FiVaTech: Page-Level Web Data Extraction from Template Pages

Mohammed Kayed
Department of Computer Science and Information Engineering, National Central University, Taiwan
kayed@db.csie.ncu.edu.tw

Chia-Hui Chang
Department of Computer Science and Information Engineering, National Central University, Taiwan
chia@csie.ncu.edu.tw

Khaled Shaalan
Institute of Informatics, the British University in Dubai, United Arab Emirates
khaled.shaalan@buid.ac.ae

Moheb Ramzy Girgis
Department of Computer Science, Minia University, El-Minia, Egypt
mrgirgis@mailer.eun.eg

Abstract

In this paper, we proposed a new approach, called FiVaTech for the problem of Web data extraction. FiVaTech is a page-level data extraction system which deduces the data schema and templates for the input pages generated from a CGI program. FiVaTech uses tree templates to model the generation of dynamic Web pages. FiVaTech can deduce the schema and templates for each individual Deep Web site, which contains either singleton or multiple data records in one Web page. FiVaTech applies tree matching, tree alignment, and mining techniques to achieve the challenging task. The experiments show an encouraging result for the test pages used in many state-of-the-art Web data extraction works.

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 Web pages for searchable Web sites 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 site is that such pages share the same template since they are generated with a predefined template by plugging data values. The extraction targets of these pages are almost equal to the data values embedded during page generation. Thus, there is no need to annotate the Web pages for extraction targets as in non-template page information extraction (e.g. Softmealy [3]) and the key to automatic extraction depends on whether we can deduce the template automatically. Finding that template requires multiple pages (e.g. EXALG [1]) or a single page containing multiple records as input (e.g. DEPTA [9]).

In this paper, we focus on page-level extraction tasks and propose a new approach, called FiVaTech, to automatically detect the schema of a Web site. 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 detail algorithm of FiVaTech. Section 4 gives the detail of template and schema deduction. Section 5 describes our experiments. Finally, section 6 concludes our work.

2. Problem formulation

In this section, we formulate the model for page creation which describes how data is embedded using a template. As we know, a Web page is created by embedding a data instance x (taken from the database) into a predefined *template*. Usually a CGI program executes the encoding function which combines a data instance with the template to form the Web page, where all data instances of the database conform to a common schema which can be defined as follows.

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 $\Phi_1, \Phi_2, \ldots, \Phi_n$ are types, then their ordered list $\Phi=<\Phi_1, \Phi_2, \ldots, \Phi_n>$ is also a type. We say the type Φ is constructed from the types $\Phi_1, \Phi_2, \ldots, \Phi_n$ using a type constructor of *order n*. An instance of the *n*-type is of the form $<x_1, x_2, \ldots, x_n>$ where $x_1, x_2, \ldots, x_n$ are instances of types $\Phi_1, \Phi_2, \ldots, \Phi_n$, respectively. A type Φ is called
   a. a *tuple*, denoted by $<\Phi>$, if the cardinality is 1 for every instantiation.
   b. an *optional*, denoted by $(\Phi)?$, if the cardinality is either 0 or 1 for every instantiation.
   c. a *set*, denoted by $\{\Phi\}$, if the cardinality is greater than 1 for some instantiation.
3. We say a *k*-tuple $\Phi=<\Phi_1, \Phi_2, ..., \Phi_k>$ is a disjunction of $\Phi_1, \Phi_2, \ldots, \Phi_k$ if the cardinality sum of Φ_1 to Φ_k equal to 1 for every instantiation of Φ.

Example 2.1: Figure 1(a) shows an example schema that contains 4 basic types (β), an optional type (τ_4), two set types (τ_1 and τ_5), and two tuple types (τ_2 and τ_3). A data instance of this schema is shown in Figure 1(c). The data instance contains a list of products (an instance of τ_1) where each product is described by its name, a price, a discount percent, and a list of features.

Instead of dealing with a Web page as a sequence of strings as in EXALG, we consider a page as a DOM tree since both data schema and Web pages themselves are tree-like structures. Therefore, our contribution is to consider the template T as tree structures. In such a tree generation model, we should consider the insertion positions since there is more than one point to append a subtree to the right most path of an existing tree. Let T_1 and T_2 be two trees constructed from some template and data. We define the operation $T_1 \oplus_i T_2$ as a new tree by appending T_2 to the i^{th} node from the leaf node on the right most path of T_1.

Definition 2.2: 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 follows.

