

Singapore Management University
Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems

School of Information Systems

6-2000

Re-engineering structures from web documents

Moh Chuang HUE


Ee Peng LIM

Singapore Management University, eplim@smu.edu.sg

Wee-Keong NG

DOI: <https://doi.org/10.1145/336597.336638>

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

 Part of the [Databases and Information Systems Commons](#), and the [Numerical Analysis and Scientific Computing Commons](#)

Citation

HUE, Moh Chuang; LIM, Ee Peng; and NG, Wee-Keong. Re-engineering structures from web documents. (2000). *5th ACM Conference on Digital Libraries (DL00)*. Research Collection School Of Information Systems.

Available at: https://ink.library.smu.edu.sg/sis_research/966

This Conference Proceeding Article is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email libIR@smu.edu.sg.

Re-engineering Structures from Web Documents

Chuang-Hue Moh, Ee-Peng Lim, Wee-Keong Ng

Center for Advanced Information Systems
School of Applied Science
Nanyang Technological University
Nanyang Avenue, Singapore 639798, SINGAPORE

ABSTRACT

To realise a wide range of applications (including digital libraries) on the Web, a more structured way of accessing the Web is required and such requirement can be facilitated by the use of XML standard. In this paper, we propose a general framework for reverse engineering (or re-engineering) the underlying structures i.e., the DTD from a collection of similarly structured XML documents when they share some common but unknown DTDs. The essential data structures and algorithms for the DTD generation have been developed and experiments on real Web collections have been conducted to demonstrate their feasibility. In addition, we also proposed a method of imposing a constraint on the repetitiveness on the elements in a DTD rule to further simplify the generated DTD without compromising their correctness.

KEYWORDS: Web information discovery, XML

1. INTRODUCTION

Motivation

The World-Wide Web (WWW) is one of the world's richest repositories of information, with a growth rate of 1 million Web pages per day and showing no signs of slowing down. Apart from the rapid growth rate and increasing popularity, the more significant impact to the WWW is the change in the way people are making use of the Web. Web documents are no longer simple hand-coded hypertext documents. Instead, they are becoming more interactive and a majority of them are generated automatically by underlying databases. As a result of the added complexity of Web applications today, we require a more structured way to access the Web.

The emergence of the Extensible Markup Language (XML) provides a partial solution to this problem. XML, in summary, is a structured format for data interchange over the Web. The main difference between HTML and XML is the use of tags - while HTML tags are primarily used to describe their way in which a data item is displayed on Web browsers,

XML tags describe the data itself. The effect of this subtle difference is that XML documents are self-describing and this facilitates the post-processing of Web documents, promising a complete change in the "behavior" of the Web in the near future. Although the structures for a set of structurally similar XML documents can be found in the Document Type Definition (DTD), there are a few problems associated with the use of DTDs as the "schema" for XML documents:

. For a given set of XML documents, the presence of the DTD is not mandatory i.e., we cannot be sure that the DTD will be available with certainty. Note that XML documents that follow the syntax of the XML specification are considered as *well-formed* XML documents. *Valid* XML documents are *well-formed* XML documents accompanied by a corresponding DTD.

- A majority of the Web documents on the WWW today are still in HTML format and do not have any DTD to describe their structure. Refer to [8] regarding the issues involved in "recycling" HTML documents into XML documents.
- The process of defining a DTD for a given set of XML documents is often a complicated and tedious task. In order to define the DTD for the documents, the user must have a clear idea how these documents are structured. To add to the complexity, DTDs do not follow the same syntax as the XML documents themselves. Hence we predict that quite a good number of Web pages on the WWW will not have any DTDs.

Objectives and Scope

The key objective of this project is to re-engineer the underlying structures of a given set of Web documents. We propose a general framework for **Structure Re-engineering** from Web documents and produce a DTD for each subset of similarly structured Web documents as a final result.

