML code on the left of Figure 3, which is a table with two rows (tr’s) and each row has two cells (td’s). The rendering engine of the browser produces the boundary coordinates (in pixels) for each HTML element shown on the right of Figure 3.

With this spatial coordinate’s information, we can build the tree in Figure 4 by following the sequence of opening tags and by containment checks. The tree construction algorithm is fairly straightforward. We will not discuss it further here.
Figure 3. A HTML code segment and the boundary coordinates

```html
1  <table>
2   <tr>
3     <td>...</td>
4     <td>...</td>
5   </tr>
6   <tr>
7     <td>...</td>
8     <td>...</td>
9   </tr>
10  </table>
```

<table>
<thead>
<tr>
<th></th>
<th>left</th>
<th>right</th>
<th>top</th>
<th>bottom</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>300</td>
<td>200</td>
<td>400</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>300</td>
<td>200</td>
<td>300</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>200</td>
<td>200</td>
<td>300</td>
</tr>
<tr>
<td>4</td>
<td>200</td>
<td>300</td>
<td>200</td>
<td>300</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>300</td>
<td>300</td>
<td>400</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>200</td>
<td>300</td>
<td>400</td>
</tr>
<tr>
<td>7</td>
<td>200</td>
<td>300</td>
<td>300</td>
<td>400</td>
</tr>
<tr>
<td>8</td>
<td>200</td>
<td>300</td>
<td>300</td>
<td>400</td>
</tr>
</tbody>
</table>

Figure 4. DOM-tree of the HTML code in Figure 3
3.3  Tree Distance Measures

In order to mine data regions in a page, tree pattern matching is exploited to find regions that contain lists of similar subtrees. Analogous to the problem of string pattern matching, tree pattern matching is a program that takes as input two trees $T_1$ and $T_2$ and finds all subtrees of $T_2$ that are isomorphic to a subtree of $T_1$ (See [10]). Two tree $T_1$ and $T_2$ are isomorphic if there is a bijection between the nodes of $T_1$ and the nodes of $T_2$. Based on the different notion of subtree, several distance measure for comparing trees have been proposed in the literature: edit distance [27], alignment distance [14], isolated-subtree distance [28], top-down distance [27; 31] and bottom-up distance [29]. In the following part of this section, we will briefly review these tree distance measures and give the reason for adopting top-down distance in our work. For the easiness of comparison, all the trees we discuss are rooted, ordered and labeled.

3.3.1  Tree Edit Distance and Mappings

Similar to string edit distance, tree edit distance [27] between two trees $T_1$ and $T_2$ is the cost associated with the minimum set of operations needed to transform $T_1$ into $T_2$. In the classic formulation, the set of allowable edit operations includes node removal, node insertion, and node replacement. A cost is usually assigned to each kind of operations. Solving the tree edit distance problem is often assisted by finding a minimum-cost mapping between two trees [27]. The concept of mapping [27] is formally defined as:

Let $T$ be a tree. $n$ denotes the number of nodes of $T$. $T[i]$ denotes the $ith$ node of tree $T$ in a preorder traversal of the tree. A triple $(M, T_1, T_2)$ is a mapping $M$ from $T_1$ to $T_2$, where $M$ is any set of pairs of integers $(i, j)$ satisfying:
(1) $1 \leq i \leq n_1$, $1 \leq j \leq n_2$;

(2) For any two pairs $(i_1, j_1)$ and $(i_2, j_2)$ in $M$,

(a) $i_1 = i_2$ iff $j_1 = j_2$;

(b) $i_1 < i_2$ iff $j_1 < j_2$;

(c) $T_1[i_1]$ is an ancestor(descendant) of $T_1[i_2]$ iff $T_2[j_1]$ is an ancestor(descendant) of $T_2[j_2]$.

Intuitively, the definition requires that each node can appear no more than once in a mapping and the order between sibling nodes and the hierarchical relation between nodes are both preserved. However, mapping can cross levels. An algorithm was given in [27] to compute the edit distance between two trees in $O(n_1n_2h_1h_2)$ time, where $n_i$ and $h_i$ are the number of nodes and the maximum depths of $T_i$. The best upper limit for this problem is due to an algorithm presented in [6] with complexity $O(n_1n_2 + l_1^2 + l_1^{2.5}l_2)$ where $l_i$ is the number of leaves in $T_i$.

The edit distance problem for unordered trees was shown to be NP-complete in [33].

### 3.3.2 Alignment Distance

Even though edit and alignment are two equivalent notions for sequences, they are very different for trees, as shown in [14]. An alignment $\mathcal{A}$ of $T_1$ and $T_2$ is obtained by first inserting nodes labeled with spaces into $T_1$ and $T_2$ such that the two resulting trees $T'_1$ and $T'_2$ are isomorphic, and then overlaying $T'_1$ on $T'_2$. A cost is defined for each pair of labels. The value of $\mathcal{A}$ is the sum of the costs of all pairs of opposing labels. In general the alignment distance is larger than the edit distance because each alignment corresponds to a restricted tree edit in
which all the insertion precede all the deletions. Figure 5 shows an example alignment, where
the alignment distance is 4 while the edit distance is 2, provided unit cost for each operation.

3.3.3 Isolated-Subtree Distance

The isolated-subtree distance was introduced in [28]. A mapping $M$ from tree $T_1$ to $T_2$ is
isolated if for all $(i_1, j_1), (i_2, j_2) \in M$, the following condition satisfies: the rightmost node of
$T_1[i_1]$ is to the left of $T_1[i_2]$ iff the rightmost node of $T_2[j_1]$ is to the left of $T_2[j_2]$.