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 n, then the template includes a parent template, $n+1$ child templates and n insertion positions: $T(\tau)=[P, (C_1, \ldots, C_{n+1}), (i_1, \ldots, i_n)]$.
   a. For single instance x of the form $(x_1, \ldots, x_n)_\tau$, $\lambda(T, x)$ is the tree produced by concatenating the $n+1$ ordered subtrees, $C_1 \oplus_{i1} \lambda(T,x_1)$, $C_2 \oplus_{i2} \lambda(T,x_2)$, …, $C_n \oplus_{in} \lambda(T,x_n)$, and C_{n+1} at the leaf on the right most path of template P.
   b. For multiple instances $e_1, e_2, ..., e_m$ where each e_i is an instance of type τ, the encoding $\lambda(T, \{ e_1, e_2, ..., e_m\})$ is the tree by inserting the m subtrees $\lambda(T, e_1)$, $\lambda(T, e_2)$, ..., $\lambda(T, e_m)$ as siblings at the leaf node on the right most path of P, *where each subtree* $\lambda(T, e_i)$ is produced by encoding each e_i using the child template and insertion positions of $T(\tau)$ with a null parent template.

Example 2.2: Suppose we have the template in Figure 1(b) for the schema in Figure 1(a) as defined above, the resulting page will look like that in Figure 1(d).

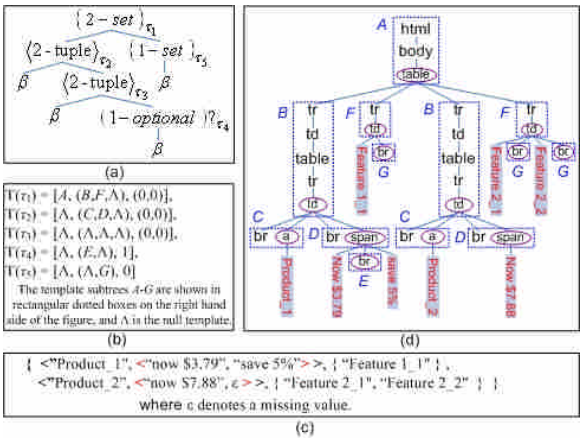

Figure 1: An example of a schema (a), its template (b), a data instance (c), and the resulted page (d).

As we can see, basic type data always reside at the leaf nodes of the generated trees. If basic type data can be identified, tuple constructors can be recognized accordingly. Thus, we shall focus on identifying "variant" leaf nodes which correspond to basic data types. Meanwhile, we should also recognize text nodes which are part of the template. However, we will not go into details inside text nodes (such as separate words inside the "now $3.79" text nodes) in this paper.

Definition 2.3: (**Problem Formulation**) Given a set of n DOM trees, $DOM_i = \lambda(T, x_i)$ $(1 \le i \le n)$, created from some unknown template T and values $\{x_1, \ldots, x_n\}$, deduce the template and values, from the set of DOM trees alone. We call this problem a *page-level* information extraction. If one single page (n=1) which contains tuple constructors is given as input, 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. The FiVaTech approach

Given a set of training pages from a Web site, we use DOM trees of the Web pages as input to detect the schema of this site. We try to merge all DOM trees at the same time into a single tree called a *fixed/variant*

pattern tree. From this pattern tree, we can recognize variant leaf nodes for basic-typed data and mine repetitive nodes for set-typed data. The resulting *pattern tree* is then used to detect the template and the schema of the Web site. The key challenge here is how to merge multiple trees at the same time. Our solution is to break down the multiple trees merging problem from a tree level to a string level and design a new algorithm for multiple string alignment that considers both missing data and multiple-value data.

Assume all input DOM trees have the same root node, the system starts by taking R=<html> as the root node for the pattern tree and tries to expand the pattern tree from the root downward in a depth-first fashion. At each internal node *n*, the system collects all first-level child nodes of the input trees (subtrees matched with the subtree from *n*) in a peer matrix M where each column keeps the child nodes for every tree. Every node in the peer matrix actually denotes a subtree. The system then enters four important steps for peer nodes recognition, peer matrix alignment, pattern mining, and optional nodes merging:

- In the peer node recognition step, two nodes with the same tag name are compared to check if they are peer subtrees. All peer subtrees will be denoted with the same symbol.
- In the peer matrix alignment step, the system tries to align nodes (symbols) in the peer matrix to get a list of aligned nodes *childList*. Additionaly, this step will recognize variant leaf nodes which correspond to basic-typed data.
- In the pattern mining step, the system takes the aligned *childList* as input to detect every repetitive pattern in this list. For each pattern, all occurrences of this pattern except for the first one are deleted and the pattern is marked as set-typed. The result of this mining step is a modified list of nodes without any repetitive patterns.
- In the last step, the system shall recognize optional nodes if a node disappears in some columns of the matrix and group nodes according to their occurrence vector in the matrix.

