

# OntoMiner: Bootstrapping and Populating Ontologies From Domain Specific Web Sites

Hasan Davulcu, Srinivas Vadrevu, and Saravanakumar Nagarajan

Department of Computer Science and Engineering,  
Arizona State University,  
Tempe, AZ, 85287, USA  
{hdavulcu, svadrevu, nrsaravana}@asu.edu

**Abstract.** RDF/XML has been widely recognized as the standard for annotating online Web documents and for transforming the HTML Web to the so called Semantic Web. In order to enable widespread usability for the Semantic Web there is a need to bootstrap large, rich and up-to-date domain ontologies that organize most relevant concepts, their relationships and instances. In this paper, we present automated techniques for bootstrapping and populating specialized domain ontologies by organizing and mining a set of relevant Web sites provided by the user. We develop algorithms that detect and utilize HTML regularities in the Web documents to turn them into hierarchical semantic structures encoded as XML. Next, we present tree-mining algorithms that identify key domain concepts and their taxonomical relationships. We also extract semi-structured concept instances annotated with their labels whenever they are available. Experimental evaluation for the News and Hotels domain indicates that our algorithms can bootstrap and populate domain specific ontologies with high precision and recall.

## 1 Introduction

RDF and XML has been widely recognized as the standard for annotating online Web documents and for transforming the HTML Web to the so called Semantic Web. Several researchers have recently questioned whether participation in the Semantic Web is too difficult for “ordinary” people [1–3]. In order to enable widespread useability for the Semantic Web there is a need to bootstrap large, rich and up-to-date domain ontologies that organizes most relevant concepts, their relationships and instances. In this paper, we present automated techniques for bootstrapping and populating specialized domain ontologies by organizing and mining a set of relevant Web sites provided by the user. As an example application, a user of the OntoMiner can use the system to rapidly bootstrap and ontology populated with instances and they can tidy-up the bootstrapped ontology to create a rich set of labeled examples that can be utilized by supervised machine learning systems such as the WebKB[4].

The user of the OntoMiner system only need to provide the system the URLs of the Home Pages of 10 to 15 domain specific Web sites that characterizes her

domain of interest. Next, OntoMiner system detects and utilizes the HTML regularities in Web documents and turns them into hierarchical semantic structures encoded as XML by utilizing a hierarchical partition algorithm. We present tree-mining algorithms that identifies most important key domain concepts selected from within the directories of the Home Pages. OntoMiner proceeds with expanding the mined concept taxonomy with sub-concepts by selectively crawling through the links corresponding to key concepts. OntoMiner also has algorithms that can identify the logical regions within Web documents that contains links to instance pages. OntoMiner can accurately separate the “human-oriented decoration” such as navigational panels and advertisement bars from real data instances and it utilizes the inferred hierarchical partition corresponding to instance pages to accurately collect the semi-structured concept instances.

A key characteristic of OntoMiner is that, unlike the systems described in [5, 6] it does not make any assumptions about the usage patterns of the HTML tags within the Web pages. Also, OntoMiner can separate the data instances from the data labels within the vicinity of extracted data and attempts to accurately annotate the extracted data by using the labels whenever they are available. We do not provide algorithms for extracting and labeling data from within HTML tables since there are existing solutions for detecting and wrapping these structures [7, 8].

Other related work includes schema learning[9–11] for semi-structured data and techniques for finding frequent substructures from hierarchical semi-structured data[12, 13] which can be utilized to train structure based classifiers to help merge and map between similar concepts of the bootstrapped ontologies and better integrate their instances.

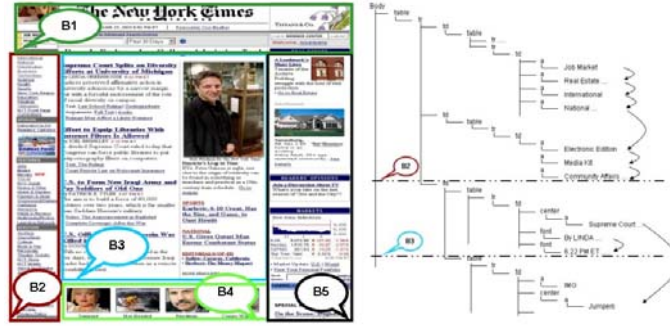
The rest of the paper is organized as follows. Section 2 outlines the hierarchical partitioning, Section 3 discusses taxonomy mining, Section 4 describes instance mining. Experimental evaluation for the News and Hotels domains indicates that our algorithm can bootstrap and populate domain specific ontologies with high precision and recall.

## 2 Semantic Partitioning

### 2.1 Flat Partitioner

Flat Partitioner detects various logical partitions of a Web page. For example, for the home page of <http://www.nytimes.com>, the logical partitions are marked in boxes B1 through B5 in Figure 1. The boxes in snapshot of Web page in Figure 1 correspond to the dotted lines shown in tree view of Web page in Figure 1..