3.3.4 Top-Down Distance

The top-down distance was introduced in [25], and an algorithm was given in [31] to compute
the distance between two trees $T_1$ and $T_2$ in $O(n_1 n_2)$ time. A mapping $M$ from tree $T_1$ to $T_2$ is
top-down if it satisfies the following condition: for all $i, j$ such that $T_1[i]$ and $T_2[j]$ are not root
nodes, if $(i, j) \in M$ then $(\text{par}(i), \text{par}(j)) \in M$.

Figure 5. The optimal alignment of $T_1$ and $T_2$. 
3.3.5 Bottom-Up Distance

On the contrary, the bottom-up distance was introduced in [29], in which for any pair of nodes \((T_1[i], T_2[j])\), where \((i, j) \in M\), all child nodes of \(T_1[i]\) and \(T_2[j]\) should be also in the mapping \(M\). An algorithm for computing this distance is proposed, which takes worst-case \(O(n_1 + n_2)\) time. The bottom-up distance coincides with the top-down distance only for isomorphic trees and both of them are restricted version of isolated-subtree distance which, in turn, is restricted version of alignment distance which, in turn, is restricted version of edit distance. This yields a hierarchy among the mappings underlying the different distance measures (see [29]).

In this proposal, we propose to deal with DOM trees that represent Web page structures. For the pages, or data records, generated from a common template, they are different because different values are retrieved from back-end database and inserted into the schema and some data items have optional, repeating or disjunctive values. These values are at the lower level of the DOM trees (or sub-trees which corresponding to data records). Therefore, these DOM trees (or sub-trees) usually have more structural dissimilarity at the lower levels (which represent data items) of the tree than at the top levels (which represent the page schemas). For this kind of situation, top-down distance is the measure that is appropriate for the tree comparison with low time complexity.

3.4 Simple Tree Matching

In this section, an algorithm for computing top-down mapping is adopted, which was first proposed in [31] to compare two computer programs in software engineering. It was called simple
tree matching (STM). STM evaluates the similarity of two trees by producing the maximum matching through preorder traversal of the trees with complexity $O(n_1n_2)$, where $n_1$ and $n_2$ are the sizes of trees $T_1$ and $T_2$ respectively. No node replacement or level crossing is allowed. Maximum matching of two trees $T_1$ and $T_2$ is a top-down mapping $M$ with the maximum number of pairs of nodes with the same label.

When two nodes $r_1$ and $r_2$ match, they must satisfy two conditions:

1. $r_1$ and $r_2$ have identical labels;
2. $r_1$ and $r_2$ do not have visual conflicts. A set of criteria is built to decide whether two nodes have visual conflicts or not. For example, if the bounding rectangles of two nodes are neither vertically nor horizontally aligned and the difference of their areas exceeds a certain threshold, these two nodes are considered to have visual conflicts. Visual conflicts imply the two nodes are unlikely to contain similar data records.

Let $T_1 = < r_1, T_{1[1]}, T_{1[2]}, ..., T_{1[m]} >$ and $T_2 = < r_2, T_{2[1]}, T_{2[2]}, ..., T_{2[n]} >$ be two trees, where $r_1$ and $r_2$ are the roots of $T_1$ and $T_2$, and $T_{1[i]}, T_{2[j]}$ are the $ith$ and $jth$ first-level sub-trees of $T_1$ and $T_2$ respectively. When $r_1$ and $r_2$ match, the maximum matching between $T_1$ and $T_2$ is $M_{T_1T_2} + 1$, where $M_{T_1T_2}$ is the maximum matching between $< T_{1[1]}, T_{1[2]}, ..., T_{1[m]} >$ and $< T_{2[1]}, T_{2[2]}, ..., T_{2[n]} >$. $M_{T_1T_2}$ can be obtained by the following dynamic programming scheme:

1. If the maximum matching between $T_{1[i]}$ and $T_{2[i]}$ is larger than any maximum matching between $T_{1[m]}$ and $T_{2[n]}$, is larger than any maximum matching between $T_{1[i]}$ and $T_{2[i]}$ (1 ≤ $i$ < $n$), then $M_{T_1T_2}$ is the maximum matching between $<
Algorithm Simple_Tree_Matching($T_1, T_2$)
1: if the roots of the two trees $T_1$ and $T_2$ do not match then
2: return (0)
3: else
4: $m :=$ the number of first-level sub-trees of $T_1$;
5: $n :=$ the number of first-level sub-trees of $T_2$;
6: Initialization: $M[i, 0] := 0$ for $i = 0..m$; $M[0, j] := 0$ for $j = 0..n$;
7: for $i = 1$ to $m$ do
8: for $j = 1$ to $n$ do
9: $M[i, j] := \max (M[i, j - 1], M[i - 1, j], M[i - 1, j - 1] + W[i, j])$;
10: where $W[i, j] = \text{Simple_Tree_Matching}(T_1[i], T_2[j])$
11: end for
12: end for
13: return ($M[m, n] + 1$)
14: end if

Figure 6. The simple tree matching algorithm

$T_1[1], T_1[2], ..., T_1[m-1] >$ and $< T_2[1], T_2[2], ..., T_2[n-1] >$ plus the maximum matching between $T_1[m]$ and $T_2[i]$.

2. Otherwise, $M_{T_1T_2}$ is the same as the maximum matching between $< T_1[1], T_1[2], ..., T_1[m] >$
and $< T_2[1], T_2[2], ..., T_2[n-1] >$, or between $< T_1[1], T_1[2], ..., T_1[m-1] >$ and $< T_2[1], T_2[2], ..., T_2[n] >$.