In the project, we do not attempt to solve all the problems pertaining to structure re-engineering. Instead, we propose a general framework for structure re-engineering of Web documents. We introduce a structural representation for a set of Web documents, in particular XML documents or semantically tagged HTML documents' that share a common structure but do not come with a DTD. We then develop the al-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and the copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Digital Libraries, San Antonio, TX.

Copyright 2000 ACM 1-581 13-231-X/00/0006...\$5.00

¹Such HTML documents are tagged with XML style user-defined tags to identify the semantic elements within the documents and their nesting relationships while retaining the HTML tags. Since XML style tagging is adopted, we may henceforth consider these HTML documents containing user-defined tags as XML documents.

gorithms for discovering the DTD from the structural representation. We have also conducted experiments on real-life examples of Web documents to demonstrate the discovery algorithms.

In this research, we focus on the textual and tag information within the Web documents. Other objects embedded in the documents such as multimedia data, hyperlinks, entity references and element attributes have not been considered so far but extensions of our algorithms to cater for such objects can be made in the future research.

Paper Organization

The rest of this paper is organized as follows. Section 2 surveys some of the work done that is related to our project. Section 3 covers the General Framework for the Structure Re-engineering process. The process of re-engineering an overall structure from these instance structures for the XML documents is discussed in Section 4. Section 5 introduces a set of heuristic rules to construct the DTD from the overall structure discovered. Section 6 provides us with some experimental results on two collections of XML documents. Section 7 presents the refinement methods used to overcome the inadequacies of the heuristic rules. Finally, Section 8 concludes the paper.

2. RELATED WORK

The automatic creation of DTD in the OCLC's GB-Engine [9] uses an approach that is fairly similar to ours. In the GB-Engine, an internal tree representation is built and converted into a grammar. The grammatical rules are then combined, generalized and reduced to produce a corresponding DTD. We see that the generation of an internal tree representation is similar to the Document Tree data structure that we propose. In their work, reduction rules like "identical bases", "off by one" and "redundant" were used to reduce the complexity of the DTDs generated. Nevertheless, the complexity of generated DTDs cannot be easily controlled by the users. In our proposal, we employ the *Longest Common Subsequence (LCS)* concept and also a user defined parameter **maximum repetition factor** to provide a more general and flexible method to reduce the complexity of the DTD generated.

In the Lore project [6], the OEM [7] was proposed to model the structures of semistructured data. The OEM model addresses the need of a more flexible data model for semistructured data like Web documents, as compared to conventional data models like object-oriented models. The main "drawback" of the OEM model is the missing ordering information about the elements in the schematic description of OEM model, also known as the DataGuides[5]. XML, on the other hand, does require the elements to conform to the ordering defined in the DTD.

Based on OEM, the *Roadmap Approach* [10, 11] to discover typical structures of documents was proposed. This approach however does not generate DTD for web documents. The typical structures generated also do not carry ordering information about elements in the documents. The *Northwestern*

Document Structure Extractor (NoDoSE) [1] presents an interactive approach to discovering Web document structures. In NoDoSE, the user is required to identify a few interesting regions in a single document and the program will attempt to identify other interesting regions in the document by decomposing the document in a top-down manner. The structure discovered from this document will then be used on documents of the same type of extract the relevant data. Although interesting, the need for extensive user intervention can prove to be tedious and non-trivial, especially when dealing with large and highly heterogeneous sources, like the WWW.

3. STRUCTURE RE-ENGINEERING FRAMEWORK

The proposed framework for the re-engineering of structures from Web documents consists of three phases as described below:

Phase I. Structure Extraction and Clustering:

The re-engineering process begins with extraction of instance structures from the Web documents and grouping them into clusters. The following steps must be carried out.