The Flat Partitioner Algorithm takes an ordered DOM tree of the Web page as input and finds the flat partitions in it. Intuitively, it groups contiguous similar structures in the Web pages into partitions by detecting a high concentration of neighboring repeated nodes, with similar root-to-leaf tag-paths. First, the partition boundary is initialized to be the first leaf node in the DOM tree. Next, any two leaf nodes in the tree are linked together with a “similarity link”



**Fig. 1.** Snapshot of New York Times Home Page and Parse Tree View of the Home Page

if they share the same path from the root of the tree and all the leaf nodes in between have different paths. Then the ratio of number of "similarity links" that crosses the current candidate boundary to the total number of "similarity links" inside the current partition is calculated. If this ratio is less than a threshold  $\delta$ , the current node is marked as the partition boundary. Otherwise, current node is added to the current partition and the next node is considered as the partition boundary. The above process terminates when the last element in the list of leaf nodes is reached. A Path Index Tree (PIT) is built from the DOM tree of the Web page, which helps to determine all the "similarity links" between the leaf nodes within a single traversal. The PIT is a trie based data structure which is made up of all unique root to leaf tag-paths and, in its leaf nodes PIT stores the "similarity links" between the leaf nodes of the DOM tree.

The tree view in Figure 1 illustrates the Flat Partitioning Algorithm. The arrows in the tree view in Figure 1 denote the "similarity links" between the leaf nodes. Let's assume the threshold  $\delta$  is set to 60%. Then, when the current node is "Job Market" the total number of outgoing unique "similarity links" (*out in line 9*) is 1 and total number of unique "similarity links" (*total in line 10*) is 1. Hence the ratio of *out* to *total* is 100% which is greater than threshold. Hence *current in line 6* becomes the next leaf node. At node "International", *out* becomes 1 and *total* is also 1. Hence the ratio is still greater than threshold. When *current* reaches "Community Affairs", *out* becomes 0 whereas *total* is 1 and hence the ratio is less than threshold  $\delta$ . Now, "Community Affairs" (*B2 in Figure 1*) is added to the set of *partition\_boundaries in line 12* and all the "similarity links" are removed from the *partition\_nodes in line 13*. The same boundary detection condition is satisfied once again when the algorithm reaches "6.22 PM ET" where *out* becomes 1 and *total* is 3. Hence "6.22 PM ET" (*B3 in Figure 1*) is added to the *partition\_boundaries*.

**Algorithm 1** Flat Partition Algorithm*Flat Partitioner**Input: T: DOM Tree**Output:  $\langle b_1, b_2, \dots, b_k \rangle$ : Flat Boundaries*


---

```

1: PIT := PathIndexTree(T)
2: current := first leaf node of T
3: Partition_Nodes :=  $\phi$ 
4: Partition_Boudaries :=  $\phi$ 
5: for each lNode in Leaf_Nodes(T) do
6:   current := lNode  $\rightarrow$  next
7:   if N = PIT.next_similar(Current) exists then
8:     Partition_Nodes := Partition_Nodes  $\cup$  N
9:   end if
10:  out =  $|\{path(m) | m \in Partition\_Nodes \text{ and } m > current\}|$ 
11:  total =  $|\{path(m) | m \in Partition\_Nodes\}|$ 
12:  if out/total  $\geq \delta$  then
13:    Partition_Boundaries := Partition_Boundaries  $\cup$  current
14:    Partition_Nodes :=  $\phi$ 
15:  end if
16: end for
17: Return Partition_Boundaries

```

---

**2.2 Hierarchical Partitioning**

Hierarchical Partitioner infers the hierarchical relationships among the leaf nodes of the HTML parse tree where all the page content is stored. The Hierarchical Partitioner achieves this through sequence of three operations: Binary Semantic Partitioning, Grouping and Promotion.

**Binary Semantic Partitioning** The Binary Semantic Partitioning of the Web page relies on a dynamic programming algorithm which employs the following cost function. The dynamic programming algorithm determines the nodes that need to be grouped together, by finding the grouping with the minimal cost. The cost for grouping any two nodes in the HTML parse tree is recursively defined as follows.

$$\begin{aligned}
& - Cost(L_i, L_j) = 0, \text{ if } i = j \\
& - Cost(L_i, L_j) = \\
& \quad \min_{i \leq k < j} \{ Cost(L_i, L_k) + Cost(L_{k+1}, L_j) + \\
& \quad \quad \quad Grouping\_Cost(L_{i\dots k}, L_{k+1\dots j}) \}, \text{ if } i < j
\end{aligned}$$

Where  $L_i, L_j$  are two leaf nodes in the HTML parse tree.

The cost function calculates the measure of dissimilarity between two nodes i.e. a high value of cost indicates that these two nodes are highly dissimilar.

Hence the dynamic programming algorithm finds the lowest cost among the various possible binary groupings of nodes and parenthesizes them into a binary-tree. The cost for grouping two consecutive sub trees is calculated as the sum of four sub-cost factors. Let  $A$ ,  $B$  be the least common ancestor of nodes  $L_i$  to  $L_k$  and  $L_{k+1}$  to  $L_j$  respectively. Then,

$$\begin{aligned} \text{Grouping\_Cost}(A, B) = & \\ \text{Sum of distances of A and B to their LCA, } & C_{LCA}(A, B) + \\ \text{Similarity of the paths from A and B to their LCA, } & C_{PSIM}(A, B) + \\ \text{Similarity of the paths in the sub trees of A and B, } & C_{STSIM}(A, B) + \\ \text{Order similarity of the paths in the sub trees of A and B, } & C_{ORD}(A, B) \end{aligned}$$

The first cost factor  $C_{LCA}(A, B)$  calculates how far the two nodes are apart from their least common ancestor. The cost for similarity between paths to the least common ancestor is determined by the second cost factor  $C_{PSIM}(A, B)$ . The third  $C_{STSIM}(A, B)$  and fourth  $C_{ORD}(A, B)$  cost factors computes the cost for similarity in the sub trees of the two nodes, former computes the similarity in the paths whereas the later computes the ordering of paths in the sub tree.

