FiVaTech: Page-Level Web Data Extraction from Template Pages

Mohammed Kayed and Chia-Hui Chang, *Member*, *IEEE*

Abstract—Web data extraction has been an important part for many Web data analysis applications. In this paper, we formulate the data extraction problem as the decoding process of page generation based on structured data and tree templates. We propose an unsupervised, page-level data extraction approach to deduce the schema and templates for each individual Deep Website, which contains either singleton or multiple data records in one Webpage. FiVaTech applies tree matching, tree alignment, and mining techniques to achieve the challenging task. In experiments, FiVaTech has much higher precision than EXALG and is comparable with other record-level extraction systems like ViPER and MSE. The experiments show an encouraging result for the test pages used in many state-of-the-art Web data extraction works.

Index Terms—Semistructured data, Web data extraction, multiple trees merging, wrapper induction.

1 Introduction

Deep Web, as is known to everyone, contains magnitudes more and valuable information than the surface Web. However, making use of such consolidated information requires substantial efforts since the pages are generated for visualization not for data exchange. Thus, extracting information from Webpages for searchable Websites has been a key step for Web information integration. Generating an extraction program for a given search form is equivalent to wrapping a data source such that all extractor or wrapper programs return data of the same format for information integration.

An important characteristic of pages belonging to the same Website is that such pages share the same template since they are encoded in a consistent manner across all the pages. In other words, these pages are generated with a predefined template by plugging data values. In practice, template pages can also occur in surface Web (with static hyperlinks). For example, commercial Websites often have a template for displaying company logos, browsing menus, and copyright announcements, such that all pages of the same Website look consistent and designed. In addition, templates can also be used to render a list of records to show objects of the same kind. Thus, information extraction from template pages can be applied in many situations.

What's so special with template pages is that the extraction targets for template Webpages are almost equal to the data values embedded during page generation. Thus, there is no need to annotate the Webpages for extraction targets as in nontemplate page information extraction (e.g., Softmealy [5], Stalker [9], WIEN [6], etc.) and the key to automatic extraction depends on whether we can deduce the template automatically.

Generally speaking, templates, as a common model for all pages, occur quite fixed as opposed to data values which vary across pages. Finding such a common template requires multiple pages or a single page containing multiple records as input. When multiple pages are given, the extraction target aims at page-wide information (e.g., RoadRunner [4] and EXALG [1]). When single pages are given, the extraction target is usually constrained to record-wide information (e.g., IEPAD [2], DeLa [11], and DEPTA [14]), which involves the addition issue of record-boundary detection. Page-level extraction tasks, although do not involve the addition problem of boundary detection, are much more complicated than record-level extraction tasks since more data are concerned.

A common technique that is used to find template is alignment: either string alignment (e.g., IEPAD, RoadRunner) or tree alignment (e.g., DEPTA). As for the problem of distinguishing template and data, most approaches assume that HTML tags are part of the template, while EXALG considers a general model where word tokens can also be part of the template and tag tokens can also be data. However, EXALG's approach, without explicit use of alignment, produces many accidental equivalent classes, making the reconstruction of the schema not complete.

In this paper, we focus on page-level extraction tasks and propose a new approach, called FiVaTech, to automatically detect the schema of a Website. The proposed technique presents a new structure, called fixed/variant pattern tree, a tree that carries all of the required information needed to identify the template and detect the data schema. We combine several techniques: alignment, pattern mining, as well as the idea of tree templates to solve the much difficult problem of page-level template construction. In experiments, FiVaTech has much higher precision than EXALG, one of the few page-level extraction system, and is

- *M. Kayed is with the Department of Mathematics, Faculty of Science, Beni-Suef University, Beni-Suef, Egypt. E-mail: mskayed@yahoo.com.*
- *C.-H. Chang is with the Department of Computer Science and Information Engineering, National Central University, No. 300, Jungda Rd, Jhongli City, Taoyuan, Taiwan 320, ROC. E-mail: chia@csie.ncu.edu.tw.*

Manuscript received 10 Jan. 2008; revised 23 Nov. 2008; accepted 10 Mar. 2009; published online 31 Mar. 2009.
Recommended for acceptance by B. Moon.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2008-01-0015.
Digital Object Identifier no. 10.1109/TKDE.2009.82.

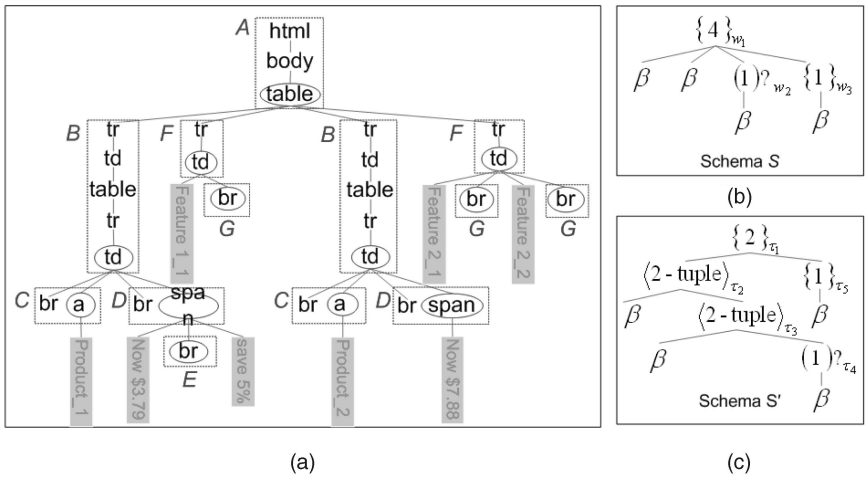

Fig. 1. (a) A Webpage and its two different schemas (b) S and (c) S'.

comparable with other record-level extraction systems like ViPER and MSE.

The rest of the paper is organized as follows: Section 2 defines the data extraction problem. Section 3 provides the system framework as well as the detailed algorithm of FiVaTech, for constructing the fixed/variant pattern tree with an example. Section 4 describes the details of template and Website schema deduction. Section 5 describes our experiments. Section 6 compares FiVaTech with related Web data extraction techniques. Finally, Section 7 concludes the paper.

2 Problem Formulation

In this section, we formulate the model for page creation, which describes how data are embedded using a template. As we know, a Webpage is created by embedding a data instance x (taken from a database) into a predefined template. Usually a CGI program executes the encoding function that combines a data instance with the template to form the Webpage, where all data instances of the database conform to a common schema, which can be defined as follows (a similar definition can also be found at EXALG [1]):

Definition 2.1 (Structured data). *A data schema can be of the following types:*

1. *A basic type β represents a string of tokens, where a token is some basic units of text.*
2. *If $\tau_1, \tau_2, \ldots, \tau_k$ are types, then their ordered list $<\tau_1, \tau_2, \ldots, \tau_k>$ also forms a type τ. We say that the type τ is constructed from the types $\tau_1, \tau_2, \ldots, \tau_k$ using a type constructor of order k. An instance of the k-order τ is of the form $<x_1, x_2, \ldots, x_k>$, where $x_1, x_2, \ldots, x_k$ are instances of types $\tau_1, \tau_2, \ldots, \tau_k$, respectively. The type τ is called*
   a. *A tuple, denoted by $<k>_\tau$, if the cardinality (the number of instances) is 1 for every instantiation.*
   b. *An option, denoted by $(k)?_\tau$, if the cardinality is either 0 or 1 for every instantiation.*
   c. *A set, denoted by $\{k\}_\tau$, if the cardinality is greater than 1 for some instantiation.*
   d. *A disjunction, denoted by $(\tau_1|\tau_2|\ldots|\tau_k)_\tau$, if all $\tau_i (i = 1, \ldots, k)$ are options and the cardinality sum of the k options $(\tau_1 - \tau_k)$ equals 1 for every instantiation of τ.*

Example 2.1. Fig. 1a shows a fictional Webpage that presents a list of products. For each product, a product name, a price, a discount percent (optional), and a list of features are presented (shaded nodes in the figure.). The data instance here is {<"Product_1," "Now $3.79," "save 5 percent," "Feature 1_1">, < "Product_2," "now $7.88," ϵ, {"Feature 2_1," "Feature 2_2"}> }, where ϵ denotes the empty string, which is missed in the second product. This data instance embedded in the page of Fig. 1a can be expressed by two different schemas S and S' as shown in Figs. 1b and 1c, respectively. Fig. 1b shows a set w_1 of order 4 (denoting the list of products in Fig. 1a): the first two components are basic types (the name and the price of the product), the third component is an option w_2 (the discount percent), and the last component is a set w_3 (a list of features for the product).

In addition to this succinct representation, the same data can also be organized by their parent nodes in its Document Object Model (DOM) tree. That is, we can reorganize the above data instance as {<<"Product_1," <"now $3.79," "Save 5 percent">>, {"Feature 1_1"}> , <<"Product_2," <"Now $7.88," ϵ>>, {"Feature 2_1," "Feature 2_2"}> }, which can be expressed by the schema S'. The second basic data and the optional data (τ_4) form a 2-tuple τ_3 (since the price and the optional discount percent of each product are embedded under the same parent node in the Webpage), which further conjugates with the first basic data (the product name) to form another 2-tuple (τ_2). Thus, the root of the new schema S' is a 2-set (τ_1), which consists of two components τ_2 and τ_5 (1-set) as shown in Fig. 1c.

As mentioned before, template pages are generated by embedding a data instance in a predefined template via a CGI program. Thus, the reverse engineering of finding the

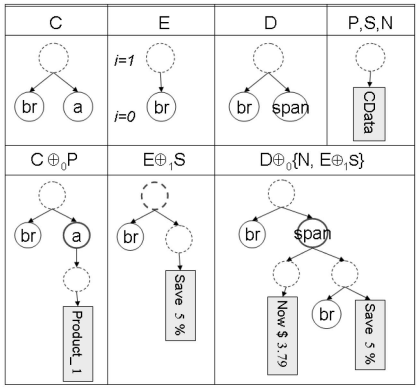

Fig. 2. Examples for tree concatenation.

template and the data schema given input Webpages should be established on some page generation model, which we describe next. In this paper, we propose a tree-based page generation model, which encodes data by subtree concatenation instead of string concatenation. This is because both data schema and Webpages are tree-like structures. Thus, we also consider templates as tree structures. The advantage of tree-based page generation model is that it will not involve ending tags (e.g., </html>, </body>, etc.) into their templates as in string-based page generation model applied in EXALG.