- **Semantic tagging of HTML documents:** XML documents are self-describing and hence the structure of individual documents can be derived from the XML tags. This however, is not the case for HTML documents - HTML tags only describe the way in which the data is to appear in Web browsers. To re-engineer the structure for a collection of XML or HTML documents, our proposed framework requires the structure of a few if not all member documents to be explicitly represented by inserting semantic tags into the HTML documents. For the HTML documents, we need to draw clues from the formatting information that is found in the HTML tags. For example, a bold short phrase may represent a section header. The process of marking the semantic structure explicit of a HTML document is termed *semantic tagging* and is very similar to the process described in [3].
- **Instance structure modeling:** The structure of an XML document or semantically tagged HTML document is inherently an *n-ary tree*, where the sub-elements are represented as child nodes of the tree node representing their parent element. We need to map the structures of each instance of the Web documents involved in the discovery process into an n-ary tree representation called the *Document Tree*. In the Document Tree representation, the hierarchical structure of the Web documents will be mapped into the hierarchical structure of the n-ary trees. For instance, if the element *area* is a sub-element of *geography*, then the node *area* will be a child node of *geography* in the Document Tree. Each node in the Document Tree is uniquely identified by a *Node-ID (NID)*. Note that the *NID* is also unique across all the Document Trees of the cluster of structurally similar documents. The other attributes in each node include:
 - *TagName*: String that corresponds to the name of the element represented by the node.
 - *AttList*: List of attribute name that represents the attributes of the corresponding XML element.
 - *PCData*: Boolean variable to determine whether the corresponding element in the DTD contains *Parsed Character Data (PCDATA)*

```

<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE document SYSTEM "country.dtd">
<document>
  <head>
    <title>Singapore</title>
  </head>
  <country>Singapore</country>
  <img href="figures/sn-t.gif" alt="[Country ...]"/>
  <img href="figures/sn-150.gif" alt="[Country ...]"/>
  <geography>
    <location> Southeastern ...</location>
    <coordinates>1 22 N, 103 48 E</coordinates>
    <map_ref>Southeast Asia</map_ref>
    <area>
      <total>647.5 sq km</total>
      <land>637.5 sq km</land>
      <water>10 sq km</water>
    </area>
    ...
    <maritime_claim>
      <exec_fishing_zone>within and beyond territorial...
      </exec_fishing_zone>
      <territorial_sea>3 nm</territorial_sea>
    </maritime_claim>
    ...
  </geography>
</document>

```

Figure 1: XML Document - Singapore

Example 1: Figures 1 and 2 show portions of the Web pages extracted from the “CIA World Factbook” on the facts and figures about Singapore and USA respectively. These documents were semantically tagged by hand to convert them to XML format. The Document Trees that represent the structures and semantics of these documents are shown in Figures 3(a) and (b). Note that each node in the Document Trees is uniquely identified by a NID attribute shown in the nodes of the Document Trees e.g., the node representing `<area>` `</area>` in the Document Tree of “sn.xml” has a NID of 9. The node attributes are not shown here for simplicity.

- **Structural clustering:** Web documents from a Web site may not always be structurally similar. For example, we would expect the content page of a collection of pages on diseases to be structured in a different manner compared to the pages describing medicines. Attempts to discover structures from a set of Web documents without considering the structural similarity or disparity between the documents will produce inaccurate results. As a result, we need to cluster documents by their structural similarity before we can begin to re-engineer the overall structure for each cluster of documents. Clustering can be applied on Document Trees based on the *Tree Pattern Matching* approach [2] or it can be coupled with the semantic tagging of HTML documents (since the semantic tagging process would already provide clues on the structure of each document instance).