Let  $S_1$  be the set of all paths in the sub tree of A,  $S_2$  be the set of all paths in the sub tree of B,  $d_1$  be the number of tags on the path from LCA to A,  $d_2$  be the number of tags on the path from LCA to B and max depth be the maximum depth of the DOM tree.

$$C_{LCA}(A, B) = \sqrt{\frac{d_1 + d_2}{2 * \text{max\_depth}}}$$

$$C_{PSIM}(A, B) = 1 - \frac{\text{Similarity between Paths } P_1 \text{ and } P_2}{\text{max}(d_1, d_2)}$$

$$C_{STSIM}(A, B) = 1 - \text{max}(\text{Separation}, \text{Overlap}),$$

$$\text{where Separation} = \frac{|(S_1 - S_2) \cup (S_2 - S_1)|}{|S_1 \cup S_2|} \text{ and Overlap} = \frac{S_1 \cap S_2}{S_1 \cup S_2}$$

$$C_{ORD}(A, B) = 1 - \text{Sim}(A, B),$$

$$\text{where } \text{Sim}(A, B) = \frac{\text{Number of Paths similar in order in Sub Trees of A and B}}{\text{Max of No of Paths in Sub Trees of A and B}}$$

For example, let a / b / c be the three tags on the path from LCA to A and a / b / d be the tags on the path from LCA to B. Let  $P_1, P_2, P_3$  be the set of paths in the sub tree of A and  $P_1, P_2, P_4$  be the set of paths in the sub tree of B.

$$d_1 = |a, b, c| = 3, d_2 = |a, b, d| = 3$$

$$S_1 = P_1, P_2, P_3, S_2 = P_1, P_2, P_4$$

$$C_{LCA}(A, B) = \sqrt{\frac{3 + 3}{2 * \text{max\_depth}}}$$

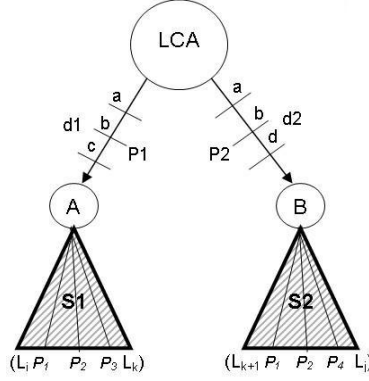


Fig. 2. Sample Tree

$$C_{PSIM}(A, B) = 1 - \frac{|\{a, b, c\} \cap \{a, b, d\}|}{\max(d_1, d_2)} = \frac{1}{3}$$

$$Separation = \frac{|\{P_3\} \cup \{P_4\}|}{|\{P_1, P_2, P_3, P_4\}|} = \frac{1}{2}$$

$$Overlap = \frac{|\{P_1, P_2\}|}{|\{P_1, P_2, P_3, P_4\}|} = \frac{1}{2}$$

$$C_{STSIM}(A, B) = 1 - \max(Separation, Overlap) = \frac{1}{2}$$

$$Sim(A, B) = \frac{|\{P_1, P_2\}|}{\max(|S_1|, |S_2|)} = \frac{1}{2}$$

$$C_{ORD}(A, B) = 1 - Sim(A, B) = \frac{1}{3}$$

These cost functions are adjusted to fit for different cases in the HTML Parse Tree. The three different cases that may arise during the cost function evaluation are shown in Figure 2.

- Case 1: LCA of the two nodes which are to be grouped in one partition is one of the nodes itself and the other node is not a leaf node.
- Case 2: LCA of the two nodes which are to be grouped in one partition is one of the nodes itself and the other node is a leaf node.
- Case 3: the LCA nodes for the ranges are identical.

In all the three cases, the second and fourth cost factors are irrelevant and hence they are ignored. For Case 3, first cost factor is also ignored. Accordingly, the first and third cost factors are modified as follows. For Case 1,

$$C_{LCA} = \frac{d}{\maxdepth} \quad C_{STSIM} = 1 - \max(Separation, Overlap)$$