Concatenation is a required operation in page generation model since subitems of data must be encoded with templates to form the result. For example, encoding of a k-order type constructor τ with instance x should involve the concatenation of template trees T, with all the encoded trees of its subitems for x. However, tree concatenation is more complicate since there is more than one point to append a subtree to the rightmost path of an existing tree. Thus, we need to consider the insertion position for tree concatenation.

Definition 2.2. *Let T_1 and T_2 be two trees, we define the operation $T_1 \oplus_i T_2$ as a new tree by appending tree T_2 to the rightmost path of T_1 at the ith node (position) from the leaf node.*

For example, given the templates trees C, E and data contents P, S (for content data "$Product\_1$" and "Save 5 percent," respectively) on top half of Fig. 2, we show the tree concatenation $C \oplus_0 P$ and $E \oplus_1 S$ on bottom half of the same figure. The dotted circles for these trees are virtual nodes, which facilitate tree representation (e.g., connecting multiple paths into trees) and can be neglected. The insertion points are marked with blue solid circle. For subtree C, the insertion point is node <a>, where the subtree P (single node) is inserted. For subtree E, the insertion point is one node above the
 node, i.e., the virtual root, where the subtree S (also a single node) is appended to. We also show two subtrees N (for content data "Now \$3.79") and $E \oplus_1 S$ inserted as sibling under template D at insertion point 0. We denote this operation by $D \oplus_0 \{N, E \oplus_1 S\}$.

With the tree-concatenation operation, we can now define the encoding of a k-order type constructor in a way similar to that in EXALG. Basically, the idea is to allow

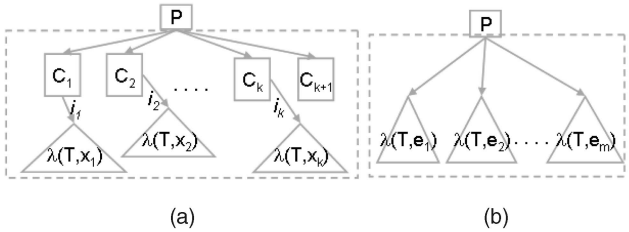

Fig. 3. Encoding of data instances for a type constructor. (a) Single instance $\mathrm{e} = (\mathrm{X}_1, \ldots, \mathrm{X}_\mathrm{k})$. (b) Multiple instance $\mathrm{e}_1, \ldots, \mathrm{e}_\mathrm{m}$.

$k+1$ templates to be placed in front of, in between, and in the end of the k subitems as follows:

Definition 2.3 (Level-aware encoding). *We define the template for a type constructor τ as well as the encoding of its instance x (in terms of encoding of subvalues of x) as described below.*

1. *If τ is of a basic type, β, then the encoding $\lambda(T, x)$ is defined to be a node containing the string x itself.*
2. *If τ is a type constructor of order k, then the template is denoted by: $T(\tau) = [P, (C_1, \ldots, C_{k+1}), (i_1, \ldots, i_k)]$, where $P, C_1, \ldots,$ and C_{k+1} are template trees.*
   a. *For single instance x of the form $(x_1, \ldots, x_k)$, $\lambda(T_\tau, x)$ is a tree produced by concatenating the $k+1$ ordered subtrees, $C_1 \oplus_{i_1} \lambda(T, x_1)$, $C_2 \oplus_{i_2} \lambda(T, x_2), \ldots, C_k \oplus_{i_k} \lambda(T, x_k)$, and C_{k+1} at the leaf on the rightmost path of template P. See Fig. 3a for illustration on single instance encoding.*
   b. *For multiple instances $e_1, e_2, \ldots, e_m$ where each e_i is an instance of type τ, the encoding $\lambda(T, e_1, e_2, \ldots, e_m)$ is the tree by inserting the m subtrees $\lambda(T, e_1), \lambda(T, e_2), \ldots, \lambda(T, e_m)$ as siblings at the leaf node on the rightmost path of the parent template P. Each subtree $\lambda(T, e_i)$ is produced by encoding e_i with template [ϕ, $(C_1, \ldots, C_{k+1})$, $(i_1, \ldots, i_k)$] using the procedure for single instance as above; ϕ is the null template (or a virtual node). See Fig. 3b for illustration on multiple instances encoding.*
   c. *For disjunction, no template is required since the encoding of an instance x will use the template of some τ_i $(1 \le i \le k)$, where x is an instance of τ_i.*

Example 2.2. We now consider the schema $S' = \{< \beta, < \beta, (\beta)?_{\tau 4} >_{\tau 3} >_{\tau 2}, \{\beta\}_{\tau 5}\}_{\tau 1}$ for the input DOM tree in Fig. 1a. We circumscribe adjoining HTML tags into template trees $A - G$ (rectangular boxes). Most of the templates are single paths, while templates C and D contain two paths. Thus, a virtual node is added as their root to form a tree structure as shown in Fig. 2. Traversing the tree in a depth-first order to give the templates and data an order, we can see that most data items are inserted at the leaf nodes of their preceding templates, e.g., item "$Product\_1$" is appended to template tree C at the leaf nodes with insertion position 0, which is equivalent to the $C \oplus_0 P$ operation in Fig. 2; similarly, "$Now\$3.79$," "$Feature1\_1$" are appended to template trees D and F, respectively, at the leaf nodes with insertion position 0. Still some have a different

position, e.g., "$Save$ 5 percent" is appended to one node above the leaf node ($i = 1$) on the rightmost path of template tree E, which is equivalent to the operation $E \oplus_1 S$ shown in Fig. 2.

We can now write down the templates for each type constructor of the schema S' by corresponding it to the respective node in the DOM tree. For example, the encoding of τ_4 with data instance "$Save$ 5 percent" can be completed by using template $T(\tau_4) = [\phi^1, (E, \phi), 1]$, i.e., the tree concatenation example $E \oplus_1 S$ in Fig. 2 after removing virtual nodes. Similarly, the encoding of τ_3 with data instance <"now \$3.79," "$save$ 5 percent"> can be completed by using template $T(\tau_3) = [\phi, (\phi, \phi, \phi), (0, 0)]$. This result when preconcatenated with tree template D at insertion position 0 produces $D \oplus_0 \{N, E \oplus_1 S\}$ in Fig. 2, a subtree for τ_2. The other subtree for τ_2 is $C \oplus_0 P$. Thus, template for τ_2 can be written as $T(\tau_2) = [\phi, (C, D, \phi), (0, 0)]$. As we can see, most parent templates are empty tree template except for the root type τ_1, which has template $T(\tau_1) = [A, (B, F, \phi), (0, 0)]$. The last type constructor τ_5 has template $T(\tau_5) = [\phi, (\phi, G), 0]$.

This encoding schema assumes a fixed template for all data types. In practice, we sometimes can have more than one template for displaying the same type data. For example, displaying a set of records in two columns of a Webpage requires different templates for the same type data records (template for data records on the left column may be different from template of data records on the right although all of them are instances of the same type). However, the assumption of one fixed template simplifies the problem. Such data with variant templates can be detected by postprocessing the deduced schema to recognize identical schema subtrees, thus, we shall assume fixed-template data encoding in this paper.

Meanwhile, as HTML tags are essentially used for presentation, we can assume that basic type data always reside at the leaf nodes of the generated trees. Under this premise, if basic type data can be identified, type constructors can be recognized accordingly. Thus, we shall first identify "variant" leaf nodes, which correspond to basic data types. This should also include the task of recognizing text nodes that are part of the template (see the "Delivery:" text node in Fig. 13).

Definition 2.4 (Wrapper induction). *Given a set of n DOM trees, $DOM_i = \lambda(T, x_i)$ $(1 \leq i \leq n)$, created from some unknown template T and values $x_1, \ldots, x_n$, deduce the template, schema, and values from the set of DOM trees alone. We call this problem a page-level information extraction. If only one single page $(n = 1)$ that contains tuple constructors is given, the problem is to deduce the template for the schema inside the tuple constructors. We call this problem a record-level information extraction task.*

3 FiVaTech Tree Merging

The proposed approach FiVaTech contains two modules: tree merging and schema detection (see Fig. 4). The first module merges all input DOM trees at the same time into a structure called fixed/variant pattern tree, which can then be used to detect the template and the schema of the Website in the second module. In this section, we will introduce how input DOM trees can be recognized and merged into the pattern tree for schema detection.

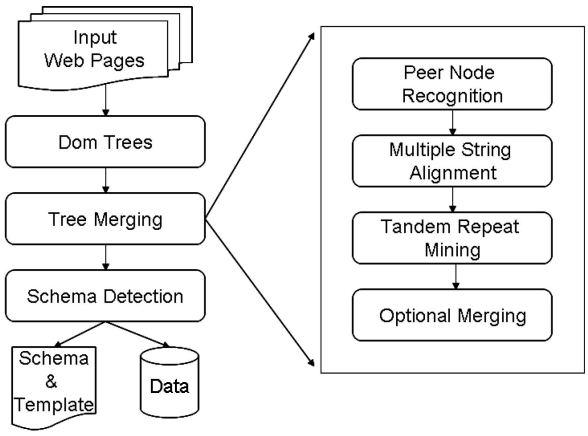

Fig. 4. The FiVaTech approach for wrapper induction.

According to our page generation model, data instances of the same type have the same path from the root in the DOM trees of the input pages. Thus, our algorithm does not need to merge similar subtrees from different levels and the task to merge multiple trees can be broken down from a tree level to a string level. Starting from root nodes <html> of all input DOM trees, which belong to some type constructor we want to discover, our algorithm applies a new multiple string alignment algorithm to their first-level child nodes.

There are at least two advantages in this design. First, as the number of child nodes under a parent node is much smaller than the number of nodes in the whole DOM tree or the number of HTML tags in a Webpage, thus, the effort for multiple string alignment here is less than that of two complete page alignments in RoadRunner [4]. Second, nodes with the same tag name (but with different functions) can be better differentiated by the subtrees they represent, which is an important feature not used in EXALG [1]. Instead, our algorithm will recognize such nodes as *peer nodes* and denote the same symbol for those child nodes to facilitate the following string alignment.

After the string alignment step, we conduct pattern mining on the aligned string S to discover all possible repeats (set type data) from length 1 to length $|S|/2$. After removing extra occurrences of the discovered pattern (as that in DeLa [11]), we can then decide whether data are an option or not based on their occurrence vector, an idea similar to that in EXALG [1]. The four steps, peer node recognition, string alignment, pattern mining, and optional node detection, involve typical ideas that are used in current research on Web data extraction. However, they are redesigned or applied in a different sequence and scenario to solve key issues in page-level data extraction.

