Automatic Discovery of Semantic Structures in HTML Documents

Saikat Mukherjee Guizhen Yang Wenfang Tan I.V. Ramakrishnan
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
{saikat, guizyang, wtan, ram}@cs.sunysb.edu

Abstract

Template-driven HTML documents posses an implicit, *fixed schema denoting concepts and their relationships in a hierarchical fashion. Discovering this schema remains a relatively unexplored problem. By exploiting a key observation that semantically related items in HTML documents exhibit spatial locality, we develop an algorithm for automatically partitioning them into tree-like semantic structures which expose the implicit schema.*

1 Introduction

A growing number of Web sites are maintained by content management software and thus a large number of Web pages are machine-generated via templates. Normally in such Web pages there is *implicitly* a fixed "schema" and what changes is the content. Informally a schema for a Web page represents concepts and relationships among them in a hierarchical fashion. For example, Figure 1 is a screen shot of the New York Times front page (see http://www.nytimes.com). Observe that this page includes: (i) a taxonomy of items such as "NEWS" (consisting of hyperlinks labeled with "International", "National", ...), "OPINION" (consisting of hyperlinks "Editorial/Op-Ed", ...), etc.; (ii) several headlines of news articles where each article begins with a hyperlink labeled with the news headline (*e.g.*, "Inspectors in Iraq ...") followed by the author of the article (*e.g.*, "By JOHN F. BURNS ..."), followed by a time-stamp and a text summary of the article (*e.g.*, "The new zero-tolerance ..."). The schema for this fragment of the New York Times front page therefore includes the taxonomy (which does not change) and the template for the news article. We should point out that the schema will also include several additional elements pertaining to other content appearing in the page.

Knowing this schema explicitly has several uses. For example, it eases the task of formulating queries to retrieve

Figure 1. New York Times Front Page

data from Web documents. In the New York Times example, one can pose a query to retrieve all the links under the "NEWS" item in the taxonomy. Knowledge of the schema is the key to transforming legacy HTML documents into more semantics-oriented document formats such as XML [3] and DAML [1]. Yet another application is audio-browseable Web content. By putting a dialog interface to the content of a Web page which is reorganized based on the knowledge of its schema, a user can easily browse its content using audio. More generally a Web site itself can be navigated using voice commands. Audio browseable Web content can significantly expand the reach of the Web to visually challenged individuals.

The important question then is: *Can the implicit schema in template-driven HTML documents be made explicit?* But discovering schema of HTML documents requires gleaning *semantic* information from HTML tags – a task that is difficult if not impossible to accomplish since their primary purpose is presentation of data in the document. Although steady progress is being made in the development of markup languages that facilitates extraction of semantic knowledge (*e.g.*, XML and DAML), HTML documents continue to proliferate.

Figure 2. Screen Shot of the Semantic Partition Tree for New York Times Front Page

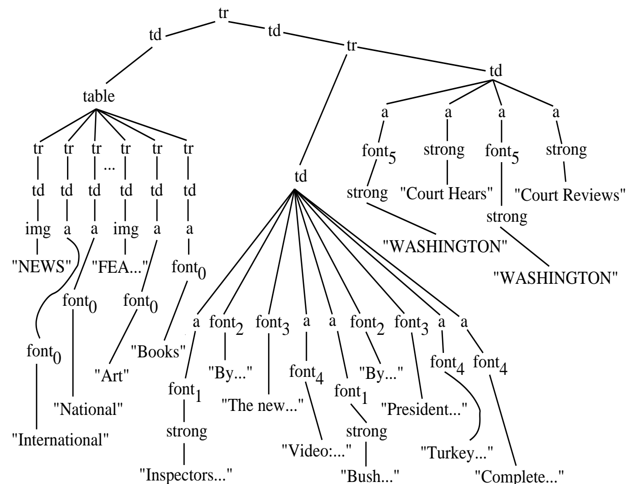

Figure 3. DOM Tree Fragment of New York Times Front Page

In this paper we formulate the problem of schema discovery from HTML documents as one of "automatically discovering semantic structures in HTML documents" and propose an algorithm for it. Our objective is to take a HTML document generated by a template and automatically discover and generate a semantic partition tree. Each partition will consist of items related to a semantic concept. For example, Figure 2 is such a tree corresponding to the New York Times front page in Figure 1. In this figure, a *Partition Node* groups semantically related contents together and structures them into a tree. In this paper we describe an algorithm for automatically discovering semantic structures in template-driven HTML documents.