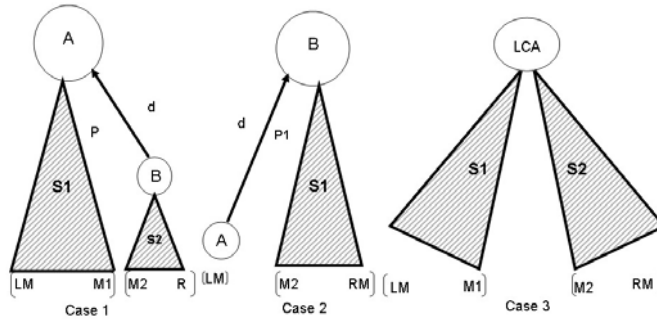


Fig. 3. Different Cases during Cost Function Evaluation

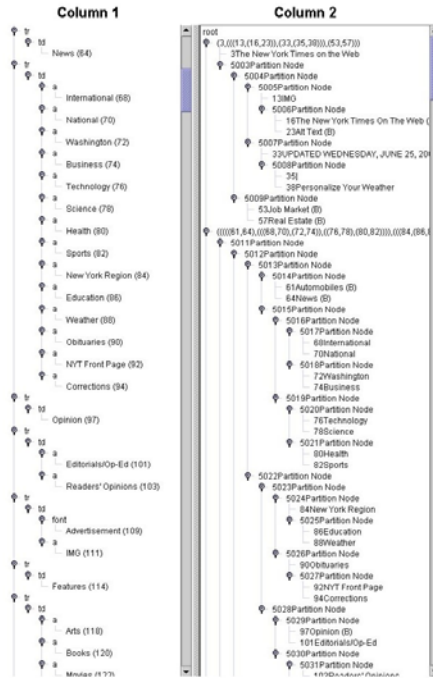


Fig. 4. Dynamic Programming

For Case 2,

$$C_{LCA} = \frac{d}{maxdepth}, \quad C_{STSIM} = 1 - \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} \text{ where } S_2 \text{ is } \{P_1\}$$

For Case 3,

$$C_{SIM}^{ST} = 1 - max(Separation, Overlap)$$

The total cost is divided by the number of applicable cost factors to normalize the cost to a value between 0 and 1. The above dynamic programming algorithm takes the DOM tree as input and produces semantic binary-tree partitions of its leaf nodes. The Column 1 of the Figure 4 represents the DOM tree of the HTML page and Column 2 represents the binary Partition Tree. For example the nodes 68 through 82 are grouped into one partition which has internal binary partitions as shown in Figure 3.

**Grouping** The next step in the Hierarchical Partitioning is grouping of similar binary partitions into group nodes. Grouping Algorithm first finds pairs of partitions which are similar by post order traversal of semantic binary partition tree. Intuitively the grouping step creates "Group" nodes made up of "Instance" nodes as its children. The Instances are identified during the post order traversal of the semantic binary partition tree whenever the "similarity" between a right sibling and its left sibling is above a certain threshold  $\delta$ . The "similarity" between siblings is based on the Grouping\_Cost explained in Binary Semantic Partition section. Then, the parent of the "Instance" nodes is marked as "Group" node. During the rest of the post order traversal the similarity between an internal node and a "Group" node is calculated by evaluating the similarity between the unmarked node and the first Instance of the "Group" node.

The Grouping Algorithm first initializes the type of the leaf nodes in the binary partition tree as "simple". While traversing the tree, if it finds two "simple" sibling nodes and if the cost for grouping these two nodes is less than a threshold  $\delta$ , then it marks these nodes as "Instances" and their parent as "Group" node. For example, in Figure 5, nodes "Sports" and "Health" are sibling nodes and the cost for grouping these two nodes is also less than the threshold  $\delta$ . Hence both are marked as "Instance" node and their parent is marked as "Group" node. Similarly, if it finds two sibling nodes that are marked as "Group" and if the cost for grouping their instances is less than threshold  $\delta$ , then it marks the parent of these sibling nodes as "Group". For example, the parent of nodes "Health" and "Sports" is already marked as "Group" node. Similarly, the parent of "Science" and "Technology" is also marked as "Group" nodes. Then, if the cost for grouping all the instances "Technology" through "Sports" is also less than threshold  $\delta$ , then grand parent of these instances is marked as "Group" node and their instances are merged as seen in Column 3 of Figure 4. Next, if one of the sibling nodes is "simple" and the other node is "Group" and the cost for grouping the "simple" node with the instances of the group node is also less than threshold  $\delta$ , then it changes the type of "simple" node to "Instance" and marks their parent as "Group" and merges the "Instance" node with instances of the "Group" node. This operation is continued until the root of the binary Partition tree is reached and all markings are done. Figure 4 shows the conversion of binary partition tree to Group Tree. The Column 2 and Column 3 represent the binary partition and Group trees.



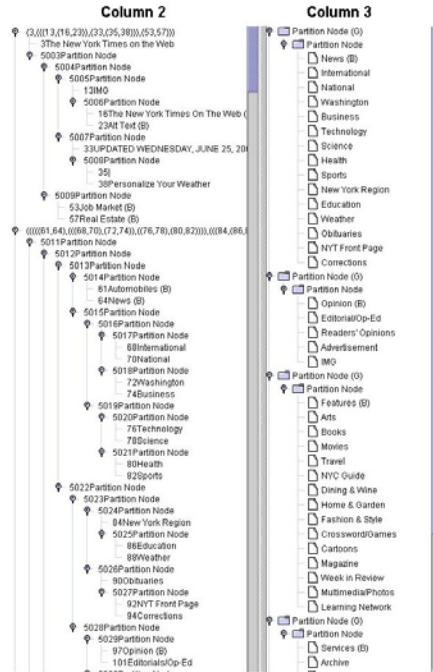


Fig. 5. Grouping

**Promotion** After Grouping, the final step in Hierarchical Partitioning is promotion. The promotion algorithm identifies the leaf nodes that should be promoted above their siblings. A node is considered for promotion only if it applies to the following rules.