Finally, the system inserts the nodes in the modified *childList* as children for the node *n*. For non-leaf child node *c*, the algorithm recursively calls the tree merging algorithm with the peer subtrees of *c* to build the pattern tree. The next four subsections will discuss in details these four steps.

3.1. Peer node recognition

We can compare whether two subtrees (with the same root tags) are similar based on tree edit distance [5, 6]. We use the algorithm proposed by Yang [8] to calculate the maximum matching of two trees through dynamic programming and modify it to consider node replacement on the leaves instead of the roots of the two trees. Meanwhile, the matching is normalized from 0 to 1 based on the consideration of set type data. However, due to space limitation, we didn't present the detailed algorithm here.

3.2. Peer matrix alignment

As mentioned above, every node in the peer matrix M actually represents a subtree. Thus, two nodes with the same tag are denoted by the same symbol if their matching score is greater than a threshold δ. For leaf nodes, two text (<img>) nodes take the same symbol when they have the same text (SRC attribute, respectively) values (otherwise, they take different symbols). The alignment algorithm (shown in Figure 2) tries to align nodes in the matrix M, row by row, to convert M into an *aligned peer matrix*, where each row has (except for empty columns) either the same symbol for every column or leaf nodes of different symbols, which will be marked as basic-typed. From the aligned matrix M, we get a list of nodes where each node corresponds to a row in the aligned peer matrix.

```
Algorithm: peerMatrixAlignment(M)
1.  row = 1;
2.  shiftLength = 0;
3.  while (M is not an aligned peer matrix)
4.    while (! alignedRow(row, M))
5.      shiftColumn = getShiftedColumn(row, shiftLength, M);
6.      makeShift(row, shiftColumn, shiftLength, M);
7.    endwhile;
8.    row++;
9.  endwhile;
10. childList = alignmentResult(M);
```

Figure 2: The alignment algorithm.

At each row, the function *alignedRow* checks if the row is aligned or not (line 4). If it is aligned (either when the row nodes have the same symbol or when they are leaf nodes of different symbols and each one of these symbols appears only in its residing column; these nodes are identified as variant), the algorithm will go to the next row (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. Now, we shall discuss the function *getShiftedColumn* in details.

The selection of a node n_{rc} located at column *c* from current row *r* to be shifted depends on two values: $span(n_{rc})$ and $checkSpan_r(n_{rc})$. The first value is defined as the maximum number of different nodes (without repetition) between any two consecutive occurrences of n_{rc} in each column *c* plus one. In other words, this value represents the maximum possible

cycle length of the node. If n_{rc} occurs at most once in each column c, then we consider it as a free node and its span will be 0. Meanwhile, the value $checkSpan_r$ of a node n_{rc} at row r depends on whether there exists a node with the same symbol at row r_{up} and column c′, such that $r_{up} < r$, i.e. $M[r][c]=M[r_{up}][c']=n_{rc}$, then

$$checkSpan_r(n_{rc}) = \begin{cases} 1, & \text{if } (r - r_{up}) > span(n_{rc}) \\ 0, & \text{if } (r - r_{up}) = span(n_{rc}) \\ -1, & \text{if } (r - r_{up}) < span(n_{rc}) \end{cases};$$

otherwise, $checkSpan_r(n_{rc})$ will equal to 1.

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

R1. Select, from left to right, a column c where the node n_{rc} (=M[r][c]) has a $checkSpan_r(n_{rc})$ value equal to -1 to be shifted to the next row of r; i.e., *shiftColumn* equal to c and *shiftLength* equal to 1.

R2. Select, a column c, where the node n_{rc} has a $checkSpan_r(n_{rc})$ value equal to 1 and there exists the same symbol at the nearest row r_{down} from r ($r_{down} > r$) and column c', (i.e. $M[r][c]=M[r_{down}][c']$), $c \neq c'$. Then, *shiftColumn* will equal to c and *shiftLength* will be (r_{down} -r).

R3. If both rule R1 and R2 fail, we then align the current row individually by dividing it into 2 parts: P_1 and P_2 (such that at least one of them is aligned). In this divide-and-conquer process, we should decide which part comes first in the aligned matrix. The principle is that a part which contains nodes with *checkSpan* value equal to 0 (the aligned part) should come first.