Figure 6 shows the Simple_Tree_Matching algorithm in which the roots of $T_1$ and $T_2$ are compared first (line 1). If the roots do not match, then the two trees do not match at all. Otherwise, the algorithm recursively finds the maximum matching between first-level sub-trees of $T_1$ and $T_2$ and save it in $W$ matrix (line 8). Based on the $W$ matrix, a dynamic programming scheme is applied to find the number of pairs in a maximum matching between two trees $T_1$ and $T_2$. 

During the matching (or after the matching), we can trace back in the $M$ matrices to find the matching nodes from the two trees. When there is more than one match for a node that gives the maximum result, visual information is used to decide which one to choose. For example, in Figure 7, node $c$ in tree $T_1$ can match either the first or the last node $c$ in tree $T_2$. We choose the one which is visually more close to/aligned with the node in $T_1$.

3.5 Mining Data Regions

Based on the top-down tree distance measure, this step mines every data region in a 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 later). Below, we define generalized nodes and data regions using the DOM-tree:

**Definition 1** A generalized node (or a node combination) of length $r$ consists of $r$ ($r \geq 1$) nodes in the DOM-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 two situations:

1) A data record may be contained in a few sibling tag nodes rather than one. For example, in Figure 1 and Figure 2, we can see that each notebook is contained in three table rows (or 3 TR nodes);

2) An object may not be contained in a contiguous segment of HTML code.

Note that we call each node in the DOM-tree a tag node to distinguish it from a generalized node.

**Definition 2** A data region is a list of 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 tree distance between adjacent generalized nodes is less than a fixed threshold.

Given two adjacent generalized node of length $r$, $G_1 = <g_1[1], g_1[2], \ldots, g_1[r]>$ and $G_2 = <g_2[1], g_2[2], \ldots, g_2[r]>$, the normalized tree distance between them is calculated as Figure 8 shows. In line 6, Terminal() returns the number of nodes that enclose images or text and such nodes are called *terminal nodes.* For example, in the tree representing a HTML segment `<TABLE><TR><TD><B>price:</B></TD></TR></TABLE>`, only `<B>` is a terminal node and Terminal() returns 1.
procedure NormTreeDist(\( < g_1[1], g_1[2], \ldots, g_1[r] >, < g_2[1], g_2[2], \ldots, g_2[r] > \))

1: if \( r=1 \) then
2: \( T_i \) = the subtree rooted at \( g_i[1] \)
3: else
4: \( T_i \) = a new tree with an artificial root node added on top of the forest rooted at \( g_1[1], g_1[2], \ldots, g_1[r] \)
5: end if
6: return \( \frac{\text{Terminal}(\text{STM}(T_1, T_2))}{\text{Max}(\text{Terminal}(T_1), \text{Terminal}(T_2))} \)

Figure 8. calculating the normalized tree distance between two adjacent generalized nodes

It is important to notice that although the generalized nodes in a data region have the same length, the 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 generalized nodes and data regions, we make use of an artificial DOM-tree in Figure 9. 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 tree 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. The node pairs (14, 15), (16, 17) and (18, 19) are generalized nodes of length 2. They together define the data region labeled 3.

We end this part with two 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 realistic.
2. A generalized node may not represent a final data record (see Section 3.6). It will be used to find the final data records.

3.5.1 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? (2) What is the length is a generalized node? 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. 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 9, 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 10 to illustrate the comparison process. Figure 10 has 10 nodes below a parent node p. We start from each node and calculate normalized tree distance between all possible combinations of component nodes. Let the maximum number of components that a generalized node can have be 3 in this example.

Start from node 1: We compute the following normalized tree distances.

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

Start from node 2: We only compute:

- \((<2,3>, <4,5>), (<4,5>, <6,7>), (<6,7>, <8,9>)\)

Figure 10. combination and comparison
Algorithm MDR(Node, K)
1: CombComp(Node.Children, K)
2: for eachChildNode ∈ Node.Children do
3:   MDR(ChildNode, K)
4: end for

Figure 11. The algorithm for mining data regions

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

1-node comparisons are omitted because they have been done when we started from node 1.

Start from node 3: We only need to compute:

- \(<3, 4, 5>, <6, 7, 8>\)

We do not need to start from any other nodes after node 3 because all the computations have been done.

The algorithm (MDR) for mining data regions is given in Figure 11. It traverses the DOM-tree from the root downward in a depth-first fashion (lines 2 and 3). At each internal node, procedure CombComp() calculate normalized tree distance between various combinations of the children sub-trees.

The main idea of CombComp() has been discussed above. In line 1 of Figure 12, it starts from each node of NodeList. It only needs to try up to the Kth node. In line 2, it compares different combinations of nodes, beginning from i-component combination to K-components.
procedure CombComp(NodeList, K)
1: for i = 1; i ≤ K; i + + do
2:     for j = i; j ≤ K; j + + do
3:         if NodeList[i + 2 * j - 1] exists then
4:             Start = i;
5:             for k = i + j; k < Size(NodeList); k + + do
6:                 if NodeList[k + j - 1] exists then
7:                     NormTreeDist(NodeList[Start..(k - 1)], NodeList[k..(k + j - 1)])
8:                 end if
9:             end for
10:         end if
11:     end for
12: end for

Figure 12. The structure comparison algorithm

combination. Line 3 tests to see whether there is at least one pair for comparison. Lines 4-8 perform Simple_Tree_Matching of various combinations by calling procedure NormTreeDist.