As shown in Fig. 5, given a set of DOM trees T with the same function and its root node P, the system collects all (first-level) child nodes of P from T in a matrix M, where each column keeps the child nodes for every peer subtree of P. Every node in the matrix actually denotes a subtree, which carries structure information for us to differentiate its

1. ϕ denotes the empty tree template (thus, simply a virtual node).

Algorithm **MultipleTreeMerge**(T, P)
// T is a set of DOM trees of the same type;
// P is the tag for the roots of T.
1. Initialize M; i = 0;
2. **for** each tree t in T
3. $j = 0$;
4. **for** each child c in t
5. $M[j{+}{+}][i] = c$;
6. **endfor**
7. $i{+}{+}$;
8. **endfor**
9. $recognizePeerNode(M)$;
10. $childList = matrixAlignment(M)$;
11. $childList = repeatMining(childList, 1)$;
12. $mergeOptional(childList)$;
13. **for** each node c in childList
14. **if** (c is a tree) **then**
15. $C = multipleTreeMerg(peerNode(c, M), tag(c))$;
16. **else** // c is a leaf node
17. $C = c$;
18. **endif**
19. Insert C as a child of P;
20. **endfor**
21. **return** pattern tree P;

FIg. 5. Multiple tree merging algorithm.

role. Then, we conduct the four steps: peer node recognition, matrix alignment, pattern mining, and optional node detection in turn.

- In the peer node recognition step (line 9), two nodes with the same tag name are compared to check if they are peer subtrees. All peer subtrees will be denoted by the same symbol.
- In the matrix alignment step (line 10), the system tries to align nodes (symbols) in the peer matrix to get a list of aligned nodes childList. In addition to alignment, the other important task is to recognize variant leaf nodes that correspond to basic-typed data.
- In the pattern mining step (line 11), the system takes the aligned childList as input to detect every repetitive pattern in this list starting with length 1. For each detected repetitive pattern, all occurrences of this pattern except for the first one are deleted for further mining of longer repeats. The result of this mining step is a modified list of nodes without any repetitive patterns.
- In the last step (line 12), the system recognizes optional nodes if a node disappears in some columns of the matrix and group nodes according to their occurrence vector.

After the above four steps, the system inserts nodes in the modified childList as children of P. For nonleaf child node c, if c is not a fixed template tree (as defined in the next section), the algorithm recursively calls the tree merging algorithm with the peer subtrees of c (by calling procedure $peerNode(c, M)$, which returns nodes in M having the same symbol of c) to build the pattern tree (line 14). The next four sections will discuss in details recognition of peer subtrees, multiple string alignment, frequent pattern mining, and merging of optional nodes, which are applied for each node in constructing the fixed/variant pattern tree (lines 9-12).

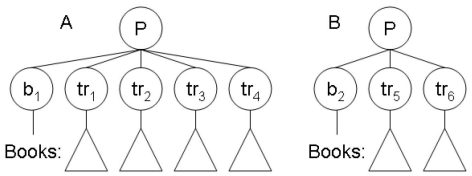

Fig. 6. Example of set-type data.

3.1 Peer Node Recognition

One of the key issues for misalignment among multiple pages (or multiple records in a single page) is that the same HTML tags can have different meanings (we do not consider different HTML tags with same meaning since we assume that templates are fixed for the same data as discussed above). As each tag/node is actually denoted by a tree, we can use 2-tree matching algorithm for computing whether two nodes with the same tag are similar. There are several 2-tree matching algorithms proposed before. We adopt Yang's algorithm [13] in which level crossing is not allowed (i.e., two tag nodes in two trees are matched only if they appear at the same level) while node replacement is allowed only on the roots of the two trees (i.e., two trees may be matching even their root nodes are labeled by different HTML tags). To fit our problem, we modified the algorithm such that node replacement is allowed at leaves instead of roots. Thus, two leaf nodes can be matched even if they have different text values. The running time of the algorithm is still $O(s_1 s_2)$, where s_1 and s_2 are the numbers of nodes of the trees.

A more serious problem is score normalization. Traditional 2-tree matching algorithm returns the number of maximum matching, which requires normalization for comparison. A typical way to compute a normalized score is the ratio between the numbers of pairs in the mapping over the maximum size of the two trees as is used in DEPTA [14]. However, the formula might return a low value for trees containing set-type data. For example, given the two matched trees A and B as shown in Fig. 6, where $tr_1 - tr_6$ are six similar data records, we assume that the mapping pairs between any two different subtrees tr_i and tr_j are 6. Thus, the tree matching algorithm will detect a mapping that includes 15 pairs: 6×2 pairs from tr_1 and tr_2 (matched with tr_5 and tr_6, respectively), 2 pairs from b_1, and finally, a mapping from the root of A to the root of B. Assume also that the size of every tr_i is approximately 10. According to such a measure, the matching score between the two trees A and B will be $15/43(\simeq 0.35)$, which is low.

For Web data extraction problem, where set-type data occur, it is unfair to use the maximum size of the two trees to compute the matching score. Practically, we noticed that this multiple-valued data have great influence on first-level subtrees of the two trees A and B. Thus, we propose FiVaTreeMatchingScore (Fig. 7) to handle this problem without increasing the complexity as follows: If the two root nodes A and B are not comparable (i.e., have different labels and both are not text nodes), the algorithm gives a score 0 for the two trees (line 2). If either of the two root nodes has no children (line 3) or they are of the same size, the algorithm computes the score as the ratio of TreeMatching(A, B) over the average of the two tree sizes (line 4),

Procedure **FiVaTreeMatchScore(**A, B**)**
// A and B are two trees;
1. **if** (the roots of A and B are not comparable) **then**
2. **return** 0;
3. **if** ((either A or B is a leaf) or size(A)==size(B)) **then**
4. **return** $\frac{2*TreeMatching(A,B)}{size(A)+size(B)}$;
5. $score$ = 0.0; m = # of children in A;
6. **for** each $child_a$ in A **do**
7. $nodeScore$ = 0.0; $matchNo$ = 0;
8. **for** each $child_b$ in B **do**
9. $temp$= $\frac{2*TreeMatching(child_a,child_b)}{size(child_a)+size(child_b)}$
10. **if** ($temp > \theta$) **then**
11. $nodeScore$ += $temp$; $matchNo$++;
12. **endif**
13. **endfor**
14. **if** ($matchNo > 0$) **then** $nodeScore$ = $\frac{nodeScore}{matchNo}$;
15. $score$ += $nodeScore$;
16. **endfor**
17. **return (**$\frac{score}{m} + \frac{2}{size(A)+size(B)}$**);**

Fig. 7. FiVaTech tree matching score algorithm.

where TreeMatching(A, B) returns the number of matching nodes between A and B. If the two root nodes are comparable and both of them have children, the algorithm matches each $child_a$ of tree A with every $child_b$ of tree B (lines 6-16). If the ratio of $TreeMatching(child_a, child_b)$ to the average size of the two subtrees is greater than a threshold θ (=0.5 in our experiment), then they are considered as matched subtrees, and we average the score for each $child_a$ by $nodeScore/matchNo$. The algorithm finally returns the summation of the average nodeScore ($score/m$) for all children of A and the ratio between 1 and the average of the two tree sizes (line 17). The final ratio term is added because the two root nodes are matched.

As an example, the normalized score for the two trees shown in Fig. 6 will be computed as follows: The first child b_1 is only matched with b_2 in the children of the second tree, so the $nodeScore$ value for b_1 is 1.0/1 = 1.0. Each tr subtree in A matches with the two tr subtrees in B, so every one will have a $nodeScore$ value equal to $(0.6 + 0.6)/2 = 0.6$. So, the average $nodeScore$ of the children for tree A will be $(1.0 + 0.6 + 0.6 + 0.6 + 0.6)/5 = 0.68$. Thus, the score of the two trees will be $0.68 + (1/Average(43, 23)) \simeq 0.71$. The computation of matching score requires $O(n^2)$ calls to 2-tree edit distance for n trees with the same root tag. Theoretically, we don't need to recompute tree edit distance for subtrees of these n trees since those tree edit distances have been computed when we conduct peer recognition at their parent nodes. However, current implementation did not utilize such dynamic programming techniques. Thus, the execution time cost is higher on average.

3.2 Peer Matrix Alignment

After peer node recognition, all peer subtrees will be given the same symbol. For leaf nodes, two text nodes take the same symbol when they have the same text values, and two $<img>$ tag nodes take the same symbol when they have the same SRC attribute values. To convert M into an aligned peer matrix, we work row by row such that each row has (except for empty columns) either the same symbol for every column or is a text ($<img>$) node of variant text (SRC attribute, respectively) values. In the latter case, it will be marked as basic-typed for variant texts. From the aligned matrix M, we get a list of nodes, where each node corresponds to a row in the aligned matrix.

Procedure **MatrixAlignment(M)**
1. row = 1;
2. $shiftLength = 0$;
3. **while** (M is not aligned) **do**
4. **while** (! $alignedRow(row, M)$) **do**
5. $shiftColumn$ = $getShiftColumn($
$row, shiftLength, M)$;
6. $makeShift(row, shiftColumn, shiftLength, M)$;
7. **endwhile**
8. row++;
9. **endwhile**
10. $childList = alignmentResult(M)$;

Fig. 8. Matrix alignment algorithm.

As shown in Fig. 8, the algorithm traverses the matrix M row by row, starting from the first one (line 1), and tries to align every row before the matrix M becomes an aligned peer matrix (line 3). At each row, the function alignedRow checks if the row is aligned or not (line 4). If it is aligned, the algorithm will go to the next one (line 8). If not, the algorithm iteratively tries to align this row (lines 4-7). In each iteration step, a column (a node) $shiftColumn$ is selected from the current row and all of the nodes in this column are shifted downward a distance $shiftLength$ in the matrix M (at line 6 by calling the function $makeShift$) and patch the empty spaces with a null node. The function $makeShift$ is straightforward and does not need any further explanation. Now, we shall discuss the two functions $alignedRow$ and $getShiftedColumn$ in details.