Figure 3 shows an example that describes how the algorithm proceeds. The span values of the nodes a, b, c, d, and e in M_1 are 0, 3, 3, 3, and 0, respectively. The first 3 rows of M_1 are aligned, so there are any changes on them. For the 4th row, according to R_1, node b (at $M_1[4][3]$) is selected to be shifted to the next row, because $checkSpan_4$(b) has value -1. Hence matrix M_2 is obtained. According to rule R_2 (R_1 doesn't apply), node e has a nearest occurrence at the 8th row with *checkSpan* value equal to 1. Therefore, *ShiftColumn*=2 and *ShiftLength*=8-5=3. Similarly, we can follow the selection rule at each row to get the aligned peer matrix M_6. Here, dashes mean null nodes. A node to be shifted at each row is shaded in the matrices of Figure 3.b.

3.3. Repetitive pattern mining

This step is designed to handle set-typed data where multiple-values occur, thus a naïve 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 (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. 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, every instance of a set-type should occur immediately to each other according to the problem definition. Due to space limitation, we didn't present the detailed algorithm in this paper. Finally, we shall add in the pattern tree a virtual node for every set type with size greater than 1.

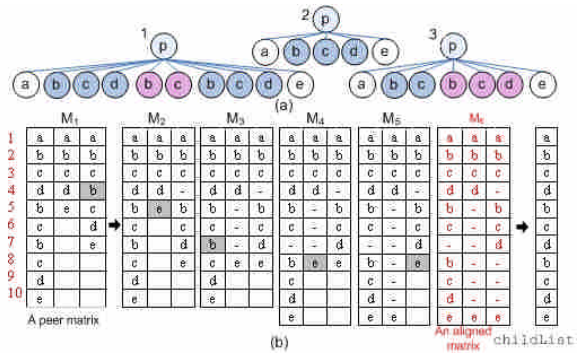

Figure 3: An example of peer matrix alignment.

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, ..., b_u), where b_i is 1 if c occurs in the i^{th} 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 pages. For example, the occurrence vector of nodes a and e in Figure 3.a is (1,1,1). For nodes b (c) and d, the occurrence vectors are (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 two 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}, ..., c_k ($i<k$) have the same occurrence vectors or complement (disjunctive type) occurrence vectors, we shall group them.

Rule 2. If an optional node c_i is a fixed node/template subtree, we shall group c_i, c_{i+1}, ..., c_k, where c_k is the non fixed template tree nearest to c_i.

Table 1. The evaluation of FiVaTech extracted schema and its comparison with EXALG schema.

Dataset: 9 Web sites on EXALG home page.																
site	N	Manual			EXALG						FiVaTech					
		A_m	O_m	{}	A_e	O_e	{}	c	Incorr.		A_e	O_e	{}	c	Incorr.	
									i	n					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%**					

4. Data schema detection

With the fixed/variant pattern tree, the schema of the input pages should be easily deduced by simply removing nodes having single child and without types and preserving all basic leaf nodes. We then mark the order of a node with *k* children as *k*. If an internal node is not a virtual node and not labeled as set type, it is marked as a tuple.

To identify the template for every type τ in the schema *S*, we define the reference node *r* in the fixed/variant pattern tree as follows:

- *r* is a node of a tuple type,
- the next (right) node of *r*, in a preorder traversing of the pattern tree, is a node of type τ, where τ is a basic type node β, a set type {}, or a virtual node,
- *r* is a leaf node on the right most path of a *k*-tuple or *k*-order set and is not of any type.

Now, templates can be identified by segmenting the pre-order traversing of the trees (skipping basic type nodes) at every reference node. We say a template is under a node P if the first node of the template is a child of P. Now, we can fill in the templates for each type as follows. For any *k*-tuple or k-order set $<\tau_1, \tau_2, \ldots, \tau_k>$ at node *n*, where every type τ_i is located at a node n_i, then the parent template P will be the null template or the one containing its reference node if τ is the first data type in the schema tree. If τ_i is a tuple type, 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 set type or 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.

5. Experiments

We conduct two experiments. The first one compares FiVaTech with EXALG, while the second experiment is conducted to evaluate the extraction of search result records (SRRs) and compare FiVaTech with three state-of-the-art approaches: Depta, ViPER [4] and MSE [10]. Unless otherwise specified, we usually take 2-3 Web pages as input.

5.1. FiVaTech as a schema extractor

Given the detected schema S_e and the manually constructed schema S_m for a Web site, Table 1 shows the evaluation for the schema S_e resulted by our system and the comparison with EXALG schema. We use the 9 sites at http://infolab.stanford.edu/~arvind/extract/ without any changes of its manual schema S_m. Columns 1 and 2 show the 9 sites and the number of pages *N* in each site. Columns 3-5 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 set type {}.

Columns 6-8 (12-14) show the details of the schema resulted by EXALG (FiVaTech). Columns 9 and 15 show the number of attributes in S_e that correspond to an attribute in S_m and its extracted values are correct (partially correct). Columns 10 and 16 show the number of incorrect attributes. Columns 11 and 17 show the number of attributes that are not extracted.