Phase II. Structure Discovery:

The *Structure Discovery* phase takes a cluster of structurally similar Document Trees as its input and attempts to re-engineer an overall structure for each cluster. The definition of an *overall structure* is one that captures all the structural information from every Document Tree in the cluster. In this paper, we describe the use of a *Spanning Graph* representation to capture the overall structure.

```

<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE document SYSTEM "country.dtd">
<document>
  <head>
    <title>United States</title>
  </head>
  <country>United States</country>
  <img href="figures/us-t.gif" alt="[Country ...]"/>
  <img href="figures/us-150.gif" alt="[Country ...]"/>
  <geography>
    <location> North America, bordering ...
      Canada and Mexico</location>
    <coordinates>38 00 N, 97 00 W</coordinates>
    <map_ref>North America</map_ref>
    <area>
      <total>9,629,091 sq km</total>
      <land>9,158,960 sq km</land>
      <water>470,131 sq km</water>
      <note>includes only the 50 states ...
    </area>
    ...
    <maritime_claim>
      <contiguous_zone>12 nm</contiguous_zone>
      <continental_shelf>not ...</continental_shelf>
      <exec_economic_zone>200 nm</exec_economic_zone>
      <territorial_sea>12 nm</territorial_sea>
    </maritime_claim>
    ...
  </geography>
</document>

```

Figure 2: XML Document - USA

Phase III. DTD Construction:

In the final phase of *DTD Construction*, we will apply a set of *heuristic rules* to the Spanning Graph to generate a DTD. Ideally, all the documents in the cluster can be parsed with the generated DTD using any validating XML parser like the “XML for Java”.

In the subsequent sections, we will only focus on phases 2 and 3, i.e. Structure Discovery and DTD Construction. To facilitate the tasks involved, we first introduce Document Trees as a means to represent the structure of individual XML documents.

4. DISCOVERING THE STRUCTURES OF SIMILAR WEB DOCUMENTS

The Spanning Graph

The Spanning Graph data structure can be viewed as an ordered, *directed acyclic graph (DAG)* that encapsulates in it, all the structural information of every Document Tree that it spans i.e., all the structurally similar Document Trees in the cluster. The Spanning Graph is very similar to the Document Tree representation defined for instances of XML documents and is logically similar to the DTD of these XML documents. A DTD can be extracted from the Spanning Graph with some heuristic rules that we will discuss in the next section. The definition of the Spanning Graph is as follows.

- Each node in the graph represents an element and is uniquely identified by a unique GID. In addition, each node contains the following attributes:
 - TagName: The tag carried by the element represented by the node. Note every node in the Spanning Graph carries a unique TagName.
 - AttList: A list of attributes for the corresponding ele-