The function $alignedRow$ returns true (the row is aligned) in two cases. The first case is when all of the nodes in the row have the same symbol. In this case, the row will be aligned and marked as fixed. The second case is when all of the nodes are text ($<img>$) nodes of different symbols, and each one of these symbols appears only in its residing column (i.e., if a symbol exists in a column c, then all other columns outside the current row in the matrix do not contain this symbol). In this case, the function will identify this leaf node as variant (denoted by an asterisk "$*$").

Before we describe the selection of a column to be shifted, we first define $span(n)$ as the maximum number of different nodes (without repetition) between any two consecutive occurrences of n in each column plus one. In other words, this value represents the maximum possible cycle length of the node. If n occurs at most once in each column, then we consider it as a free node and its span value will be 0. For example, the $span$ values of the nodes a, b, c, d, and e in the peer matrix M_1 (in Fig. 6) are 0, 3, 3, 3, and 0, respectively.

The function $getShiftedColumn$ selects a column to be shifted from the current row r (returns a value to $shiftColumn$) and identifies the required shift distance (assigns a value to $shiftLength$) by applying the following rules in order:

- (R1.) Select, from left to right, a column c such that the expected appearance of the node n ($= M[r][c]$) is not reached, i.e., there exists a node with the same

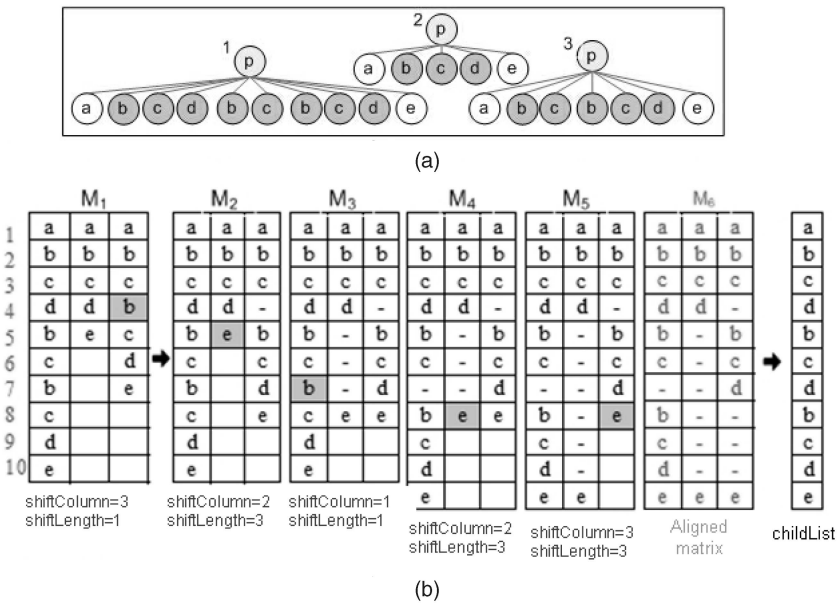

Fig. 9. An example of peer matrix alignment.

symbol at some upper row r_{up} ($r_{up} < r$), where $M[r_{up}][c'] = n$ for some c' and $r - r_{up} < span(n)$. Then, $shiftColumn$ will equal to c and $shiftLength$ will be 1.

- (R2.) If R1 fails (i.e., no column satisfies the condition in R1), then we select a column c with the nearest row r_{down} ($r_{down} > r$) from r such that $M[r_{down}][c'] = M[r][c]$ for some $c' \neq c$. In such a case, $shiftLength$ will be $r_{down} - r$.
- (R3.) If both rules R1 and R2 fail, we then align the current row individually by dividing it into two parts: P1 (aligned at row r) and P2 (not aligned at row r). In this divide-and-conquer process, the aligned symbol for P1 and P2 may be different at row r. In such cases, the part which contains symbol n with $r - r_{up} = span(n)$ should come first (r_{up} is as defined in R1).

The alignment algorithm tries to consider missing attributes (optional data), multiple-valued attributes (set data), and multiple-ordering attributes. Usually, handling such problems is a challenge, especially when they occur simultaneously. By computing the $span$ value of each symbol, we get to predict the expected location in a global view and decide the more proper symbol by the prioritized rules at current row.

Fig. 9 shows an example that describes how the algorithm proceeds. The first three rows of M_1 are aligned, so the algorithm does not make any changes on them. The fourth row in M_1 is not aligned, so the algorithm tries to align this row by iteratively making a suitable shift for some columns according to the three previous mentioned rules. According to rule R1, column 3 is selected since there is a node b at row 2 such that $4 - 2 < span(b) = 3$. Hence, matrix M_2 is obtained. Since the 4th row in M_2 is aligned now, so it goes to the next row (row 5 in M_2) and detects that it is not aligned. According to rule R2 (R1 doesn't apply here), column 2 is selected since node e has the nearest occurrence at the 8th row at column 1 ($\neq 2$). Therefore, $shiftColumn = 2$ and $shiftLength = 8 - 5 = 3$. Similarly, we can follow the selection rule at each row and get the matrices M_4, M_5, and the final aligned peer matrix M_6. Here, dashes mean null nodes. The alignment result $childList$ is shown at the rightmost of the figure, where each node in the list corresponds to a row in the aligned peer matrix M_6. In the worst case, it might take $O(r^2c^2)$ comparisons to align an $r \times c$ matrix for plain implementation. Practically, it is more efficient since we could reduce the comparison by skipping symbols that are already compared by recording distinct symbols for each row. This list is then forwarded to the mining process.

3.3 Pattern Mining

This pattern step is designed to handle set-typed data, where multiple values occur; thus, a naive approach is to discover repetitive patterns in the input. However, there can be many repetitive patterns discovered and a pattern can be embedded in another pattern, which makes the deduction of the template difficult. The good news is that we can neglect the effect of missing attributes (optional data) since they are handled in the previous step. Thus, we should focus on how repetitive patterns are merged to deduce the data structure. In this section, we detect every consecutive repetitive pattern (tandem repeat) and merge them (by deleting all occurrences except for the first one) from small length to large length. This is because the structured data defined here are nested and if we neglect the effect of optional, instances of a set-type data should occur consecutively according to the problem definition.

To detect a repetitive pattern, the longest pattern length is predicated by the function $compLvalue(List, t)$ (Line 1) in Fig. 10, which computes the possible pattern length, called L value, at each node (for extension t) in $List$ and returns the maximum L value for all nodes. For the tth extension, the possible pattern length for a node n at position p is the distance between p and the tth occurrence of n after p, or 0 otherwise. In other words, the tth extension deals with patterns that contain exactly

```
Procedure PatternMining(List, extent)
1. K = compLvalue(List, extend);
2. for (i = 1; i <= K; i++)
3.     st = 1;
4.     while((st = Next(List, i, st)) ≥ 0)
5.         newRep = 0;
6.         for (j = st + i; j + i − 1 ≤ |List|; j += i)
7.             if (!match(List[st..st + i − 1],
                       List[j..j + i − 1])) then break;
8.             newRep++;
9.         endfor
10.        if (newRep) then
11.            modifyList(List, st, newRep + 1, i);
13.            K = compLvalue(List, extend);
14.            st += i;
15.        else st++;
15.        endif
16.    endwhile
17. endfor
20. if (patternCanExtend(List)) then
21.    patternMining(List, extend + 1);
22. endif
```

Fig. 10. Pattern mining algorithm.

t occurrences of a node. Starting from the smallest length $i = 1$ (Line 2), the algorithm finds the start position of a pattern by the $Next(List, i, st)$ function (Line 4) that looks for the first node in $List$ that has L equal to i (i.e., the possible pattern length) beginning at st. If no such nodes exist, $Next$ returns a negative value which will terminate the while loop at line 4.

For each possible pattern starting at st with length i, we compare it with the next occurrence at $j = st + i$ by function $match$, which returns true if the two strings are the same. The algorithm continues to find more matches of the pattern (j += i) until either the first mismatch (Line 7) or the end of the list has encountered, i.e., $j + i - 1 \geq |List|$ (line 6). If a pattern is detected ($newRep > 0$), the algorithm then modifies the list ($modifyList$ at line 11) by deleting all occurrences of the pattern except for the first one, recomputes the possible pattern length for each node in the modified list (line 12), reinitializes the variables to be ready for a new repetitive pattern (line 5), and continues the comparisons for any further repetitive patterns in the list.

Note that a pattern may contain more than one occurrence of a symbol; so the function recursively (with extension increased by 1) tries to detect such patterns (line 21). The termination condition is when there is no more nodes with more than one occurrence or the list cannot be extended by the function $patternCanExtend$, which is verified by checking if the length of $List$ is greater than twice the length of the shortest repetitive pattern, i.e., $|List| < 2(lb)(extend + 1)$, where lb is the minimum L value in the current list. The complexity of the algorithm is quadratic ($O(n^2), n = |List|$).

As an example, we apply the frequent pattern mining algorithm on $List_1$ in Fig. 11 with $extend = 1$. The L values for the 11 nodes are 3, 1, 2, 2, 4, 2, 4, 2, 0, 0, and 0, respectively. The patterns have length at most 4 (=K). Note that, the value of K may be changed after each modification of the list. First, it looks for 1-combination repetitive patterns by starting at the 2nd node (n_2), which is the first node with L value 1. The algorithm starts at the 2nd (=st) node to compare every consecutive 1-combination of nodes, and the comparison will continue until reaching the first mismatch at 4th node (n_1). At this moment, the algorithm modifies the list by deleting the 3rd node (n_2) to get $List_2$. The new L values for the 10 nodes in $List_2$ in order are 2, 2, 2, 4, 2, 4, 2, 0, 0, and 0 (the value of K is still 4). The algorithm looks for another repetitive pattern of length 1 in $List_2$ starting from the 3rd node ($st + 1 = 3$), but finds no such nodes (the function $Next$ returns a value -1). This will end the while loop (Line 4) and search for 2-combination on $List_2$ from beginning (Lines 2 and 3). With L value equals 2 at the first node of $List_2$, it compares the 2-combination patterns 1-2, 3-4 of $List_2$ to detect a new repetitive pattern of length 2. The algorithm then deletes the second occurrence of the new detected pattern and outputs $List_3$ with L values 2, 4, 2, 4, 2, 0, 0, and 0. The process goes on until all i-combinations, $i \leq K$, have been tried. The algorithm then executes for the second time with extend=2 (Line 21). The new L values for $List_3$ will be 4, 0, 4, 0, 0, 0, 0, and 0. Again, starting by 1-combination comparisons until the 4-combination, the algorithm detects a repetitive pattern of length 4 by comparing the two 4-combination 1-4 and 5-8, and finally gets $List_4$ as a result. Finally, we shall add a virtual node for every pattern detected.