Of the 128 manually labeled attributes, 116 are correctly extracted by both EXALG and FiVaTech. However, EXALG produced a total of 153 basic types while 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 produced less number of attributes since we do not analyze the contents inside text nodes.

5.2. FiVaTech as a SRRs extractor

Of the popular approaches that extract SRRs from a Web page, the main problem is to detect record boundary. 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.

To recognize data sections of a Web site, FiVaTech identifies a set of nodes n_{SRRs} that is the outer most set type node, i.e. the path from this node to the root of the schema tree has no other nodes of set type.

Table 2: Comparison results with Depta.

Data set: 11 Web site from Testbed Ver. 1.02.				
	Step 1: SRRs Extraction		**Step 2:** 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: Comparison results with ViPER and MSE

Dataset	**TBDW**		**MSE [10]**	
#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%

In the second experiment, we get the system demo from the author and run Depta on the manually labeled Testbed TBDW Ver. 1.02 [7]. Unfortunately, Depta gives a result only for 11 Web sites, and could not produce any output for the remaining 40 sites. Table 2 shows the results for these 11 sites. For SRRs extraction (columns 2 and 3), we just use Web pages that have multiple data records.

For the second step (columns 3 and 4), by the help of the manually labeled data in Testbed, we count the number of attributes inside data records of each data section (92 attributes). An attribute is considered extracted correctly if 60% of its instances (data items) are extracted correctly and aligned together in one column. In summary, the recall and precision is below 50% for Depta, while FiVaTech has a nearly 90% performance for both precision and recall. We shall analyze the reasons in the next section.

The last experiment compares FiVaTech with ViPER and MSE. We use the 51 Web sites of Testbed to compare FiVaTech with ViPER, and the 38 multiple sections Web sites used in MSE to compare our system with MSE. 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 Web page. FiVaTech fails to extract SRRs when the peer node recognition algorithm incorrectly measures the similarities among SRRs.

6. Conclusion

In this paper, we proposed a new Web data extraction approach, called FiVaTech to merge multiple DOM trees simultaneously. We design a new algorithm for multiple string alignment which takes optional and set-type data into consideration. With the constructed fixed/variant pattern tree, we can easily deduce the schema and template for the input Web site.

Although many unsupervised approaches have been proposed for Web data extraction (see [2] for a survey), very few works (RoadRunner and EXALG [1]) solve this problem at a page-level. The proposed page generation model with tree-based template matches the nature of Web pages. Meanwhile, the merged pattern tree gives very good result for schema and template deduction. For efficiency's sake, we only use 2 or 3 pages as input. Whether more input pages can improve the performance requires further study.

Acknowledgement

Our thanks to Yanhong Zhai and Bing Liu for providing simple tree matching code for us. This work is sponsored by National Science Council under grant NSC96-2221-E-008-091-MY2.

References

[1] Arasu, A. and Garcia-Molina, H., Extracting structured data from Web pages. SIGMOD-03, pp. 337-348, 2003.

[2] Chang, C.-H., Kayed, M., Girgis, M. R., Shaalan, K., A Survey of Web Information Extraction Systems, IEEE TKDE (SCI, EI), Vol. 18, No. 10, pp. 1411-1428, Oct. 2006.

[3] Hsu, C.-N. and Dung, M., Generating finite-state transducers for semi-structured data extraction from the web. Journal of Information Systems 23(8): 521-538, 1998.

[4] Simon, K. and Lausen, G. ViPER: Augmenting Automatic Information Extraction with Visual Perceptions. CIKM 2005.

[5] Tai, K. The tree-to-tree correction problem. J. ACM, (3):422–433, 1979.

[6] Valiente, G. Tree edit distance and common subtrees. Research Report LSI-02-20-R, Universitat Politecnica de Catalunya, Barcelona, Spain, 2002.

[7] Yamada, Y., Craswell, N. , Nakatoh, T., and Hirokawa, S. Testbed for information extraction from deep web. WWW-13, pp. 346–347, New York, NY, USA, 2004.

[8] Yang, W. Identifying syntactic differences between two programs. Softw. Pract. Exper., 21(7):739–755, 1991.

[9] Zhai, Y. and Liu, B. Web Data Extraction Based on Partial Tree Alignment. WWW-14, Japan, pp. 76-85, 2005.

[10] Zhao, H., Meng, W. and Yu, C. Automatic Extraction of Dynamic Record Sections From Search Engine Result Pages. VLDB, pp.989-1000, 2006.