- **Rule 1:** A node can be marked as BOLD and it is the first child of its parent and the parent is not marked as "Group" node. A node is marked as BOLD, if it satisfies any of the following conditions.
  1. If there is a bold tag like  $\mathbf{j}b_{\mathbf{j}}$ ,  $\mathbf{j}bold_{\mathbf{j}}$  etc. on its path from the root node
  2. If any bold tag is defined for the "class" attribute value of this node.
  3. If the labeled text of the node is fully capitalized.
- **Rule 2:** A node can be promoted if it is the first child of its parent and its parent is not marked as "Group" node and the only other sibling to this node is a "Group" node.

The nodes which satisfy the bold conditions are marked as BOLD nodes (indicated by letter (B) in column 3).The BOLD node replaces its parent "Partition Node". If the promotion rules can be applied again, the BOLD node is promoted once more. Figure 6 illustrates the Promotion Algorithm. Column 3 represents the Group Tree and Column 4 represents the Hierarchical Partition Tree. The Node "News" is marked as BOLD and it is the first child of its parent as shown

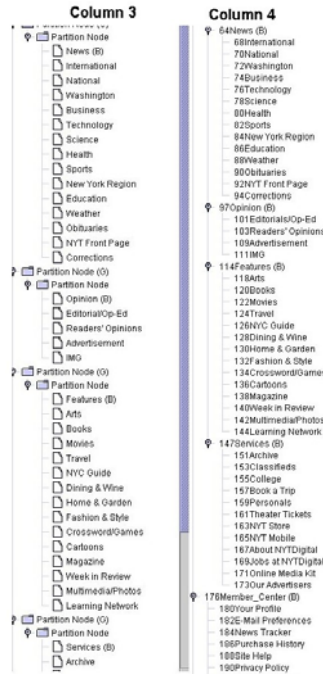


Fig. 6. Promotion

in Column 3 of Figure 6. Hence it is promoted on top of all the nodes "International" through "Corrections". Similarly the nodes "Opinion", "Features" and "Services" are also promoted.

**Experimental Results** In order to calculate precision and recall for the generated Hierarchical Partitioning Trees, Ideal Hierarchical Semantic Partitioned Trees are manually generated for every page. Next, transitive closure of all parent-child relationships implied by each tree is generated. The Precision and Recall are calculated as follows.

The algorithm is applied for home pages of the following 13 websites and the experimental results are shown in the Table 1.

### 3 Taxonomy Mining

The taxonomy mining involves several tasks including separating important concepts (the categories that define the context) from instances (the content that are members of each concept), identification of similar concepts, and mining relationships among the concepts. Our goal is to automatically mine the taxonomy for a domain given relevant web pages from the domain. To demonstrate the efficacy of our algorithms we implemented and tested our approach with two

Domain	Precision	Recall
www.abcnews.go.com	0.79	0.86
www.ninemsn.com	0.75	0.72
www.cbc.ca	0.94	0.73
www.cnn.com	0.81	0.75
www.foxnews.com	0.83	0.82
www.msnbc.com	0.80	0.86
www.nytimes.com	0.83	0.88
www.time.com	0.80	0.86
www.timesonline.co.uk	0.84	0.92
www.un.org	0.79	0.86
www.usnews.com	0.85	0.64
www.washingtonpost.com	0.77	0.71
www.washingtontimes.com	0.75	0.73
Average	0.81	0.80

Table 1. Precision and Recall for Hierarchical Partitioner

Fig. 7. Experimental Results for Hierarchical Partitioning

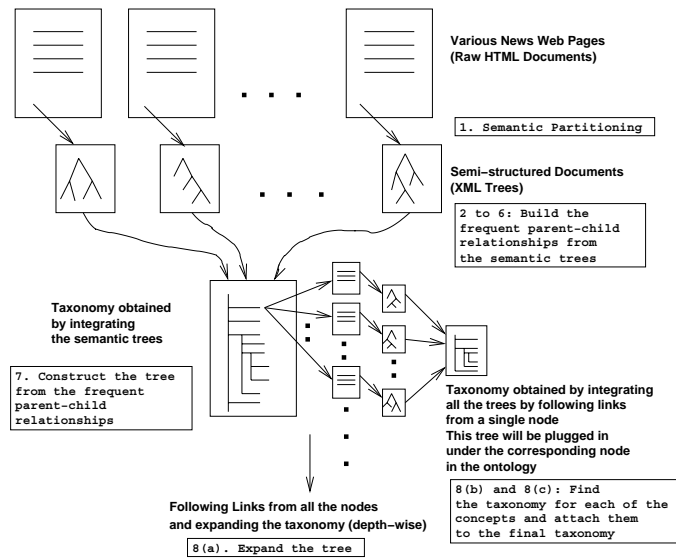


Fig. 8. Main Idea of the algorithm. The numbers in the boxes indicate the step numbers in the algorithm given in Table 1

separate domains; News Web pages and Hotel Web pages. The key ideas in taxonomy mining are illustrated in Figure 8. Various phases involved in taxonomy mining are explained in the following subsections.