3.4 Optional Node Merging

After the mining step, we are able to detect optional nodes based on the occurrence vectors. The occurrence vector of a node c is defined as the vector $(b_1, b_2, \ldots, b_u)$, where b_i is 1 if c occurs in the ith occurrence, or 0 otherwise. If c is not part of a set type, u will be the number of input pages. If c is part of a set type, then u will be the summation of repeats in all

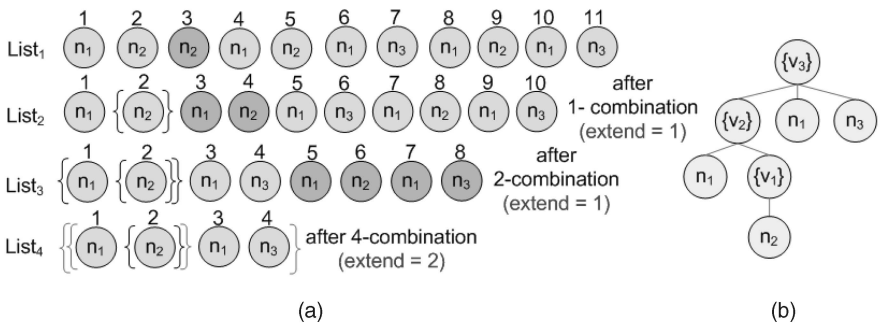

Fig. 11. (a) Pattern mining example and (b) virtual nodes added.

pages. For example, the childList "$abcdbcdbcde$" in Fig. 9 will become "$abcde$" after the mining step, where the three nodes b, c, and d will be considered as set-typed data with 6 repeats (3 from the first tree, 1 from the second, and 2 from the third tree of Fig. 9a). For the two nodes a and e, the occurrence vector is (1,1,1). The occurrence vectors of the three nodes b, c, and d are (1,1,1,1,1,1), (1,1,1,1,1,1), and (1,0,1,1,0,1), respectively. We shall detect node d as optional for it disappears in some occurrences of the pattern.

With the occurrence vector defined above, we then group optional nodes based on the following rules and add to the pattern tree one virtual node for the group.

- Rule 1. If a set of adjacent optional nodes $c_i, c_{i+1}, \ldots, c_k$ $(i < k)$ have the same occurrence vectors, we shall group them as optional.
- Rule 2. If a set of adjacent optional nodes $c_i, c_{i+1}, \ldots, c_k$ $(i < k)$ have complement occurrence vectors, we shall group them as disjunction.
- Rule 3. If an optional node c_i is a fixed node, we shall group it with the nearest nonfixed node (even if they have different occurrence vectors), i.e., group $c_i, c_{i+1}, \ldots, c_k$, where $c_i, c_{i+1}, \ldots, c_{k-1}$ are fixed, while c_k is not fixed (contains data).

Rule 3 is enforced to group fixed templates with the next following optional data such that every template is hooked with some other data. This rule is used to correct conditions due to misalignment as the running example in the next section. Note that a virtual node is added for each merged optional and disjunctive just like set-type data.

3.5 Examples

We first look at the fixed/variant pattern tree constructed from Fig. 1a. As discussed above, this pattern tree is initially started from the $<html>$ tag node as a root node. Then, children of the $<html>$ nodes of the input DOM trees are collected for alignment. As there is only one input page in Fig. 1a, the single child list is automatically aligned and pattern mining does not change the childList. The second call to MultipleTreeMerge for $<body>$ node is similar until the first $<table>$ tag node. Here, peer node recognition will compare the four $<tr>$ nodes and find two types of $<tr>$ nodes. The pattern mining step will then discover the repeat pattern $<tr_1, tr_2>$ from the (single) $childList = [tr_1, tr_2, tr_1, tr_2]$ (see Fig. 12b) and represent it by a virtual node $\{v_1\}$ that is inserted as a child for the $<table>$ node in the pattern tree.

For the two occurrences of node $<tr_1>$, the peer matrix contains two columns, where each contains a single child node $<td>$. Thus, the occurrence vector is expanded due to set-type data. As these two $<td_1>$ nodes are peer nodes, the matrix is thus aligned. Node $<td_1>$ is then inserted as a child node of $<tr_1>$. We show in Fig. 12c the peer matrix for $<td_2>$ node and its occurrence vector. All tags in the aligned childList will then be inserted under $<td_2>$. Fig. 12d shows two missing elements in one of the two occurrences for node $<span>$. As both $
$ and $<Text>$ nodes have the same occurrence vector (0,1), they are merged to form an optional node v_3, which is inserted under $<span>$ together with its previous node $Text$, which is marked as a variant data denoted by an asterisk node. The process goes on until the whole pattern tree in Fig. 12a is constructed, where three virtual nodes are added in addition to original HTML tag nodes: v_1 and v_3 are added because of repeat patterns and v_2 is added due to missing data.

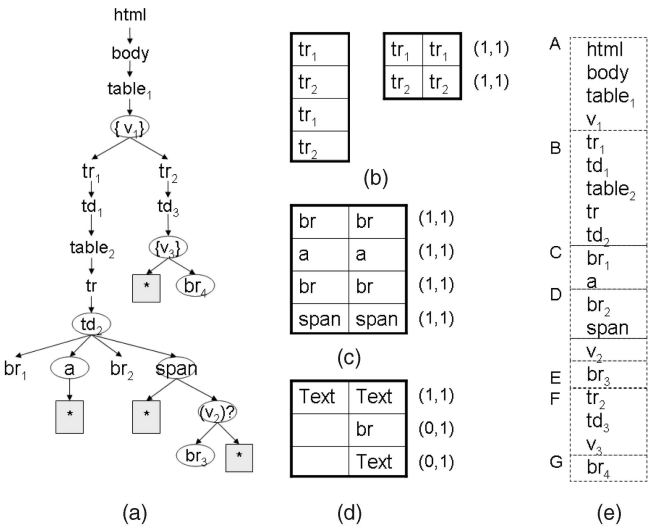

Fig. 12. The pattern tree constructed from Fig. 1a.

We shall use another example to illustrate how the fixed/variant pattern tree is built from a set of input DOM trees. Fig. 13 shows an example of three fictional DOM trees (Webpages). Every Webpage lists a set of products (in two columns), where each product corresponds to a data record. For each product, an image, a name, a price before discount, a current price, a discount percent, a set of features about the product, and delivery information for the product are presented. Starting from the root node $<html>$, we put child nodes of the $<html>$ nodes from the three DOM trees in three columns. Since the only one row in the peer matrix has the common peer nodes $<body>$, so the matrix is aligned and the node $<body>$ will be inserted as a child for the $<html>$ tag node in the pattern tree.

The merging procedure for $<body>$ is similar and the $<table>$ node will be inserted as a child node under $<body>$. Next, the system collects all children of the $<table>$ node to form the peer matrix M_1 as shown in the upper right corner of Fig. 14. All of the $<tr>$ nodes inside M_1 are matched nodes (take the same symbol using the 2-tree matching algorithm). So, after applying the alignment algorithm for the matrix M_1, the aligned result $childList$ will contain two $<tr>$ nodes. Passing this list into the mining algorithm, it detects a repetitive pattern of length 1 and then deletes the second occurrence of this pattern ($<tr>$). The $<tr>$ node will be the only child for the $<table>$ node and is marked as a repetitive pattern (denoted by virtual node v_1) of four occurrences. Thus, there will be four columns in the peer matrix when we process $<tr>$ node. There, we shall detect repeat pattern from the aligned childList=$[td, td]$. Hence, a virtual node v_2 is inserted under $<tr>$ with six occurrences and all the following nodes will be operated in a peer matrix with six columns.

The tricky part comes when we align the matrix M_2 for the $<div>$ node. The childList after both alignment and

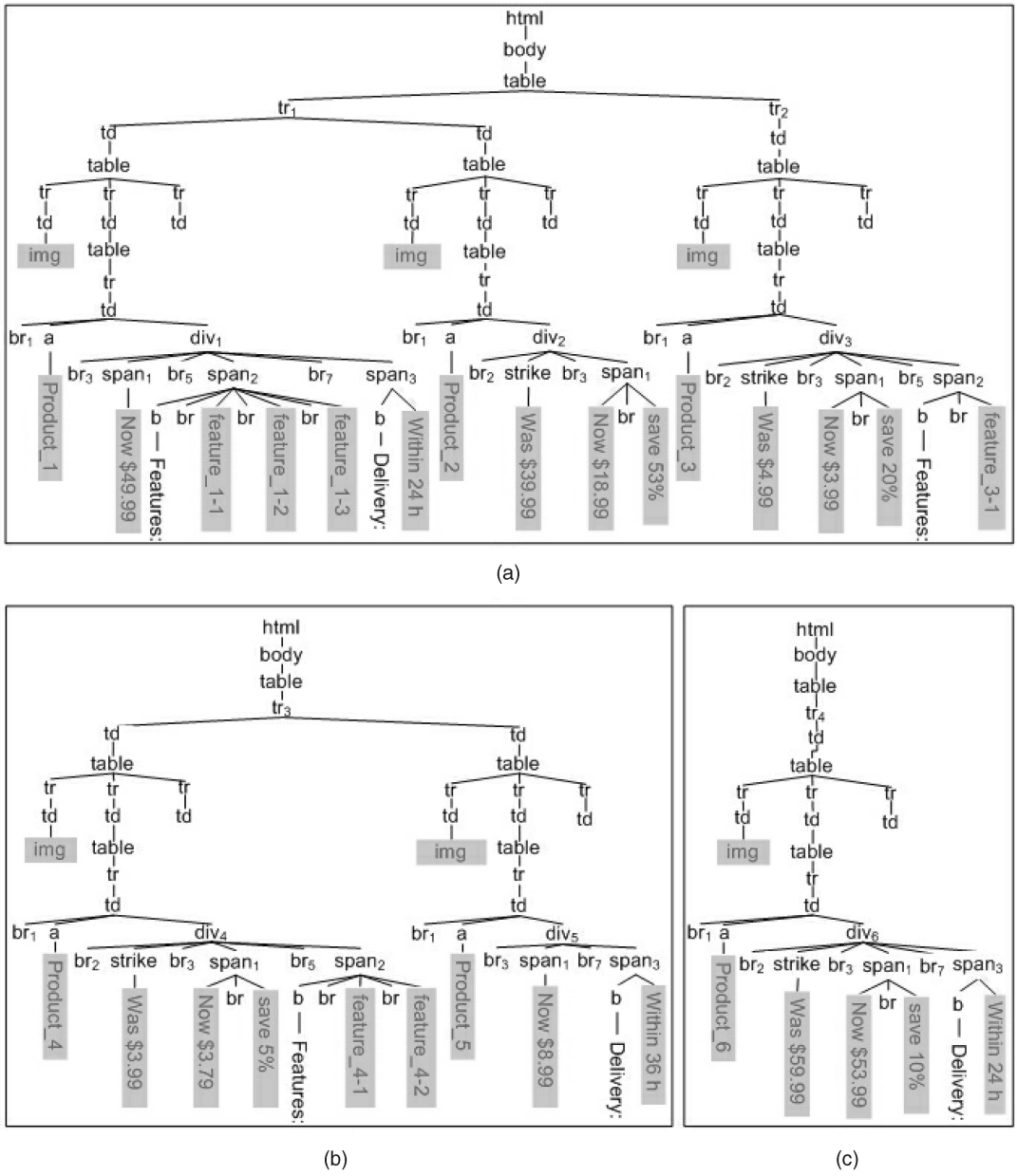

Fig. 13. Example of three DOM trees.