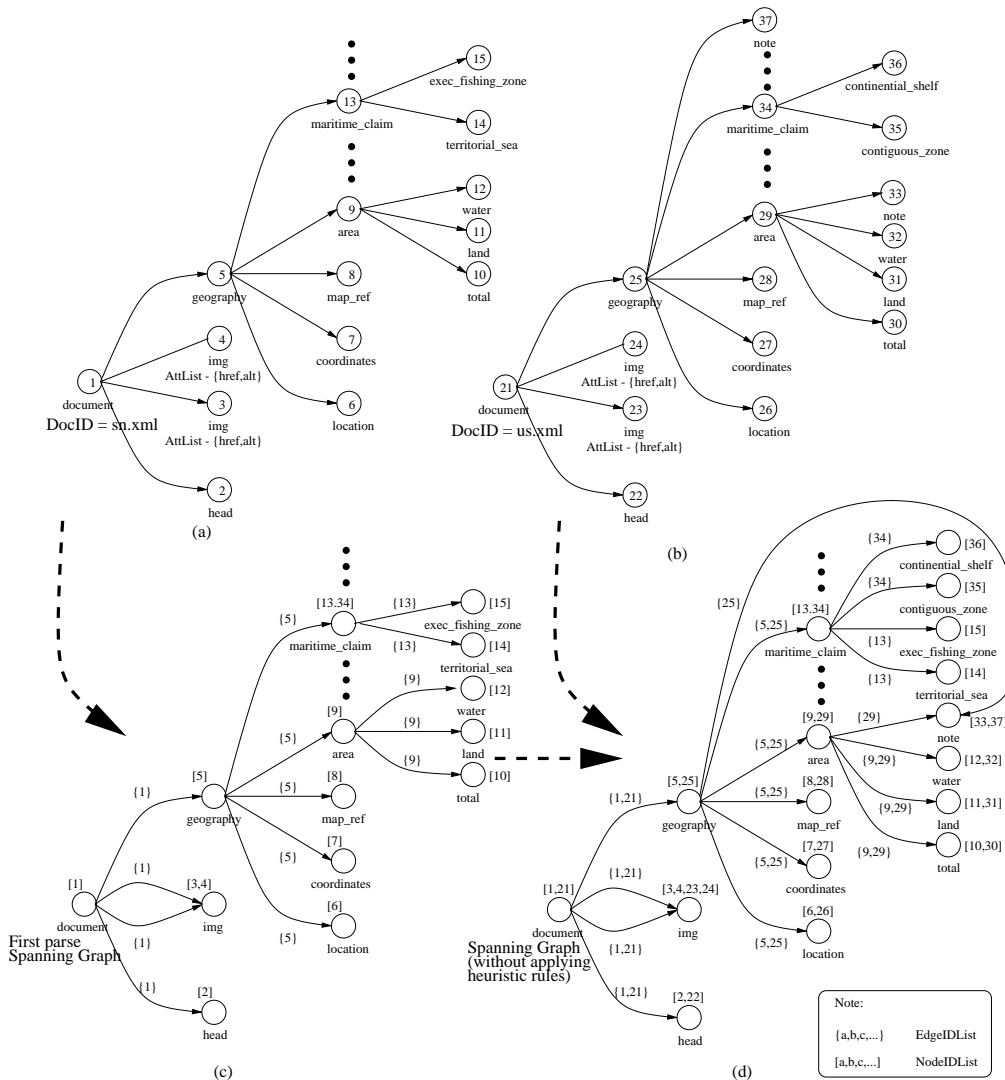


Figure 3: Document Tree Derived from XML Documents “sn.xml” and “us.xml” and the Spanning Graph

ment represented by the node. This is a union of all the AttList in the nodes of the Document Trees with matching TagNames as the TagName of the Spanning Graph node.
 – NodeIDList: A list of all the NID of the nodes in all the Document Trees with the same TagName as the Spanning Graph node.

- The edges in the Spanning Graph models the hierarchical relationships between elements.
- The left-right ordering of sibling nodes denotes the left-right ordering of sub-elements of a single parent element. For example, if the node `coordinates` is the left sibling of the node `area` in the Spanning Graph, then the same order can be found between the tag pairs `<coordinates>` `</coordinates>` and `<area>` `</area>` in every document tree where they exist.
- Every edge in the Spanning Graph is uniquely identified by an *Edge-ID* (EID). Furthermore, each edge is assigned an *EdgeList* that identifies the parent nodes in the Document Trees in which the parent-child relationship represented by

this Spanning Graph edge exists.

Example 2: The Spanning Graph for the document trees of “sn.xml” and “us.xml” is shown in Figure 3(d). The hierarchical relationships between the elements in the document instances are captured in the Spanning Graph as they are in the Document Trees. We also observe that each node in the Spanning Graph has a unique TagName field, which corresponds to the name of the element represented by the node. In addition, the ordering of the nodes in every Document Tree is preserved in the Spanning Graph. For example, we see that in the Document Tree “sn.xml”, the node `location` is a left-sibling node of `coordinates` under the parent node `geography`. This relationship is preserved in the Spanning Graph.

Example 3: As shown in Figure 3(d), each edge contains an `EdgeIDList` corresponding to the `NID` of the parent node in which the relationship represented by the edge exists. For instance, we see that the Spanning Graph edge `<document, geography>` has `EdgeIDList = {1, 21}`. The `NodeIDList` contains the `NID` of the nodes in the Document Trees that have the same `TagName` as the Spanning Graph node. The node `water` in the Spanning Graph has a `NodeIDList` of `[12, 32]`. The numbers in this list are identical to the `NIDs` of the nodes in the Document Trees “sn.xml” and “us.xml” with `TagName` of `water`.