Our Approach The idea underlying our approach is based on the key observation that in such documents semantically related items, as discerned in their rendered views, exhibit spatial locality. For example, observe that in Figure 1 all the taxonomic items such as "NEWS", "FEATURES", etc., and the corresponding hyperlinks under them are all spatially clustered together in the rendered view. The same observation holds for all the headline news items, their associated authors, and the corresponding news summary. Each of these items appear as the leaf nodes in the DOM [2] tree corresponding to the page (see the DOM tree fragment in Figure 3).

How do we algorithmically determine spatial locality? It turns out that in such HTML documents spatial locality can be captured as "similarity" of path structures in DOM trees. For example, observe that for all the links under the "NEWS" item ("International", "National", etc.) in Figure 1, their root-to-leaf paths (see Figure 3) are all $tr \cdot td \cdot table \cdot tr \cdot td \cdot a \cdot font_0$[1] and hence identical. Similarly, since the items "NEWS", "OPINION", and "FEATURES" are all related in the sense of belonging to the concept of a taxonomy, they also have similar path structures.

We can exploit this observation to define a notion of *similarity* between paths based on their path structures. A strict notion requires that they be identical while weaker forms can be defined based on edit distance. In this paper we chose the former. Based on this notion we can group all the links under the "NEWS" item in one partition, all the links under "FEATURES" in a different partition, and so on. Note that the paths of all these hyperlinked items in each of these partitions are identical. Let us denote any such path as β. Now how do we build the tree of partitioned segments? We reduce this problem to one of finding repeated substrings in a string. Specifically, observe in Figure 3 that the different taxonomic items such as "NEWS" and "FEATURES" have identical root-to-leaf path structures. So they all become part of the same partition. Let us denote this path as α. Notice how all of the taxonomic items and their associated hyperlinks appear within a $table$ structure in the DOM tree – a manifestation of spatial locality. Suppose we label all the paths of all the leaf nodes under this tree (corresponding to the table structure) in sequence. This will result in the string sequence $\alpha\beta^*\alpha\beta^*\ldots$ ($*$ is the Kleene star operator). Since $\alpha\beta^*$ is a repeated substring, all the nodes cor-

[1] Here $font$ tags with different subscripts denote $font$ tags with different attribute values such as $size$, $color$, etc.

responding to each $\alpha\beta*$ will belong to one partition. Thus "NEWS" and all its associated links will belong to one partition. Similarly "FEATURES" and its associated links belong to another partition. Finally both these partitions are made the children of the same parent node. Continuing in this fashion yields the partition tree in Figure 2. Details of this algorithm appear in Section 2.

Contributions We have developed a novel algorithm for automatically discovering semantic structures for template-driven HTML documents. The algorithm exploits a key observation that such structures normally exhibit spatial locality in these documents. Consequently, in contrast to [14, 5], it makes no *a priori* assumptions about the markup tags of HTML document. Thus, our approach is much more scalable and applicable to pages across different domains.

2 Partition Algorithm

2.1 Building Blocks

In this section, we present the essential building blocks of our algorithm for automatically discovering semantic structures and illustrate its working steps using examples.

Our algorithm is based on discovering syntactic similarity in document markup structures. Central to our algorithm is the notion of the *type* of a node in the DOM tree of a HTML document. Intuitively, the type of a node reflects the path structure of the subtree rooted at that node. Formally, we have the following definition.

Definition 1 (Types) *Given a DOM tree T:*

- *Let $t_1, t_2 \ldots, t_k$ be the sequence of HTML tags, with their attribute values, on the path from the root of T to a leaf node of T, then $t_1 \cdot t_2 \ldots t_k$ is a primitive type;*
- *If $T_1, \ldots, T_k$ are types, then $seq(T_1 \ldots T_k)$ is a compound type.*

Given any two types as defined above, their equivalence is defined straightforwardly: two types are equivalent if and only if they are syntactically the same.