mining will be $[br_2, strike, br_3, span_1, br_5, span_2, br_7, span_3]$, where the three $<span>$ nodes are denoted by different symbols at peer node recognition. Of the eight nodes, only the two nodes $<br_2>$ and $<span_1>$ are not optional (with occurrence vector (1,1,1,1,1,1)) and all $
$ nodes are fixed as they are leaf nodes with the same content. The system should then group the two optional nodes $<strike>$ and $<br_3>$ based on Rule 1 (since they have the same occurrence vector (0,1,1,1,0,1)); and group $<br_5>$ and $<span_2>$ (similarly, $<br_7>$ and $<span_3>$) based on Rule 3.

The resulting pattern tree is similar to the original DOM tree with two differences. First, virtual nodes are added as parents of merged optional nodes (nodes marked by $()?$) and repetitive patterns (nodes marked by $\{\}$). Second, leaf Text nodes are classified as either fixed (e.g., the nodes "Features:") or variant (asterisk nodes). Beside these, every node in the input DOM trees has a corresponding node in the pattern tree. Now, the fixed/variant pattern tree carries all of the information that we need to detect the Website schema and identify the template of this schema.

4 Schema Detection

In this section, we describe the procedure for detecting schema and template based on the page generation model and problem definition. Detecting the structure of a Website includes two tasks: identifying the schema and defining the template for each type constructor of this schema. Since we already labeled basic type, set type, and optional type, the remaining task for schema detection is to recognize tuple type as well as the order of the set type and the optional data.

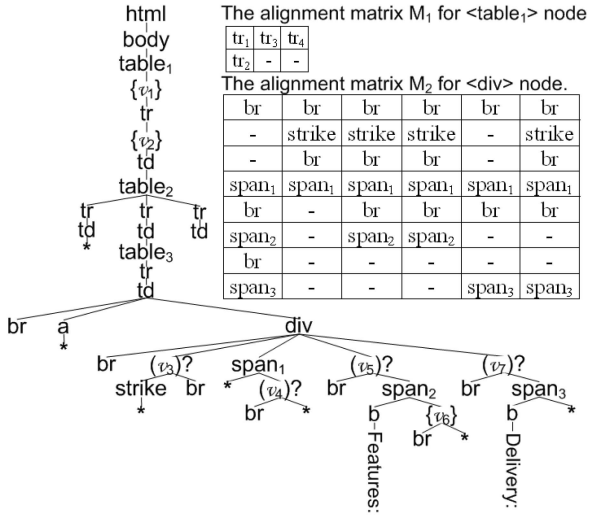

Fig. 14. The pattern tree constructed from Fig. 13.

```
Function MarkTheOrder(P)
1. if (P is a leaf node) then
2.     if (P is fixed) then return 0; else return 1;
3. endif
4. if (P has only one child C) then
5.     if (P is already marked as a type constructor) then
6.         Mark the type as 1-order;
7.     return MarkTheOrder(C);
8. endif
9. k= 0;
10. for each child C of P do
11.         k += MarkTheOrder(C);
12. endfor
13. if (k==0) then return 0;
14. if (P is a virtual node) then
15.         Mark P as k-order;
16. else
17.         Mark P as k-tuple;
18. endif
19. return 1;
```

Fig. 15. Identifying the orders of type constructors in the pattern tree.

As shown in Fig. 15, the system traverses the fixed-variant pattern tree P from the root downward and marks nodes as k-order (if the node is already marked as some data type) or k-tuple. For nodes with only one child and not marked as set or optional type, there is no need to mark it as 1-tuple (otherwise, there will be too many 1-tuples in the schema); thus, we simply traverse down the path to discover other type nodes. For nodes with more than one branch (child), we will mark them as k-order if k children have the function $MarkTheOrder(C)$ return true. The identified tuple nodes of the running example are marked by angle brackets <>. Then, the schema tree S can be obtained by excluding all of the tag nodes that have no types. For example, the schema for Fig. 12a is Fig. 1c and the schema for Fig. 14 is shown in Fig. 16.

Once the schema is identified, the template of each type can be discovered by concatenating nodes without types. The insertion positions can also be calculated with reference to the leaf node of the rightmost path of the template subtree. Formally, templates can be obtained by segmenting the pattern tree at reference nodes defined below:

Definition 4.1 (Reference nodes). *A node r is called a reference node if*

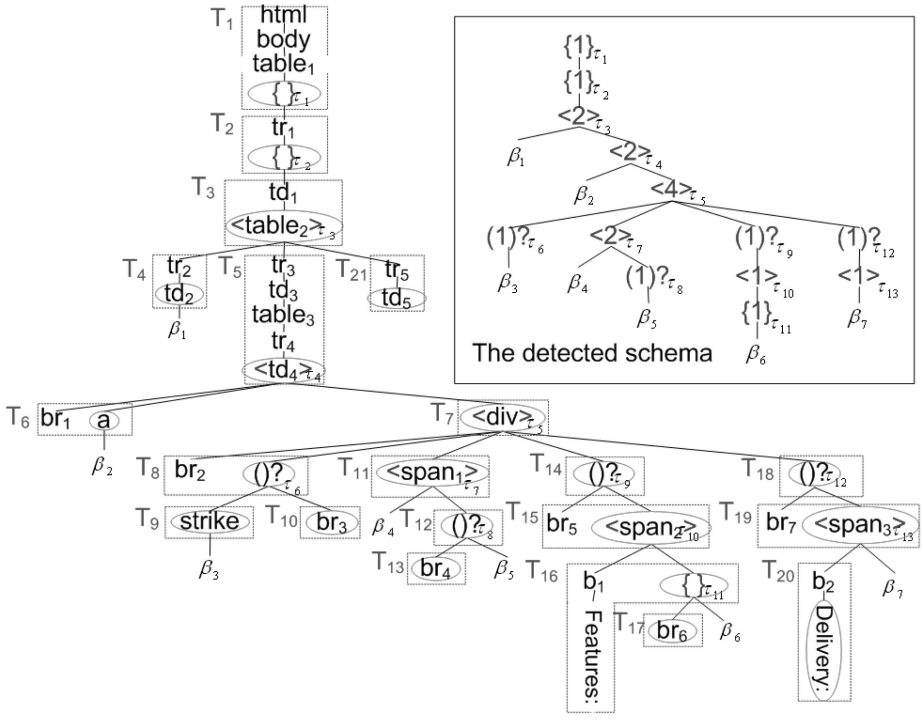

Fig. 16. Detecting the schema from Fig. 14.

- *r is a node of a type constructor;*
- *the next (right) node of r, in a preorder traversing of the pattern tree, is a basic type β; and*
- *r is a fixed leaf node on the rightmost path of a k-order data and is not of any type. We call r a rightmost reference node.*

For example, the reference nodes for the pattern tree in Fig. 12a are circled with blue ovals. Nodes $v_1, v_2,$ and v_3 are circled because they are type constructor as well as td_2 and $span$. Nodes a and br_3 are circled because their next nodes are a basic type. Finally, br_4 is circled because it is a rightmost reference node. Similarly, we can also marked the reference nodes for the pattern tree in Fig. 14 according to the three types defined: type constructor reference nodes include: $v_1, v_2, \ldots, v_7$ and $table_2, td_4, div, span_1, span_2, span_3$; the second type reference nodes include: $td_2, a, strike, br_4, br_6,$ "$Delivery$:"; and the rightmost reference nodes include br_3, td_5 as marked in Fig. 16.

With the definition of reference nodes, we can identify a set of templates by segmenting the preorder traversing of the pattern tree (skipping basic type nodes) at every reference node. For example, Fig. 12e shows the preorder traversal of the pattern tree, where the templates are segmented by reference nodes. Similarly, the rectangles (with dotted lines) in Fig. 16 show the 21 templates segmented by all reference nodes: T_1 is the first template starting from root node to the first reference node $\{\}_{\tau 1}$; template T_4 begins at tr_2 and ends at td_2 (the node before the β_1); template T_5 starts at node tr_3 and ends at the reference node td_4 (a 2-tuple). We say a template is under a node p if the first node of the template is a child of p. For example, the templates under $<div>$ include T_8, T_{11}, T_{14}, and T_{18}. Now, we can fill in the templates for each type as follows.

For any k-order type constructor $<\tau_1, \tau_2, \ldots, \tau_k>$ at node n, where every type τ_i is located at a node n_i $(i = 1, \ldots, k)$, then the template P will be the null template or the one containing its reference node if it is the first data type in the schema tree. If τ_i is a type constructor, then C_i will be the template that includes node n_i and the respective insertion position will be 0. If τ_i is of basic type, then C_i will be the template that is under n and includes the reference node of n_i or null if no such templates exist. If C_i is not null, the respective insertion position will be the distance of n_i to the rightmost path of C_i. Template C_{i+1} will be the one that has the rightmost reference node inside n or null otherwise.

For example, in Fig. 16, τ_3 (which is a 2-tuple of $<\beta_1, \tau_4>$) has child templates T_4, T_5, and T_{21}. The first two are related to the reference nodes of β_1 and τ_4, respectively. While the last child template T_{21} contains the rightmost reference node td_5. Thus, the templates for τ_3 can be written as $T(\tau_3) = (\phi, (T_4, T_5, T_{21}), (0, 0))$. As another example, τ_{11} with order 1 has child template T_{17} that relates to the reference node of β_6. Since β_6 is inserted to the virtual node of T_{17}, the inserted point is 1 node above the reference node. Thus, the templates for τ_{11} can be written as $T(\tau_{11}) = (\phi, (T_{17}, \phi), 1)$.