### 3.1 Frequency Based Mining

The inputs to our system are the Home Pages of the co-domain Websites. We first preprocess the HTML documents using “Hierarchical Partitioning” (as described in Section 2) to generate semi-structured XML documents. We use these XML documents to mine the taxonomy. We exploit the observation that important concepts in a given domain are often frequent. Using this observation, our system mines frequent labels in the input XML documents among Home Pages of co-domain Websites. By using an experimentally determined threshold for support (the minimum number of times a label should occur in order to be frequent), we separate concepts from the rest of the document. For example in News domain, our system identifies Business, Sports, Politics, Technology, Health, Entertainment, etc. as important concepts as they are frequent across various news home pages.

### 3.2 Candidate Label Extraction Phase

In our simple frequency based mining, our system may miss some labels that are relevant but are not frequent (not present in many Home Pages pages for the domain). For example in <http://www.washtimes.com/>, our system identified “Entertainment” to be a frequent label but it missed “Civil War”, “Culture, etc.”. To identify such relevant labels our system learns attributed tag paths to the labels in the obtained from frequency based mining and applies these paths on the corresponding regions of the Home Pages to retrieve more labels. An attributed tag path of a label has XPath syntax and it is made up of HTML tag names along its path from the root of the HTML tree to the label itself with attributes and their values. For example the attribute tag path for “Entertainment” in <http://www.washtimes.com/> is `//HTML//BODY[@bgColor]//TABLE[@cellpadding=0 and @cellspacing=0 and border=0 and @width=760]//TR//TD[@width=140 and @rowspan=8 and @valign=top]//TABLE[@cellpadding=0 and @cellspacing=0 and @border=0 and @width=140]//TR//TD[@height=20 and @class=twt-menu1a]//A//SPAN[@class=twt-mentext1]//#TEXT.`

### 3.3 Abridgment Phase

During the extraction phase, it is possible to identify some labels that are irrelevant to the domain (for example, “NYT Store” in <http://nytimes.com/>). To eliminate these irrelevant labels, we adopt the following rules.

- Eliminate a label if it does not have a URL or if the URL goes out of the domain.
- Eliminate a label if its URL does not have new frequent labels and valid instances (as described in ??).

---

**Algorithm 2** Mine is-a Relationships

---

*is-aMiner* Input:  $C$ : Set of Concepts,  $S$ : Set of Semantically Partitioned Web Pages,  $Sup$ : Support

Output: Tree representing the hierarchy of concepts

---

- 1: Compute parent-child relationships  $R$  among set of concepts  $C$  along with their frequencies:  $a - b$  is a direct parent-child relationship if  $a$  and  $b$  are concepts and  $b$  is immediate child of  $a$
  - 2: Separate frequent relationships (those that satisfy the minimal support  $Sup$ ),  $FR$  from the non-frequent ones,  $NFR$ .
  - 3: Identify the grand-parent relationships from  $NFR$ : for all non-frequent relationships  $a - b$  and  $b - c$ , increment the count for  $a - c$  in  $R$ .
  - 4: Populate frequent relationships  $FR$  from  $R$  again based on the support,  $Sup$ .
  - 5: Construct a tree,  $T$  from the relationships  $FR$
  - 6: Return the tree  $T$
- 

### 3.4 Grouping the Labels into Concepts

From the above phases, we collect the important labels (keywords) from the relevant Home Pages. But the same label may appear different in various documents and this introduces redundancy. To accommodate for this, we group them according to their lexicographic similarity. First our system stems the labels using Porter Stemming Algorithm [14] and then it applies Jaccard's Coefficient [15] (calculated as  $\frac{|A \cap B|}{|A \cup B|}$ , where  $A$  and  $B$  are stemmed vectors of words in two labels) on them to organize into groups of equivalent labels. We denote each such group of corresponding labels as a *concept*. This simple similarity measure is able to group the labels that are only lexicographically related (like "Sport" and "Sports") but does not identify labels that are semantically related (like "World" and "International"). The issue of identifying and merging semantically related labels is beyond the scope of this paper and we plan to investigate it later.

### 3.5 Mining Parent-Child Relationships From Hierarchically Partitioned Web Pages

The concepts obtained from the grouping phase are flat (there are no is-a relationships among them). In order to organize the concepts in a taxonomy, we need 'is-a' relationships among the concepts. These relationships are mined from the hierarchically partitioned web pages generated from "Semantic Partitioning" (Section 2 using the algorithm described in Algorithm 2. After this phase, we have a taxonomy of concepts that represents the input home pages.

### 3.6 Expanding the Taxonomy beyond Home Pages

The taxonomy obtained from the previous phase represents only the Home Pages in the domain. In order to expand the domain taxonomy, we follow the links

**Algorithm 3** Algorithm to Mine Hierarchy of Concepts From Home Pages*TreeMiner**Inputs: N: The Root Node, H: Set of Home Pages, Sup: Support Output: The Taxonomy of concepts*