Normally, primitive types are assigned to leaf nodes. For instance, in Figure 3, the type of the leaf node "Inspectors..." is $tr \cdot td \cdot tr \cdot td \cdot a \cdot font_1 \cdot strong$. On the other hand, compound types are usually assigned to internal nodes and capture the structural recurrence that is discovered at the subtree rooted at that node. Note that our definition of compound types allows nesting to any arbitrary depth. We will show in the following Section 2.2 how to exploit such a simple type system to restructure documents.

The idea of structural recurrence is centered around the notion of *maximal repeating substrings*.

Definition 2 (Maximal Repeating Substrings) *Given a string S, a substring α that repeats k times in S is a maximal repeating substring if and only if:*

- *$k \geq 2$ and $|\alpha| \times k \geq |S|/2$; and*
- *$|\alpha| \times k$ is the maximum among all substrings that satisfy the above condition; and*
- *k is the maximum among all substrings that satisfy the above two conditions.*

Basically, the above definition says that a maximal repeating substring should be, first of all, a repeating substring that covers a majority of elements. In addition, its coverage should be maximized and its length be minimized (under the prerequisite that its coverage be maximized).

Clearly, given a string S, the number of substrings of S is quadratic in the size of S. Therefore, a naive algorithm which performs an exhaustive search of all these substrings of S can find a maximal repeating substring in polynomial time, if there is such a substring. In the sequel, we will use $MaximalRepeatingSubstring(S)$ to represent any algorithm that returns a maximal repeating substring of the input string S if such a substring exists. Otherwise, we assume that it returns the empty string ε.

2.2 Algorithm

To transform the DOM tree of a HTML document into a tree-like semantic structure, we simply invoke the top-level algorithm $PartitionTree$ on the root of the given DOM tree. This algorithm first traverses the DOM tree top-down and then restructures it bottom-up.

Algorithm PartitionTree(n)
input
    n : a node in a DOM tree
begin
1. **if** n is a leaf node **then**
2. $n.type$ = the sequence of HTML tags from the root to n
3. **else if** n has only one child node c **then**
4.   PartitionTree(c)
5.   Replace n with c and remove n from the DOM tree.
6. **else**
7.   **for** each child node x of n **do** PartitionTree(x) **endfor**
8.   FindPartition(n)
9. **endif**
end

In our data structure, each node of the tree has an additional attribute, $type$, which stores the type assigned to this node. This attribute basically encodes the summary of structural recurrence discovered for the subtree rooted at this node. We will use the notation $n.type$ to represent the $type$ attribute of a node n.

In Line 2 of the algorithm $PartitionTree$, all the leaf nodes are typed. Internal nodes with only one child are handled in Lines 4–5. In such a case, the type of this only child node is computed and then simply propagated up the tree. However, for an internal node with multiple children, we first invoke $PartitionTree$ on all of its children to collect their type information (Line 6). Then the algorithm $FindPartition$ is invoked upon this node to perform a pattern discovery on its children nodes (Line 7).

```
Algorithm FindPartition(n)
input
    n : an internal node in a DOM tree
begin
1.  S = the sequence of all the child nodes of n
2.  for each node c in S do
3.     if c.flatten = true then
4.        Replace c with the sequence of all the child nodes of c.
5.     endif
6.  endfor
7.  τ = ε
8.  do
9.     Collapse adjacent nodes in S which share the same type.
10.    α = MaximalRepeatingSubstring(TypeStr(S))
11.    if α ≠ ε then τ = α endif
12.    if |α| > 1 then
13.       for each substring ρ in S such that TypeStr(ρ) = α do
14.          Replace ρ with NewNode(ρ,seq(α)).
15.       endfor
16.    endif
17. while |α| > 1
18. if τ = ε then
19.    n.flatten = true
20. else
21.    Partition S into β_0γβ_1 ... γβ_m, where TypeStr(γ) = τ.
22.    for each γβ_i do
23.       Replace γβ_i with NewNode(γβ_i, NewType(τ)).
24.    endfor
25.    n.type = NewType(τ)
26. endif
27. Make the nodes in S the new children of n.
end
```

The algorithm $FindPartition$ takes an internal node, n, as input. Its main function is to discover structurally similar items among all the children of n and restructure the subtree rooted at n accordingly. Because our algorithm climbs up a DOM tree from leaf nodes to the root, structural similarity may not be observed until it reaches a node high enough. Therefore, we associate a boolean attribute, $flatten$, with each node to signal whether a structural similarity pattern has been discovered at this node. The value of this attribute is initialized to $false$ for each node. However, if a pattern (or type) is not found at a node, then its $flatten$ attribute is set to $true$ (Line 19).