Other templates for the schema in Fig. 16 include

$$
\begin{aligned}
T(\tau_1) &= (T_1, (T_2, \phi), 0),\\
T(\tau_2) &= (\phi, (T_3, \phi), 0),\\
T(\tau_4) &= (\phi, (T_6, T_7, \phi), (0, 0)),\\
T(\tau_5) &= (\phi, (T_8, T_{11}, T_{14}, T_{18}, \phi), (0, 0, 0, 0)),\\
T(\tau_6) &= (\phi, (T_9, T_{10}), 0),\\
T(\tau_7) &= (\phi, (\phi, T_{12}, \phi), (0, 0)),\\
T(\tau_8) &= (\phi, (T_{13}, \phi), 1),\\
T(\tau_9) &= (\phi, (T_{15}, \phi), 0),\\
T(\tau_{10}) &= (\phi, (T_{16}, \phi), 2),\\
T(\tau_{12}) &= (\phi, (T_{19}, \phi), 0), \quad \text{and}\\
T(\tau_{13}) &= (\phi, (T_{20}, \phi), 2).
\end{aligned}
$$

5 Experiments

We conducted two experiments to evaluate the schema resulted by our system and compare FiVaTech with other recent approaches. The first experiment is conducted to evaluate the schema resulted by our system, and at the same time, to compare FiVaTech with EXALG [1]; the page-level data extraction approach that also detects the schema of a Website. The second experiment is conducted to evaluate the extraction of data records or interchangeably search result records (SRRs), and compare FiVaTech with the three state-of-the-art approaches: DEPTA [14], ViPER [10], and MSE [16]. Unless otherwise specified, we usually take two Web pages as input.

To conduct the second experiment, FiVaTech has an extra task of recognizing data sections in a Website. A data section is the area in the Webpage that includes multiple instances of a data record (SRRs). FiVaTech recognizes the set of nodes n_{SRRs} in the schema tree that corresponds to different data sections by identifying the outermost set type nodes, i.e., the path from the node n_{SRR} to the root of the schema tree has no other nodes of set type. A special case is when the identified node n_{SRR} in the schema tree has only one child node of another set type (as the example in Fig. 16 of the running example), this means that data records of this section are presented in more than one column of a Webpage, while FiVaTech still catches the data.

Given a set of Webpages of a Website as input, FiVaTech outputs three types of files for the Website. The first type (a text file) presents the schema (data values) of the Website in an XML-like structure. We use these XML files in the first experiment to compare FiVaTech with EXALG. The second type of file (an html file) presents the extracted SRRs (of each dynamic section) of the test and the training Webpages of the Website. A simple extractor program that uses both the identified n_{SRR} nodes in the schema tree and the templates associated with these nodes is implemented to output these HTML files. We use these files in the second experiment to evaluate FiVaTech as an SRRs extractor and compare the system with the three record-level approaches DEPTA, ViPER, and MSE. Finally, the third type of file (an Excel file) contains the data items of the set of all attributes of a basic type; every column in the file has the set of all instances of a basic type that are collected from the test and the training Webpages. We use these Excel files in the

TABLE 1
Performance Comparison between FiVaTech and EXALG

Dataset: 9 Web sites on EXALG home page.																
site	N	Manual			EXALG						FiVaTech					
									Incorr.						Incorr.	
		A_m	O_m	{}	A_e	O_e	{}	c	i	n	A_e	O_e	{}	c	i	n
Amazon (Cars)	21	13	0	5	15	0	5	11	4	2	8	1	4	8	0	0
Amazon (Pop)	19	5	0	1	5	0	1	5	0	0	5	0	1	5	0	0
MLB	10	7	0	4	7	0	4	7	0	0	6	0	1	6	0	1
RPM	20	6	1	3	6	1	3	6	0	0	5	0	3	5	0	1
UEFA (Teams)	20	9	0	0	9	0	0	9	0	0	9	0	0	9	0	0
UEFA (Play)	20	2	0	1	4	2	1	2	2	0	2	0	0	2	0	0
E-Bay	50	22	3	0	28	2	0	18	10	4	20	5	0	19	1	3
Netflix	50	29	9	6	37	2	1	25	12	4	34	12	7	29	5	0
US Open	32	35	13	10	42	4	10	33	9	2	33	14	11	33	0	2
Total	242	128	26	25	153	11	23	116	37	12	122	32	20	116	6	7
Recall					**90.6%**						**90.6%**					
Precision					**75.8%**						**95.1%**					

second experiment to compare the alignment results of FiVaTech with the alignment results of DEPTA.

5.1 FiVaTech as a Schema Extractor

Given the detected schema S_e of a Website and the manually constructed schema S_m for this site, EXALG evaluates the resulted schema S_e by comparing data extracted by leaf attributes A_e of this schema from collections of Webpages of this site. However, this is not enough for two reasons. First, many Web applications (e.g., information integration systems) need such schemas as input, so it is very important to evaluate the whole schema S_e. Second, for Web data extraction, the values of an attribute may be extracted correctly (partially correct as defined by EXALG [1]) but its schema is incorrect, and vice versa. For example, the first instance of a repetitive data record is often excluded from the set but is recognized as a tuple. Thus, all instances of the data record are extracted although the schema is wrong (the first instance is identified as of a tuple type while the remaining are instances of a set type). Meanwhile, many disjunctive types and empty types (corresponding to no data in the schema S_m) are extracted by EXALG but are considered correct because they did not extract wrong results.

Table 1 shows the evaluation of the schema S_e resulted by our system and the comparison with the schema resulted by EXALG. We use the nine sites that are available at http://infolab.stanford.edu/arvind/extract/. We do not change the manual schema S_m that has been provided by EXALG. The first two columns show the nine sites used for the experiment and the number of pages N in each site, respectively. Columns 3-5 (Manual) show the details of the manual schema S_m; the total number of attributes (basic types) A_m, the number of attributes that are optional O_m, and the number of attributes that are part of the set type.

Columns 6-8 (12-14) show the details of the schema resulted by EXALG (FiVaTech). Note that we consider each disjunctive attribute detected by EXALG as a basic-type attribute but ignore empty-type attributes. Columns 9 and 15 (denoted by "c") show the number of attributes in the deduced schema (for both EXALG and FiVaTech, respectively) that correspond to an attribute in the manual schema and its extracted values from the N Webpage are correct or partially correct. If two attributes from A_e correspond to one attribute in the manual schema, we consider one of the two as correct and the other is incorrect. Columns 10 and 16 (denoted by "i") show the number of incorrect attributes (i.e., those from A_e that have no corresponding ones in A_m). Columns 11 and 17 (denoted by "n") show the number of attributes that are not extracted (i.e., those from A_m which have no corresponding ones in A_e).

Of the 128 manually labeled attributes, 116 are correctly extracted by both EXALG and FiVaTech. However, EXALG produced a total of 153 basic types and FiVaTech produced 122 basic types. Thus, the precision of FiVaTech is much higher than EXALG. One of the reasons why EXALG produces so many basic types is because the first record of a set type is usually recognized as part of a tuple. On the other hand, FiVaTech usually produces less number of attributes since we do not analyze the contents inside text nodes in this version.

5.2 FiVaTech as an SRRs Extractor

Of the popular approaches that extract SRRs from one or more data sections of a Webpage, the main problem is to detect record boundaries. The minor problem is to align data inside these data records. However, most approaches concern with the main problem except for DEPTA, which applies partial tree alignment for the second problem. Therefore, we compare FiVaTech with DEPTA in both steps and focus on the first step when comparing with ViPER and MSE.

In the second experiment (a comparison with DEPTA), we configure FiVaTech to detect the schema from a single Webpage, although this will give an incorrect schema outside the span of sections of multiple data records (n_{SRRs}), but we are only concerning with data sections and the SRRs inside each section. We got the system demo from the author and ran DEPTA on the manually labeled Testbed for Information Extraction from Deep Web TBDW [12] Version 1.02 available at http://daisen.cc.kyushu-u.ac.jp/TBDW/. Unfortunately, DEPTA gave a result only for 11 Websites and could not produce any output for the remaining 40 sites. So, we conducted the following experiment for these 11 Websites. For SRRs extraction (columns 2 and 3 in Table 2), we just used the Webpages that have multiple data records. DEPTA gave a good result for six Websites and extracted incorrect SRRs for

TABLE 2
Performance on 11 Websites from Testbed Version 1.02

	SRRs Extraction		Alignment	
	#Actual SRRs: 419		#Actual attributes: 92	
	DEPTA	FiVaTech	DEPTA	FiVaTech
#Extracted	248	409	93	91
#Correct	226	401	45	82
Recall	53.9%	95.7%	48.9%	89.1%
Precision	91.1%	98.0%	48.4%	90.1%

TABLE 3
Performance Comparison between ViPER and MSE

Dataset	TBDW [12]		MSE [16]	
#Actual SRRs:	693		1242	
System	ViPER	FiVaTech	MSE	FiVaTech
#Extracted	686	690	1281	1260
#Correct	676	672	1193	1186
Recall	97.6%	97.0%	96.1%	95.5%
Precision	98.5%	97.4%	93.1%	94.1%

four Websites. For the last Website (the site numbered 13 in Testbed), DEPTA merged every two correct data records and extracted them as a single data record. We considered half of the data records are not extracted for this last site.

For the second step of the comparison with DEPTA (columns 3 and 4 in Table 2), by the help of the manually labeled data in Testbed, we counted the number of attributes (including optional attributes) inside data records of each data section (92 attributes). An attribute is considered extracted correctly if 60 percent of its instances (data items) are extracted correctly and aligned together in one column. For example, the Amazon (Books) Website has three data sections, which include three, 10, and four different attributes, respectively. DEPTA extracted data items of three attributes from the first section, five attributes from the second, and 0 attribute from the last section; however, the total number of extracted attributes (number of columns in an Excel output file) is 24. Thus, the recall and precision are below 50 percent for DEPTA, while FiVaTech has a nearly 90 percent performance for both precision and recall. Our explanation for the poor results of DEPTA is due to the shortcomings that we have been discussed in details in Section 6.3.