Assume that the number of elements in NodeList is n. Without considering the comparison of trees, the time complexity of CombComp is \(O(nK)\). Since \(K\) is normally small (< 10), the algorithm can be considered linear in \(n\). Assume that the total number of nodes in the DOM-tree is \(N\), the complexity of MDR is \(O(NK)\) without considering the comparison of trees.

3.5.2 Determining Data Regions

After all comparisons have been done, we can identify each data region by finding its generalized nodes. We use Figure 13 to illustrate the 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.
The algorithm basically uses the comparison results at each parent node to find similar children node combinations to obtain candidate \textit{generalized nodes} and \textit{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 one. For example, in Figure 13, at a low level cell 1 and cell 2 are candidate \textit{generalized nodes} and they together form a candidate \textit{data region}, row 1. However, they are covered by the \textit{data region} including all the 4 rows at a higher level. In this case, we only report the higher level one as the data region and each row as a generalized node. The reason for taking this approach is to avoid the situations where many low level nodes (with very small sub-trees) are very similar but do not represent true data records.

2. A property about similar trees is that if a set of trees $T_1, T_2, T_3, \ldots, T_n$ are similar to one another, then a combination of any number of them is also similar to another combination of the same number. Thus, we only report \textit{generalized nodes} of the smallest length that
cover a data region, which helps us to find the final data records later. In Figure 13, we only report each row as a generalized node rather than a combination of two rows (rows 1-2, and rows 3-4).

The algorithm for this step is given in Figure 14. It finds every data region and its generalized nodes under a given node. \( T \) is the normalized tree distance threshold. \( K \) is the maximum number of tag nodes in a generalized node (we use 10 in our experiments, which is sufficient). \( \text{Node.DRs} \) is the set of data regions under \( \text{Node} \), and \( \text{tempDRs} \) is a temporal variable storing the data regions passed up from every Child of \( \text{Node} \). The basic idea of the algorithm is to traverse the DOM-tree depth-first. It performs one function at each node when it goes down (line 1), and performs another when it backs up before going down to another branch of the tree (line 5).

1. When it goes down, at each node it identifies all the data regions of the node using procedure \( \text{IdentDRs} \). Note that these are not the final data regions of the Web page, but only the candidate ones.

2. When it backs up, it checks to see whether the parent level data regions in \( \text{Node.DRs} \) cover the child level data regions. Those covered child level data regions are discarded. Those that are not in \( \text{Child.DRs} \) are stored in \( \text{tempDRs} \) (line 5). After all children nodes of \( \text{Node} \) are processed, \( \text{Node.DRs} \cup \text{tempDRs} \) gives the current data regions discovered from the sub-tree starting from \( \text{Node} \) (line 6).
procedure FindDRs(Node, K, T)
1: Node.DRs = IdenDRs(1, Node, K, T);
2: tempDRs = ∅;
3: for each Child ∈ Node.Children do
4:   FindDRs(Child, K, T);
5:   tempDRs = tempDRs ∪ UnCoveredDRs(Node, Child);
6: end for
7: Node.DRs = Node.DRs ∪ tempDRs

Figure 14. Finding data regions under a given node

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 under 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.

Basically IdentDRs checks each combination and each starting point. For each possibility, it finds the first continuous region with a set of generalized nodes. smaller generalized nodes are used unless the larger ones cover more nodes and starts no later than the smaller ones. Finally, procedure UnCoveredDRs is given in Figure 15.

Assume that the total number of nodes in the DOM-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.
procedure UnCoveredDRs(Node, Child)
1: for each data region DR in Node.DRs do
3: return null
4: end if
5: end for
6: return Child.DRs

Figure 15. The UnCoveredDRs procedure

3.6 Identifying Data Records

After all data regions have been identified, we identify the data records from generalized nodes. We note that a generalized node may not represent a single data record in the following cases:

1. Procedure UnCoveredDR() always reports higher level data regions but the actual data records may be at a lower level. That is, a generalized node may contain one or more data records.

2. The information of a data record is not in a contiguous segment of the HTML code. That is, a generalized node may contain part of a data record.

Figure 16 shows an example of the first case. In the example, a data region contains two table rows and each row is a generalized node. However, they are not individual data records. Each row actually contains two data records in the two table cells. To find data records from each generalized node in a data region, this 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 similar objects or data records. An example of the second case is shown in Figure 17. In this example, the HTML code segment has the following sequence:

name1, name2, description1, description2, name3, name4, description3, description4.

Accordingly, the identified data region contains two generalized nodes, and each generalized node contains two tag nodes, which indicates that these two tag nodes (rows) are not similar to each other. But each tag node has the same number of children nodes and the children nodes are similar to each other. For this kind of situation, the corresponding children nodes
procedure FindRecords(G)
1: if \( \text{Length}(G) = 1 \) AND all children nodes of \( G \) are similar then
2: each child node of \( G \) is a data record
3: else
4: if \( \text{Length}(G) > 1 \) AND the children nodes of each node in \( G \) are similar AND each node also has the same number of children then
5: the corresponding children nodes of every node in \( G \) form a non-contiguous data record
6: end if
7: else
8: \( G \) itself is a data record
9: end if

Figure 18. Algorithm for Finding Data Records

of every tag node in a generalized node form a non-contiguous data record. This is illustrated by the DOM-tree at the bottom of Figure 17, where \( r \) represents row, \( n \) represents name and \( d \) represents description. \( G1 \) and \( G2 \) are generalized nodes. \( (n1, d1), (n2, d2), (n3, d3), \) and \( (n4, d4) \) form four data records. The algorithm for this step is given in Figure 18.
CHAPTER 4

DATA EXTRACTION

We now present the partial tree alignment technique for data extraction. The key task is how to match corresponding data items from all data records. There are two sub-steps:

1. Produce one rooted DOM-tree for each data record: After all data records are identified, the sub-trees of each data record are re-arranged into a single tree. As shown above, each data record may be contained in more than one sub-tree. Thus, this sub-step is needed to compose a rooted tree for each data record (an artificial root node may also need to be added). We will not discuss this further as it is fairly simple.