In Lines 1–6, all the child nodes of n are collected into a sequence, which will be partitioned into semantically related items later if they share structural similarity. But if we encounter a node, c, whose $flatten$ attribute has the value $true$ (which means a pattern is not found at this node), then we move all the child nodes of c into this sequence for further processing.

Note that when the algorithm $FindPartition$ is invoked on a node, all of its descendant nodes are already typed. Intuitively, since the type of a node summarizes the structure of the subtree rooted at that node, analysis of the sequence of sibling types is essential for structural similarity pattern discovery, which is done in two stages by our algorithm.

In the first stage, consecutive nodes having equivalent types are collapsed into a single node (Line 9). The intuition behind this is that they all relate to the same item. Next, in Line 10, an attempt is made to find a maximal repeating substring of the string corresponding to the type sequence of S (returned by $TypeStr(S)$).

If such a substring does not exist (hence no structural similarity), then the loop in Lines 8–17 is exited and the $flatten$ attribute of the current node is set to $true$ (Line 19). However, if a maximal repeating substring, α, is found and α contains at least two elements ($|\alpha| > 1$), then the sequence of consecutive nodes whose type sequence matches α is merged into a new node created by the procedure $NewNode$ (Lines 12–16). The first argument of $NewNode$ contains the sequence of nodes to be merged while the second argument indicates the type of this new node. The above collapsing-pattern-discovering-merging process is repeated until it cannot be performed any more.

In the main part of the second stage (Lines 21–25), the last pattern discovered during the first stage is used to partition the remaining sequence of nodes further. This is a simple heuristic that we apply to handle variations in document structures (*e.g.*, missing data items). Note that if τ contains only one type, then $NewType(\tau)$ returns τ directly; otherwise, it returns the compound type $seq(\tau)$.

Now we illustrate the working steps of the algorithm $FindPartition$ using an example. For simplicity, we will just show how it manipulates a sequence of types and omit other details. Suppose the type sequence of S is $T_1T_2T_3T_2T_3T_4T_1T_2T_3T_5$ immediately before the algorithm executes the loop starting at Line 8. T_2T_3 is a maximal repeating substring. Let us use a new type T_6 to denote $seq(T_2T_3)$. Then after the first iteration of the loop, the type sequence becomes $T_1T_6T_6T_4T_1T_6T_5$. The first two occurrences of T_6 can be collapsed into one, resulting in $T_1T_6T_4T_1T_6T_5$, in which T_1T_6 is a maximal repeating substring. Again, we use a new type T_7 to represent $seq(T_1T6)$. So after the second iteration the type sequence becomes $T_7T_4T_7T_5$ and the loop terminates. It is not hard to see that the first T_7 and the following T_4 will be put into one partition and the rest into another partition. T_7 is the type assigned to the current node.

The algorithms $PartitionTree$ and $FindPartition$ are illustrated using the DOM tree fragment shown in Figure 3. Let us consider the subtree rooted at the node td spanning the leaf nodes from "Inspectors..." to "Complete...". The type of the "Inspectors..." leaf node, denoted by T_1, is $tr \cdot td \cdot tr \cdot td \cdot a \cdot font_1 \cdot strong$. Observe that the leaf node "Bush..." has the same type T_1. So we can assign the types $T_1, T_2, T_3, T_4, T_1, T_2, T_3, T_4, T_4$ to the leaf nodes from "Inspectors..." to "Complete...", respectively. Observe that all these leaf nodes are the only child of their parent node. As a result, their ancestor nodes are deleted (Lines 4–5 of $PartitionTree$) until they are propagated up the subtree and become siblings under the nearest td node.