- 1: Semantically Partition the input Home Pages to obtain semi-structured XML documents and add them to  $S$
- 2: Collect all the labels along with their URLs and their base URLs from all the XML documents (each label  $l$  has a text value, its URL and its base URL, the URL of the web page that it is present in)
- 3: Frequency Based Mining: Separate frequent labels,  $L$  (frequent ones are those that satisfy the support,  $Sup$ ) from the non-frequent ones.
- 4: Label Extraction Phase: Learn the attribute paths to each label in  $L$  and apply these paths to the document in which it is present and get the candidate labels and add them to  $L$
- 5: **for** each element  $l$  in  $L$  **do**
- 6:   Remove  $l$  from  $L$  if it does not have a valid url
- 7:   Get instances for  $l$  by invoking instance miner,  $l.Instances = InstanceMiner(l, S, Sup)$
- 8:   **if**  $l.Instances = \phi$  **then**
- 9:     Remove  $l$  from  $L$
- 10:   **end if**
- 11: **end for**
- 12: Grouping: Group the frequent labels  $FL$  according to their lexicographic similarity into concepts  $C$ .
- 13: Mine Relationships: Get the taxonomy of concepts in home pages by invoking relationship miner with the set of concepts  $C$ ,  $T = RelationshipMiner(C, S, Sup)$ .
- 14: **for** each concept  $c$  in the taxonomy  $T$  **do**
- 15:   Follow the links corresponding to the labels in this concept, fetch the Web pages and add them to  $S'$ .
- 16:   Invoke the taxonomy miner to get sub taxonomy for  $c$ ,  $T' = TaxonomyMiner(c, S', Sup)$
- 17:   Attach the sub taxonomy  $T'$  under the concept  $c$  in the taxonomy  $T$ .
- 18: **end for**
- 19: Return  $N$

corresponding to every concept  $c$ , find sub concepts and expand the taxonomy depth-wise and repeat the above phases (from Section 3.1 to 3.5) to identify sub concepts of  $c$ . For example, “Sports” is a concept in the taxonomy obtained from the previous phase. If we follow these links corresponding to “Sports” and repeat the above procedure we get taxonomy for “Sports”, that contains concepts like “Baseball”, “Tennis”, and “Horse Racing”. After this phase, we will have a complete taxonomy that represents the key concepts in the domain along with their taxonomical relationships.

The entire Taxonomy Mining algorithm is detailed in Algorithm 3.

**Experimental Results for Ontology Mining**

The mined ontology is evaluated in the same way as explained in the experi-

mental section for the semantic partition An Ideal Ontology is manually created and the parent/child relationships for both the ontologies are determined. The Precision for the mined ontology is .75 and the recall is .92

The taxonomy that we mined from the news domain is shown in Figure 9.

<b>Home</b>	<b>Sports</b>	<b>Tennis</b>
<b>Business</b>	<b>Soccer</b>	<b>2003 Wimbledon</b>
<b>World</b>	<b>Golf</b>	<b>Men</b>
<b>Entertainment</b>	<b>Tennis</b>	<b>Women</b>
<b>Sports</b>	<b>Horse Racing</b>	<b>TV Schedule</b>
<b>Weather</b>	<b>College Basketball</b>	
<b>Technology</b>	<b>Hockey</b>	
<b>Education</b>	<b>Olympics</b>	
<b>Health</b>	<b>Olympic Sports</b>	
<b>Privacy Policy</b>	<b>Pro Hockey</b>	

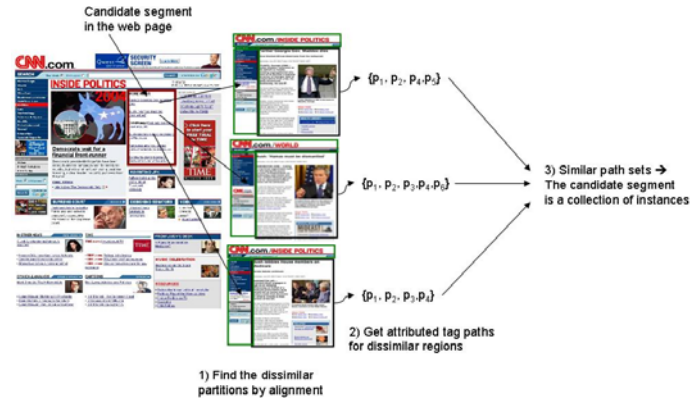
**Fig. 9.** A sample taxonomy obtained using the approach described for the News domain

## 4 Instance Extraction

This section describes our approach to extract instances from Web pages. Instances correspond to members of concepts. Our system can extract flat instances made up of list attributename-value pairs as well as complex, semi-structured instances. Our system is able to extract the labels whenever they are available. We first present our approach to extract appropriate instance segments from HTML documents. Later we describe our instance extraction algorithm and conclude with discussion on experimental results.

### 4.1 Instance Segments Extraction

The HTML documents usually contain many URLs and we describe an approach to extract the appropriate URLs that point to the instances in this section. To extract the correct instance URLs that point to instances, we adopt the following algorithms. Our system first partitions the Web page using the flat partitioning algorithm described in Section 2.1 and selects the segments of URLs that point to instances. A collection of links from a segment point to instances if the target pages contain similarly structured distinct instances. Our system first extracts the dissimilar regions from each of the collections of urls. This is done by flat partitioning each URL in the collection and aligning the segments based on the content similarity in the segments. Our system uses Levenstein Distance (aka edit distance) Measure to align the segments, by making use of Jaccard's Coefficient [15] (calculated as the ratio of the common words to the total number of words in the two segments). After aligning the segments, our system finds the dissimilar segments that are not aligned properly (the places where an insertion,



**Fig. 10.** Finding the segment that contains collection of instance URLs. Each candidate segment in the web page is examined. Each URL in the candidate segment is flat partitioned and the segments are aligned according to their content similarity. The green segments in the URL pages indicate the similar segments and they will be eliminated. The black segments which correspond to dissimilar segments are used to extract attributed tag paths. If the path sets are similar (have more common paths) then the candidate segment in the web page is chosen as the one with instances in it.

deletion or a replacement has occurred). Next it utilizes Hierarchical Partitioning algorithm described in 2.2 to convert these dissimilar regions into semi-structured XML documents. Later it extracts the attributed tag paths from the LCA (lowest common ancestor) of the leaf nodes in the segment to leaf nodes themselves. These path sets extracted from dissimilar regions (instance segments) represent the signature of the instances and our system chooses those instance segments for which these path sets are similar. This process is illustrated in Figure 10 for News Websites.