2. Partial tree alignment: The DOM-trees of all data records in each data region are aligned using partial tree alignment. It should be noted that in the alignment process, we only use tags. The content of data items are not involved.

Below, we will discuss multiple alignments and present the partial tree alignment method for aligning multiple data records based on their DOM-trees. We note here that string edit distance is not suitable as a string does not consider the tree structure, which is very useful in determining the correct alignment of data items. Due to the fact that more than one alignment of two strings may result in the same edit distance, string alignment can result in many errors. The matter is made worse by the fact that most tags used to form data records are *tr’s* and *td’s*. After string matching, it is hard to decide which alignment is the correct one as there are many possible
alignments. However, tree matching significantly reduces the number of possible alignments because of the tree structure constraint.

4.1 Multiple Alignment

Since each data region in a page contains multiple data records, we need to align multiple DOM-trees in order to produce a single database table with all the corresponding data items in the same column of the table. In this data table, each row represents a tree (data record), and each column represents a data item in each data record. Several existing algorithms can perform alignment of multiple sequences/trees. In [4], a multiple alignment method is proposed using multidimensional dynamic programming. The method is optimal but its time complexity is exponential, and thus not suitable for practical use. Many heuristic methods are also proposed [21; 11; 2]. Center string method, which is used in [21], is a particular heuristic method for multiple sequence alignments, which can also be used for trees. In this method, a sequence $x_c$ that minimizes $\sum_{i=0}^{k} D(x_i, x_c)$ is selected as the center ($D(x_i, x_c)$ is the distance of two strings). Then a pair-wise alignment is performed for each pair $(x_i, x_c)$, where $i \neq c$. Assuming there are $k$ sequences and all sequences have length $n$, finding the center takes $O(k^2n^2)$ time and each step of the iterative pair-wise alignment takes $O(n^2)$ time. Therefore the overall time cost is $O(k^2n^4)$. Similarly, we can find a center tree $T_c$ and align all the other trees with $T_c$. There are two main drawbacks with this technique: Firstly, although the algorithm has a polynomial time complexity, it runs slowly for pages containing many data records or data records containing many attributes. Secondly, if the center tree does not have a particular data item, other data records that contain the same data item will not be aligned. We implemented the method and
the results were poor. Other popular multiple alignment methods include progressive alignment [11] and iterative alignment [2]. They work like hierarchical clustering, and all requires upfront \(O(k^2)\) pair-wise matching. For our task, we can do better because we know that data records follow some predefined template.

4.2 Partial Tree Alignment

Our proposed approach aligns multiple DOM-trees by progressively growing a seed tree. The seed tree, denoted by \(T_s\), is initially picked to be the tree with the maximum number of data fields. The reason for choosing this seed tree is clear as it is more likely for this tree to have a good alignment with data fields in other data records. Then for each \(T_i(i \neq s)\), the algorithm tries to find for each node in \(T_i\) a matching node in \(T_s\). When a match is found for node \(T_i[j]\), a link is created from \(T_i[j]\) to \(T_s[k]\) to indicate its match in the seed tree. If no match can be found for node \(T_i[j]\), then the algorithm attempts to expand the seed tree by inserting \(T_i[j]\) into \(T_s\). The expanded seed tree \(T_s\) is then used in subsequent matching.

4.2.1 Partial alignment of two trees

Before presenting the full algorithm for aligning multiple trees, let us first discuss the idea of partial alignment of two trees. As indicated above, after \(T_s\) and \(T_i\) are matched, some nodes in \(T_i\) can be aligned with their corresponding nodes of \(T_s\) because they match one another. For those nodes in \(T_i\) that are not matched, we want to insert them into \(T_s\) as they may contain optional data items. There are two possible situations when inserting a new node \(T_i[j]\) into the seed tree \(T_s\), depending on whether a location in \(T_s\) can be uniquely determined to insert \(T_i[j]\). In fact, instead of considering a single node \(T_i[j]\), we can consider a sequence of unmatched
consecutive sibling nodes $T_i[j]...T_i[m]$ together. Without loss of generality, we assume that the parent node of $T_i[j]...T_i[m]$ has a match in $T_s$ and we want to insert $T_i[j]...T_i[m]$ into $T_s$ under the matching parent node. We only insert $T_i[j]...T_i[m]$ into $T_s$ if the location for the insertion can be uniquely determined in $T_s$. Otherwise, they will not be inserted and left unaligned. The alignment is thus partial. The location for insertion of $T_i[j]...T_i[m]$ can be uniquely decided in the following cases:

1. if $T_i[j]...T_i[m]$ have two neighboring siblings in $T_i$, one on the right and one on the left, that are matched with two consecutive siblings in $T_s$. Figure 19(A) shows such a situation, which gives one part of $T_s$ and one part of $T_i$. We can see that node $c$ and node $d$ (which are consecutive sibling nodes) in $T_i$ can be inserted into $T_s$ between node $b$ and node $e$ in $T_s$ because node $b$ and node $e$ in $T_s$ and $T_i$ match. The new (extended) $T_s$ is also shown in Figure 19(A). It should be noted that nodes $a$, $b$, $c$, $d$ and $e$ may also have their own children. We did not draw them to save space. This applied to all the cases below.