Now the algorithm $FindPartition$ is invoked on the sequence of types $T_1T_2T_3T_4T_1T_2T_3T_4T_4$. First, the last two consecutive occurrences of T_4 are collapsed together (Line 9 of $FindPartition$). The resulting type sequence is $T_1T_2T_3T_4T_1T_2T_3T_4$, in which $T_1T_2T_3T_4$ is a maximal repeating substring. So the original sequence of nodes is partitioned into two parts, each corresponding to the pattern $T_1T_2T_3T_4$. The type assigned to the td node (nearest to the "Inspectors..." leaf node) is $seq(T_1T_2T_3T_4)$.

3 Experimental Results

We have implemented the algorithms presented in this paper using Java. We chose the *portals*, *news*, and *office products* domains to measure the efficacy of our algorithms. Specifically, we selected Yahoo (http://www.yahoo.com), New York Times (http://www.nytimes.com), and Office Max (http://www.officemax.com) as representative examples of these domains. Observe that while the structural layouts of these three pages differ widely yet all of them have some regularity in the way semantic concepts are presented. The results of our algorithms on these pages are displayed in Figures 5, 2, and 6, respectively.[2]

Figure 4 shows the time taken to execute the algorithms on a Pentium III 800MHz machine with 256MB memory. The low execution time demonstrates the potential of our system for applications requiring fast processing times, *e.g.*, as a navigation guide for the visually impaired.

Web Site	HTML File Size (KB)	Exec. Time (ms)
nytimes	79.2	731
yahoo	54.7	691
officemax	67.5	811

Figure 4. Timing Statistics

In Figure 2 our system was able to successfully group all the links "International", "National", etc. into a single partition. Similar results were achieved with other partitions such as "Opinion", "Features", etc. In Figure 5 our algorithm properly partitioned the directories "Shop", "Find", etc. and "Business & Economy", "Computers & Internet", etc. Figure 6 shows our hierarchical partition tree for OfficeMax. The fragment shows the taxonomy consisting of "Supplies", "Furniture", etc. as well as the promotions in that page such as "Max Means More...", "Extra Savings...", etc. Observe that we were able to label some of the partitions (*e.g.*, the partition labeled with "Find" in Figure 5). We will discuss the issues of labeling partitions in Section 5.

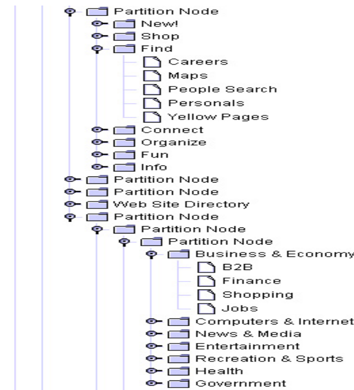

Figure 5. Screen Shot of the Semantic Partition Tree for Yahoo Front Page

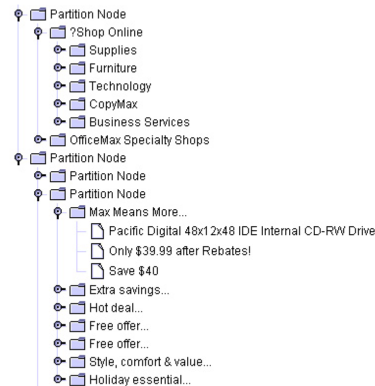

Figure 6. Screen Shot of the Semantic Partition Tree for OfficeMax Front page

4 Related Work

There is a large body of work on discovering schema information from either XML documents [8, 11, 7] or XML queries [12, 13]. However, the problem of recovering semantic structures from HTML documents has only been explored recently.

[2] Due to space limitations only a fragment is shown

In [14] Yang and Zhang propose to build semantic structures from HTML documents by detecting patterns and separation boundaries between different categories in the sources. They view a HTML document as a sequence of HTML tags and texts instead of a tree structure. Their pattern discovery technique relies on a hand-coded similarity function that measures the "distance" between different HTML tags and texts. However, the threshold values of this function must be set *a priori* and may need to be manually adjusted when their technique is applied to different domains. Moreover, they do not consider the problem of labeling a partition.