The last experiment compares FiVaTech with the two visual-based data extraction systems, ViPER and MSE. The first one (ViPER) is concerning with extracting SRRs from a single (major) data section, while the second one is a multiple section extraction system. We use the 51 Websites of the Testbed referred above to compare FiVaTech with ViPER, and the 38 multiple sections Websites used in MSE to compare our system with MSE. Actually, extracting of SRRs from Webpages that have one or more data sections is a similar task. The results in Table 3 show that all of the current data extraction systems perform well in detecting data record boundaries inside one or more data sections of a Webpage. The closeness of the results between FiVaTech and the two visual-based Web data extraction systems ViPER and MSE gives an indication that until this moment visual informations do not provide the required improvement that researchers expect. This also appeared in the experimental results of ViNTs [15]; the visual-based Web data extraction with and without utilizing visual features. FiVaTech fails to extract SRRs when the peer node recognition algorithm incorrectly measures the similarities among SRRs due to the very different structure among them. Practically, this occurred very infrequently in the entire test page (e.g., site numbered 27 in the Testbed). Therefore, now, we can claim that SRRs extraction is not a key challenge for the problem of Web data extraction.

On a Pentium 4 (3.39 GHz) PC, the response time is about 5-50 seconds, where the majority of time is consumed at the peer node recognition step (line 9 in Fig. 5). Therefore, the running time of FiVaTech has a wide range (5-50 seconds) and leaves room for improvement.

6 Comparison with Related Work

Web data extraction has been a hot topic for recent 10 years. A lot of approaches have been developed with different task domain, automation degree, and techniques [3], [7]. Due to the space limitation, we only compare our approach with two related works, DEPTA and EXALG. We compare FiVaTech with the first one (DEPTA) because it includes the same two components of frequent pattern mining and data alignment, while we compare FiVaTech with EXALG because it has the same task of schema detection.

Although both FiVaTech and DEPTA include the same two main tasks of data alignment and frequent pattern mining, the two approaches are completely different. First, DEPTA mines frequent repetitive patterns before the alignment step, so the mining process may not be accurate because of missing data in the repetitive patterns. Also, this order contradicts the assumption that repetitive patterns are of fixed length. So, FiVaTech applies the alignment step before the mining to make this assumption correct and get accurate frequent pattern mining results. Second, DEPTA [14] only relies on HTML tags (tag trees) for both data alignment and frequent mining tasks. Actually, a tag tree representation of a Webpage carries only structural information. Important text information is missed in this representation. The new version of DEPTA tries to handle this problem by using a DOM tree instead of the HTML tag tree and by applying a naive text similarity measure. Further, the partial tree alignment results rely on the order in which trees are selected to be compared with the seed tree. In the frequent mining algorithm, using of a tag tree prohibits DEPTA from differentiating between repetitive patterns that contain noisy data and those that contain data to be extracted. So, the algorithm detects repetitive patterns regardless of their contents. Third, DEPTA cannot extract data from singleton Webpages (pages that have only one data record) because its input is a single Webpage, while FiVaTech can handle both singleton and multiple data records Webpages because it can take more than one page as input.

Regardless of the granularity of the used tokens, both FiVaTech and EXALG recognize template and data tokens in the input Webpages. EXALG assumes that HTML tags as part of the data and proposes a general technique to identify tokens that are part of the data and tokens that are part of the template by using the occurrence vector for each token and by differentiating the role of the tokens according to its DOM tree path. Although this assumption is true,

differentiating HTML tag tokens is a big challenge and causes many problems. Also, EXALG assumes that a pair of two valid equivalence classes is nested, although this is not necessarily true. Two data records may be intertwining in terms of their HTML codes.

Finally, a more compact schema can be conducted by compressing continuous tuples, removing continuous sets and any 1-tuples. A list of types $\tau_1, \tau_2, \ldots, \tau_n$ is continuous if τ_i is a child of τ_{i-1} (for $n > i > 1$). If $\tau_1, \tau_2, \ldots, \tau_n$ are of tuples of order $k_1, k_2, ..., k_n$, respectively, then the new compressed tuple is of order $k_1 + k_2 + ... + k_n - n + 1$. For the above example, we can compress $\tau_3, \tau_4, \tau_5, \tau_7$ to get a 7-tuple ($=2+2+4+2+2-4+1$) and the new schema

$$S = \{\{\beta_1, \beta_2, (\beta_3)?_{\tau 6}, \beta_4, (\beta_5)?_{\tau 8}, \\ (< \{\beta_6\}_{\tau 11} >_{\tau 10})?_{\tau 9}, (< \beta_7 >_{\tau 13})?_{\tau 12}\}_w\}_{\tau 1},$$

where w is a 7-set.

7 Conclusions

In this paper, we proposed a new Web data extraction approach, called FiVaTech to the problem of page-level data extraction. We formulate the page generation model using an encoding scheme based on tree templates and schema, which organize data by their parent node in the DOM trees. FiVaTech contains two phases: phase I is merging input DOM trees to construct the fixed/variant pattern tree and phase II is schema and template detection based on the pattern tree.

According to our page generation model, data instances of the same type have the same path in the DOM trees of the input pages. Thus, the alignment of input DOM trees can be implemented by string alignment at each internal node. We design a new algorithm for multiple string alignment, which takes optional- and set-type data into consideration. The advantage is that nodes with the same tag name can be better differentiated by the subtree they contain. Meanwhile, the result of alignment makes pattern mining more accurate. With the constructed fixed/variant pattern tree, we can easily deduce the schema and template for the input Webpages.

Although many unsupervised approaches have been proposed for Web data extraction (see [3], [7] for a survey), very few works (RoadRunner and EXALG) solve this problem at a page level. The proposed page generation model with tree-based template matches the nature of the Webpages. Meanwhile, the merged pattern tree gives very good result for schema and template deduction. For the sake of efficiency, we only use two or three pages as input. Whether more input pages can improve the performance requires further study. Also, extending the analysis to string contents inside text nodes and matching schema that is produced due to variant templates are two interesting tasks that we will consider next.

Acknowledgments

This project is sponsored by the National Science Council, Taiwan, under grants NSC96-2221-E-008-091-MY2 and NSC97-2627-E-008-001. The work was done during Dr. Kayed's study at the NCU, Taiwan.

References

[1] A. Arasu and H. Garcia-Molina, "Extracting Structured Data from Web Pages," *Proc. ACM SIGMOD*, pp. 337-348, 2003.

[2] C.-H. Chang and S.-C. Lui, "IEPAD: Information Extraction Based on Pattern Discovery," *Proc. Int'l Conf. World Wide Web (WWW-10)*, pp. 223-231, 2001.

[3] C.-H. Chang, M. Kayed, M.R. Girgis, and K.A. Shaalan, "Survey of Web Information Extraction Systems," *IEEE Trans. Knowledge and Data Eng.*, vol. 18, no. 10, pp. 1411-1428, Oct. 2006.

[4] V. Crescenzi, G. Mecca, and P. Merialdo, "Knowledge and Data Engineerings," *Proc. Int'l Conf. Very Large Databases (VLDB)*, pp. 109-118, 2001.

[5] C.-N. Hsu and M. Dung, "Generating Finite-State Transducers for Semi-Structured Data Extraction from the Web," *J. Information Systems*, vol. 23, no. 8, pp. 521-538, 1998.

[6] N. Kushmerick, D. Weld, and R. Doorenbos, "Wrapper Induction for Information Extraction," *Proc. 15th Int'l Joint Conf. Artificial Intelligence (IJCAI)*, pp. 729-735, 1997.

[7] A.H.F. Laender, B.A. Ribeiro-Neto, A.S. Silva, and J.S. Teixeira, "A Brief Survey of Web Data Extraction Tools," *SIGMOD Record*, vol. 31, no. 2, pp. 84-93, 2002.

[8] B. Lib, R. Grossman, and Y. Zhai, "Mining Data Records in Web pages," *Proc. Int'l Conf. Knowledge Discovery and Data Mining (KDD)*, pp. 601-606, 2003.

[9] I. Muslea, S. Minton, and C. Knoblock, "A Hierarchical Approach to Wrapper Induction," *Proc. Third Int'l Conf. Autonomous Agents (AA '99)*, 1999.

[10] K. Simon and G. Lausen, "ViPER: Augmenting Automatic Information Extraction with Visual Perceptions," *Proc. Int'l Conf. Information and Knowledge Management (CIKM)*, 2005.

[11] J. Wang and F.H. Lochovsky, "Data Extraction and Label Assignment for Web Databases," *Proc. Int'l Conf. World Wide Web (WWW-12)*, pp. 187-196, 2003.

[12] Y. Yamada, N. Craswell, T. Nakatoh, and S. Hirokawa, "Testbed for Information Extraction from Deep Web," *Proc. Int'l Conf. World Wide Web (WWW-13)*, pp. 346-347, 2004.

[13] W. Yang, "Identifying Syntactic Differences between Two Programs," *Software—Practice and Experience*, vol. 21, no. 7, pp. 739-755, 1991.

[14] Y. Zhai and B. Liu, "Web Data Extraction Based on Partial Tree Alignment," *Proc. Int'l Conf. World Wide Web (WWW-14)*, pp. 76-85, 2005.

[15] H. Zhao, W. Meng, Z. Wu, V. Raghavan, and C. Yu, "Fully Automatic Wrapper Generation for Search Engines," *Proc. Int'l Conf. World Wide Web (WWW)*, 2005.

[16] H. Zhao, W. Meng, Z. Wu, V. Raghavan, and C. Yu, "Automatic Extraction of Dynamic Record Sections from Search Engine Result Pages," *Proc. Int'l Conf. Very Large Databases (VLDB)*, pp. 989-1000, 2006.

Mohammed Kayed received the MSc degree from Minia University, Egypt, in 2002, and the PhD degree from Beni-Suef University, Egypt, in 2007. He is currently an assistant professor at Beni-Suef University. His research interests include Web information integration, Web mining, and information retrieval. He was a member of the Database Lab in the Department of Computer Science and Information Engineering at the National Central University, Taiwan.

Chia-Hui Chang received the BS and PhD degrees from the Department of Computer Science and Information Engineering at National Taiwan University, Taiwan, in 1993 and 1999, respectively. She is currently an associate professor in the Department of Computer Science and Information Engineering at National Central University in Taiwan. Her research interests include Web information extraction, knowledge discovery from databases, machine learning, and data mining. She is a member of the IEEE.