2. if $T_i[j]...T_i[m]$ has only one left neighboring sibling $x$ in $T_i$ and $x$ matches the right most node $x$ in $T_s$, then $T_i[j]...T_i[m]$ can be inserted after node $x$ in $T_s$. Figure 19(B) illustrates this case.

3. if $T_i[j]...T_i[m]$ has only one right neighboring sibling $x$ in $T_i$ and it matches the left most node $x$ in $T_s$, then $T_i[j]...T_i[m]$ can be inserted before node $x$ in $T_s$. This case is similar to above.

Otherwise, we cannot uniquely decide a location for unmatched nodes in $T_i$ to be inserted into $T_s$. This is illustrated in Figure 19(C). In this case, the unmatched node $x$ in $T_i$ could be
inserted into $T_s$ either between nodes $a$ and $b$, or between node $b$ and $e$. In this situation, we will not insert it into $T_s$.

### 4.2.2 Partial Alignment of Multiple Trees

Figure 20 gives the algorithm for multiple tree alignment based on partial alignment of two trees. We use an example in Figure 21 to explain the algorithm.

We have three example trees, all of which have only two levels. Lines 1 and 2 (Figure 20) basically find the tree with the most data items which becomes the seed tree. In Figure 21, the seed tree is the first tree (we omitted many nodes on the left of $T_1$). Line 3 does some initializations. Line 4 starts the while loop to align each of the rest trees against $T_s$. Line 5 picks the next unaligned tree, and line 6 does the tree matching. Line 7 finds all the matched pairs by tracing the matrix results of line 6. This procedure is similar to align two strings using edit distance. Note that line 6 and line 7 can be integrated. We present them separately for easiness to understand. In Figure 21, $T_s$ and $T_2$ produce one match, node $b$. Nodes $n, c, k$
Algorithm PartialTreeAlignment($S$)
1: sort trees in $S$ in descending order according to the number of data items that are not aligned;
2: $T_s = S.front(); S.pop();$ \{pick the first one and delete it from $S$ \}
3: flag = false; $R = \emptyset; I = false;$
4: while $S \neq \emptyset$ do
5: $T_i = S.front(); S.pop();$
6: SimpleTreeMatching($T_s, T_i$);
7: $L = \text{alignTrees}(T_s, T_i);$ \{based on the result from line 6\}
8: if $T_i$ is not completely aligned with $T_s$ then
9: $I = \text{InsertIntoSeed}(T_s, T_i);$ 
10: if not all unaligned items in $T_i$ are inserted into $T_s$ then
11: insert $T_i$ into $R$
12: end if
13: end if
14: if ($L$ has new alignment) or ($I = true$) then
15: flag = true
16: end if
17: if $S = \emptyset$ and flag = true then
18: $S = R; R = \emptyset; flag = false; I = false$
19: end if
20: end while
21: output data items from each $T_i$ to a database table based on the alignment result.

Figure 20. The partial tree alignment algorithm
Figure 21. Iterative tree alignment with two iterations

and g are not matched to $T_s$. Line 9 attempts to insert them into $T_s$. This is the partial tree alignment discussed above. In Figure 21, none of the nodes n, c, k and g in $T_2$ can be inserted into $T_s$ because no unique location can be found. Line 11 inserts $T_2$ into $R$, which is a list of trees that may need to be further processed. In Figure 21, when matching $T_3$ with $T_s$, all unmatched nodes c, h and k can be inserted into $T_s$. Thus, $T_3$ will not be inserted into $R$.

Lines 14-16 set "flag = true" to indicate that some new alignments/matches are found or some unmatched nodes are inserted into $T_s$. Lines 17-19 check for stopping conditions. "$S = \emptyset$ and flag = true" means that we have processed all the trees in $S$, and some new alignments are found or insertions are done. Then trees in $R$ should be processed again. In Figure 21, $T_2$ is the only tree in $R$, which will be matched to the new $T_s$ in the next round. Now every node in
$T_2$ can be matched or inserted. The process completes. Line 21 outputs the data items from each tree according to the alignment produced. Note that if there are still un-matched nodes with data after the algorithm completes, each un-matched data will occupy a single column by itself. Table I shows the data table for the trees in Figure 21. We use "1" to indicate a data item.

\begin{table}[h]
\centering
\caption{FINAL DATA TABLE ("1" INDICATES A DATA ITEM)}
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
 & ... & x & b & n & c & d & h & k & g \\
\hline
$T_1$ & ... & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
$T_2$ & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
$T_3$ & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\hline
\end{tabular}
\end{table}

The complexity of the algorithm is $O(k^2)$ without considering tree matching, where $k$ is the number of trees. However, in practice, almost always we only need to go through $S$ once (i.e., $R = \emptyset$). It should be noted that the resulting alignment $T_s$ can also be used as an extraction pattern for extracting data items from other pages generated using the same template.
CHAPTER 5

EMPIRICAL EVALUATIONS

This section evaluates our system, DEPTA (Data Extraction based on Partial Tree Alignment), which implements the proposed techniques. We conducted experiments on pages collected from 50 different Web sites. These sites are the ones used by other systems (RISE \footnote{http://www.isi.edu/muslea/RISE/}, OMINI \footnote{http://www.cs.uic.edu/yzhai/data/} [3], IEPAD [5], ROADRUNNER [8], EXALG [1]) and some well-known sites from a wide range of domains. The whole dataset and extraction result are available for download\footnote{http://www.isi.edu/muslea/RISE/}. The system is fully automatic without human intervention. The input is a single page and the output is: (i) A new page that contains a list of data regions. Each data region contains a list of data records in the input page. (ii) An Excel file that contains a list of tables. Each table corresponds to a data region with data items of the same attribute aligned in the same column. Note that currently the system can not automatically label aligned data items. This problem is addressed in [17] but the solution proposed is not general enough. This is also one of our future work.