The work of Chung et al. [5] takes advantage of tree structures of HTML documents to transform them into XML counterparts. They assume that all the input documents are already known to belong to a particular domain of interest and have homogeneous content. Therefore, their approach can make better use of domain knowledge that is hand-coded into a concept classifier to identify elementary concepts and group them into bigger, structural concepts. However, their techniques do not fully explore layout regularity which is commonly observed in template-driven HTML documents.

Finally, it is worth contrasting the problem of schema discovery for template-driven HTML documents to the important, well-studied problem of wrapper-based data extraction [9, 6, 10]. We should point out that wrappers generate domain-specific queriable interface to HTML documents which is orthogonal to the schema discovery problem.

5 Discussion and Future Work

In this paper we described an algorithm for automatic discovery of semantic structures in template-driven HTML documents and provided preliminary experimental evidence of its efficacy in practice.

The output of our algorithm is a tree of semantic partitions. The information associated with a partition is the content present in the leaf nodes of the partition. This information is summarized by a *label* of the partition. It is important to label the semantic partitions for the purpose of recovering the implicit schema of the document. Labeling any arbitrary partition may involve complicated natural language processing and thus is a difficult problem in general.

However, under certain circumstances, it is possible to label a partition using heuristics based on syntactic analysis.[3] One such heuristic is to make the content associated with the first type in a maximal repeating substring of types as the label of the partition. This is illustrated in Figure 2 where the content of the first type, "Inspectors in Iraq...", in the repeating substring becomes the label of the entire partition. Another labeling heuristic that we used is that the content of a type, T_i, preceding a collection of (consecutive) types, T_j's, where T_i and all T_j's are siblings, is made the label of this collection of T_j's (and so this collection of T_j's is "sunk" under T_i). This is illustrated in Figure 5 where "Find" is the label of the sequence "Careers", "Maps", etc.

The heuristics based on syntax alone, however, are not enough to label all partitions properly. We are exploring the use of domain knowledge, through WordNet [4] or through ontologies, for a better labeling algorithm.

Moreover, note that our current algorithm is purely based on syntactic analysis and there is room for further improvement. In particular we can use light-weight semantic information, gleaned from sources such as WordNet [4], to assign costs to partitions and formulate the discovery of semantic structures as an optimization problem. The semantic information latent in partitions can be deployed for developing *self-repairing* wrappers for web sites. Brittleness of wrappers due to web site changes is a critical problem. Coupling semantic knowledge with purely syntax-based data extraction techniques that are currently employed in wrappers can facilitate self-repair. The above problems are worthy of further investigation.

[3]For want of space, we omitted the steps involved in labeling a partition in our algorithms presented in this paper.

References

[1] DARPA agent markup language. http://www.daml.org/.
[2] Document object model. http://www.w3.org/DOM/.
[3] Extensible markup language. http://www.w3.org/XML/.
[4] WordNet. http://www.cogsci.princeton.edu/∼wn/.
[5] C. Y. Chung, M. Gertz, and N. Sundaresan. Reverse engineering for web data: From visual to semantic structures. In *ICDE*, 2002.
[6] W. Cohen, M. Hurst, and L. Jensen. A flexible learning system for wrapping tables and lists in html documents. In *International World Wide Web Conference*, 2002.
[7] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: A system for extracting document type descriptors from xml documents. In *ACM SIGMOD*, 2000.
[8] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.
[9] J. Hammer, H. Garcia-Molina, S. Nestorov, R. Yerneni, M. M. Breunig, and V. Vassalos. Template-based wrappers in the TSIMMIS system. In *ACM SIGMOD*, 1997.
[10] L. Liu, C. Pu, and W. Han. XWRAP: An XML-enabled wrapper construction system for web information sources. In *ICDE*, 2000.
[11] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *ACM SIGMOD*, 1998.
[12] Y. Papakonstantinou and P. Velikhov. Enhancing semistructured data mediators with document type definitions. In *ICDE*, 1999.
[13] Y. Papakonstantinou and V. Vianu. DTD inference for views of xml data. In *ACM PODS*, 2000.
[14] Y. Yang and H. Zhang. HTML page analysis based on visual cues. In *ICDAR*, 2001.