## 4.2 Instance Extraction for Labeled and Unlabeled Attributes

From the previous phase we have the hierarchically partitioned instance segments of the instance URLs. To extract instances from these segments, we use the tree miner algorithm described in Section ???. The tree miner algorithm provides us with the hierarchy among the frequent concepts among the instance segments. These frequent concepts correspond to the names of the attributes of the segments. For example, in hotels domain our system identified “Room Amenities”, “Hotel Services”, “Local Attractions”, etc. as frequent labels. These labels correspond to the attribute names and we extract the values for these attributes by finding the children of these labels in the hierarchically partitioned instance segments. We organize these labeled attributes along with their hierarchy in XML documents. Figure 11 shows one of the hotel pages that we used in our experiments and the attributes that we extracted from that hotel page.



The figure shows a screenshot of the Sheraton San Jose Hotel website. The website content includes a navigation bar, a header with the hotel name and address, a main image of the hotel, and several sections: Overview, Guest Rooms & Amenities, Hotel Services, Meeting & Event Facilities, Restaurants & Lounges, Local Area, and Driving Directions. To the right of the screenshot, there is a list of XML code snippets. Each snippet consists of a label (e.g., 'GUEST ROOMS & AMENITIES') followed by a series of 'node label' attributes that correspond to the content on the website. For example, under 'GUEST ROOMS & AMENITIES', there are labels for 'High Speed Internet Access in All Rooms (Charge)', 'Air-Conditioned Rooms', 'Alarm Clock', 'Balcony', and 'Cable Channels'. The XML code is enclosed in angle brackets and uses the label name as an attribute.

**Fig. 11.** A Sample Hotel Page that we used in our experiments and the attributes that we extracted from the page encoded as XML. The attribute name labels are capitalized in the XML file to distinguish them from attribute value labels

But some of the labels may not have any frequent label above or below them. For example in our experiments with News domain, we found that there are no frequent labels across News instance segments. Therefore the labels in these segments correspond to the values of attributes (such as the title of the article, body of the article, author of the article, city, etc.) whose names are not explicitly available in the instance segments. In this case we organize the attributes according to their paths, i.e., we maintain a table of attributes where columns correspond to the paths and rows correspond to each instance segment. Here we the attribute names are unknown and we plan to investigate techniques to mine the attributes by training classifiers for each path.

### Experimental Results for Instance Extraction

The precision and recall for the Instance mining for unlabelled attributes is .64 and .97 respectively. Similarly, for the hotel pages the precision and recall values are .80 and .91 respectively.

## References

1. Luke McDowell, Oren Etzioni, Steven D. Gribble, Alon Halevy, Henry Levy, William Pentney, Deepak Verma, and Stani Vlasheva. Evolving the semantic web with mangrove.
2. B. McBride. Four steps towards the widespread adoption of a semantic web. In *International Semantic Web Conference*, 2002.
3. S. Haustein and J. Pleumann. Is participation in the semantic web too difficult? In *International Semantic Web Conference*, 2002.
4. Mark Craven, Dan DiPasquo, Dayne Freitag, Andrew K. McCallum, Tom M. Mitchell, Kamal Nigam, and Seán Slattery. Learning to extract symbolic knowledge from the World Wide Web. In *Natl. Conf. on Artificial Intelligence*, pages 509–516, Madison, US, 1998. AAAI Press, Menlo Park, US.
5. Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *Proceedings of 27th International Conference on Very Large Data Bases*, pages 109–118, 2001.
6. A. Arasu and H. Garcia-Molina. Extracting structured data from web pages. In *ACM SIGMOD*, 2003.
7. William Cohen, Matthew Hurst, and Lee Jensen. A flexible learning system for wrapping tables and lists in html documents. In *Intl. World Wide Web Conf.*, 2002.
8. Y. Wang and J. Hu. A machine learning based approach for the table detection on the web. In *Intl. World Wide Web Conf.*, 2002.
9. Svetlozar Nestorov, Serge Abiteboul, and Rajeev Motwani. Extracting schema from semistructured data. In *ACM SIGMOD*, 1998.
10. Minos 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.
11. Yannis Papakonstantinou and Victor Vianu. Dtd inference for views of xml data. In *ACM PODS*, 2000.
12. Gao Cong, Lan Yi, Bing Liu, and Ke Wang. Discovering frequent substructures from hierarchical semi-structured data. In *Proceedings of the Second SIAM International Conference on Data Mining*, 2002.
13. M. Zaki. Efficiently mining frequent trees in a forest, 2002.
14. M.F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.
15. R.R. Korfhage. *Information Storage and Retrieval*. John Wiley Computer Publications, New York, 1999.