The evaluation consists of two parts:

1. Data record extraction (step 1): Given a single page, extract lists of data records (e.g., products, news) in it.
2. Data items alignment and extraction (step 2): If there are any data records extracted out in step 1, align and extract data items from them.

The comparison of step 1 (Mdr, old version of DEPTA, which realize the step of data record extraction) with two existing systems, OMINI [3] and IEPAD [5] is conducted in [19]. Table II shows the comparison result. For the detailed information of the comparison, please see [19]. For the step of data item extraction, [5] is able to perform the same task. However, as shown in [19], it performs unsatisfactory in finding right data records, and thus could not extract data items well. We do not compare with the system in [8; 1; 18] as they require multiple pages that conform to a common schema as the input.

<table>
<thead>
<tr>
<th>number of sites</th>
<th>number of data records</th>
<th>MDR</th>
<th>OMINI</th>
<th>IEPAD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>correct</td>
<td>found</td>
</tr>
<tr>
<td>46</td>
<td>621</td>
<td>620</td>
<td>242</td>
<td>432</td>
</tr>
<tr>
<td>Recall</td>
<td></td>
<td></td>
<td>99.8%</td>
<td>39%</td>
</tr>
<tr>
<td>Precision</td>
<td></td>
<td></td>
<td>100%</td>
<td>56%</td>
</tr>
</tbody>
</table>

5.1 Tree distance threshold

We used a number of training pages in building our systems and in selecting our default distance threshold. The training pages were not used in testing our system. Based on the
training result, we select a normalized tree distance threshold \( T = 0.4 \), which is set as the default value for our system and used in all our experiment.

5.2 Experimental Result

We show in Table III the performance of DEPTA on data record extraction and data item alignment.

Column \( a \) denotes the numbers of actual data records and data items. For comparison purpose, these numbers are manually counted. Note that we only count the data records which contain valued information. Some area such as navigation bars in a page also contains data with regular patterns and they will be identified by our proposed technique. DEPTA takes into consideration the visual information of identified data records to decide wether to output them or not. For example, if a list of data records locate on the boundary part of a page and each occupies a comparatively small area, they are considered as unimportant data and will not be outputted. The criteria used to decide the importance of a list of data records is similar to the ones presented in [26]. This part is not the focus of our work, so we will not go into the details.

Columns \( c \) denotes the numbers of correct data records and data items extracted by DEPTA.

Columns \( w \) denotes the numbers of data records and data items extracted/aligned incorrectly by DEPTA. For a data record, incorrect extraction usually has the following cases:

1. only part of the content of the data record is extracted;

2. information outside of the data record boundary are extracted and enclosed in it;

For a data item, incorrect alignment usually happens in the following cases:
### TABLE III

<table>
<thead>
<tr>
<th>Site</th>
<th>Data Records</th>
<th>Data Items</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>buy</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>yahoo</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>ebay</td>
<td>50</td>
<td>49</td>
</tr>
<tr>
<td>froogle</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>cnet</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>zdnet</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>slashdot</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>usopen</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>uefa</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>wine</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>excite</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>altavista</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>hotbot</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>lycos</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>gateway</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Barnes &amp; Noble</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>abtelectronics</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>acehardware</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>adesso</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>alibris</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>amazon</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>asford</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>bargainoutfitter</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>bestbuy</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>bobsdiscountmart</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>cameraworld</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>circuitcity</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>cooking</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>compusa</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>dealtime</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>drugstore</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>essentialapparel</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>magazinesofamerica</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>nextag</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>nothingbutsoftware</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>megella</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>officedepot</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
**TABLE IV**

**EXPERIMENT RESULTS (CONTINUED)**

<table>
<thead>
<tr>
<th>Site</th>
<th>Data Records</th>
<th>Data Items</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>overstock</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>pricegrabber</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>radioshack</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>randomhouse</td>
<td>25</td>
<td>24</td>
</tr>
<tr>
<td>refurbdepot</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>rochesterclothing</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>shoebuy</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>shopping</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>smartbargain</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>target</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>tigerdirect</td>
<td>18</td>
<td>14</td>
</tr>
<tr>
<td>kmart</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>sonystyle</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>670</td>
<td>653</td>
</tr>
<tr>
<td><strong>Recall</strong></td>
<td>97.46%</td>
<td></td>
</tr>
<tr>
<td><strong>Precision</strong></td>
<td>98.79%</td>
<td></td>
</tr>
</tbody>
</table>
1. items representing the same attribute are placed into different columns;

2. items of different attributes are placed into the same column;

Columns \( m \) denotes the numbers of data records and data items that were not identified and extracted by DEPTA.

The last three rows of Table III give the total of each column, the recall and precision of the system. For data record extraction, the recall and precision are computed based on the total number of correct data records found in all pages and the actual number of data records in these pages. For data item extraction, the precision and recall computation has considered the 8 incorrect and 9 missing data records introduced in step 1 of DEPTA.

5.3 Evaluation

As it can be seen from the table, for a large majority of pages the system was able to extract data records and align data items correctly. If not consider the error introduced by step 1 of DEPTA, the recall and precision of data item alignment will be higher. In some cases the system is unable to extract data records correctly. There are two main sources for these behaviors, namely:

1. If an object is very dissimilar to the surrounding objects, it will be considered as a separator or a commercial and will not be extracted.

2. In a DOM tree, if an data record is not under the same parent with other similar data records, it can not be identified. This kind of page rarely exists.
There are some pages where a single attribute is encoded into more than one HTML tag. To avoid segmenting one attribute into several columns, a preprocessing is exploited to remove formatting HTML tags embedded in plain text. There are some cases in which the system is unable to align data items correctly:

1. Data items of the same attribute are enclosed by different HTML tags, which results them being placed in different columns.

2. Sometimes several data items are encoded together into one text string that is not separated by HTML tags. Parsing phrases and segmenting them is not the focus of our current work and there are several existing techniques capable of doing this. This problem can be solved by a postprocessing.
CHAPTER 6

CONCLUSION

In this work, we proposed a new approach to extract structured data from Web pages. Although the problem has been studied by several researchers, existing techniques are either inaccurate or make many assumptions. Our method does not make these assumptions. It only requires that the page contains more than one data record, which is almost always true for pages with data records. Our technique consists of two steps: (1) identifying data records without extracting data items in the records, and (2) aligning corresponding data items from multiple data records and put them in a database. We proposed an enhanced method based on visual information for step 1. For step 2, we proposed a novel partial tree alignment technique to align corresponding data fields of multiple data records. Empirical results using a large number of Web pages show the effectiveness of the proposed technique.
CHAPTER 7

FUTURE WORK

7.1 Nested Data Extraction and Schema Generation

The DEPTA framework has the potential to handle nested data records (records in records) due to its nested structure of similarity comparison. We did not explicitly handle such data records in current work. However, such ability can be realized by changing the order in which we traverse the tree.

Figure 22 shows an example page containing nested data. There are two categories of products in the page: canning jars and canning tools. Each category contains one or multiple products and each product contains one or multiple detailed information depending on the product models. These nested data can be represented by a schema: 

\[ ((\text{image} \ (\text{size}, \text{type}, \text{ranking}, \text{price})^+)^+)^+ , \]

where the outmost '+' denotes the repetition of categories, the middle '+' denotes the repetition of products and the innermost '+' denotes the repetition of detailed information for each model. In order to capture this situation, the new method needs to do a postorder traversal (bottom-up) of DOM-trees instead of a preorder traversal which is used in our current implementation. During the postorder traversal, the nested data at lower level (innermost) will be identified first. For each sequence of sibling nodes, a pattern can be generated that describes the repeating, optional and disjunctive cases. A schema can then be formed when the traversal reaches the root node.
7.2 Labeling Extracted Data

Most current work is deficient in providing users the meaning of the attributes of the extracted data. This problem is addressed in [30; 17] but the solution proposed is not general enough. In [30], queries are sent out through complex search forms and the search results are used for data extraction and labeling. However, most Web sites do not provide complex search forms and therefore the use of this method is limited.

The problem of labeling extracted data can be formally defined as the following: each extracted data record is denoted as $R = < i_1 i_2 ... i_N >$ where $i_k (1 \leq k \leq N)$ represents the $k$th data item. We need to assign an attribute name from the attribute set $A = \{a_1, a_2, ..., a_M\} (M \gg N)$ to each data item $i_k$ to determine the corresponding label sequence $L = < l_1 l_2 ... l_N >$. 
In our work, the extracted items aligned in columns may contain some meta data. For example, in two adjacent columns, all the cells in the first column are “our price:”, and the cells in the second column contain different “dd.dd” values. In such case, the values in the first column are the labels (meta data) of the values in the second column. We can partially label the data items in this way. For the rest of data items, a probabilistic model can be used to find their best labels. In a probabilistic framework, the labeling task can be defined as the following:

Given an observation $o$ (a data item) and a probabilistic model $\lambda$, find a label $l^*$ that gives the highest posterior probability,

$$l^* = \max_l p(l|o, \lambda)$$

$\lambda$ could be a probabilistic model such as HMMs (Hidden Markov Models), MEMMs (Maximum Entropy Markov Models), CRF [16] (Conditional Random Fields) and etc. Among them, CRF is the state of art approaches in information extraction that gives better performance on data labeling than other existing models. We will investigate the potential use of these probabilistic models, especially CRF, in our system for data labeling task.

7.3 Extracting Data from Detail Pages

There are mainly two types of “data-rich” pages, one is list pages (the pages containing at least two objects with similar patterns) and another is detail pages (the pages containing detailed information of one object). Our current algorithm is only applicable to list pages. Extracting data from a single detail page heavily depends on the page domain. For example, in the news domain, we can extract news title and news body by utilizing some domain characteristics such as:
1. news bodies usually contain comparatively less hyperlinks and longer text.

2. new titles are usually located (geometrically) on top of new bodies.

It is very difficult to extract data from a single detail page without training or domain knowledge (what do the data describe). Most existing techniques need multiple pages that conform to a common schema as the input, and attempt to infer the schema by comparing these pages. The inferred schema is then used to extract data. How to construct a framework that can be used to extract data from a single page, and how to design algorithms that can be conveniently applied to Web pages in various domains is an interesting research problem. We are currently investigating this problem and will keep pursuing this direction in the future.