Constructing The Spanning Graph To construct the Spanning Graph from a cluster of similarly structured Document Trees, we employ an incremental strategy in which one Document Tree is merged into the Spanning Graph at a time. The intermediate structure created when some but not all of the Document Trees had been merged into the Spanning Graph is called the *intermediate Spanning Graph*. The algorithm stops when all the Document Trees are merged into the Spanning Graph.

In the process of merging Document Trees into the Spanning Graph, we need to determine how nodes with the same tag name from the same or different Document Trees can be merged together while preserving the relative orderings of their child nodes. After two nodes of the same tag name are merged together, the problem of merging their sequences of child nodes is analogous to the problem of merging of two strings. There are often more than one possible merged string one can derive. In order to produce an amalgamated sequence of child nodes that has the minimum number of elements (which in turns leads to shorter DTD rules), we have adopted the *longest common subsequence* approach to merge the child nodes [4]. The algorithm for constructing the Spanning Graph using the longest common subsequence approach is shown in Figures 4 and 5.

Observation: Given two strings X and Y , we see that if we merge the two strings by combining elements in both X and Y that are in the LCS, we can obtain the shortest string Z , where

- $X \subseteq Z$ and $Y \subseteq Z$.
- The order of all the elements in X and Y are preserved.

For example, if $X = \{A, B, C, D, E\}$ and $Y = \{A, F, C, G, H\}$, then $Z = \{A, B, F, C, D, E, G, H\}$. We see that the above two properties hold.

The algorithm begins with the `SpanGraphMerge()` procedure that integrates a Document Tree into a Spanning Graph by calling the `LCSMerge` procedure with the roots of both the Tree and Graph as input. The recursive `LCSMerge()` procedure first determines the longest common subsequence between the first level child nodes of the roots of the Doc-

Procedure : SpanGraphMerge

Input : A document tree *DocTree* to be merged into a spanning graph *SpanGraph* (immediate).

Output : *SpanGraph* after merging.

```
LCSMerge(DocTree.root, SpanGraph.root);
return SpanGraph;
```

Figure 4: Procedure: Merging of a Document Tree into the Spanning Graph Based on Longest Common Subsequence

ument Tree and Spanning Graph, and merges the the common nodes by updating the `EdgeIDList` and `NodeIDList` of the respective edges and nodes in the Spanning Graph. For these common nodes, the `LCSMerge` procedure is called recursively to further merge their child nodes.

For each non-common first level child node in the Document Tree, we call the `InsertLeftSibling()` or `InsertRightSibling()` procedure to insert the child node into Spanning Graph at the appropriate places. The two procedures are not shown here due to space constraint. In the process of inserting the Document Tree node into the Spanning Graph, one must attempt to merge the Document Tree node with a Spanning Graph node that carries the same `TagName`². If such a Spanning Graph node exists, the `LCSMerge()` procedure is invoked again to merge the child nodes of the Document Tree node and Spanning Graph node.

The algorithm guarantees a minimum number of edges being created when merging the child nodes of matching Document Tree and Spanning Graph nodes i.e., nodes with the same `TagName`. This implies that a minimum number of edges will be created when merging the Document Tree and the Spanning Graph. However, this solution is sub-optimal when applied to merge a set of Document Trees into the Spanning Graph. Optimally, we should compare the sequences of child nodes for all the identical nodes in each Document Tree and find the LCS for these sequences. The merging should then be done based on this LCS. However, the resultant complexity of finding the LCS for several sequences of child nodes (instead of just two sequences) would be infeasibly expensive. Therefore, in this paper, we will adopt the sub-optimal solution to control the complexity of the algorithm.

²This is possible as the Spanning Graph node with the same `TagName` may not be a child node of G_S , or it may be child node of G_S not appearing in the longest common subsequence.

Procedure : LCSMerge

Input : Node T_D of a Document Tree T_{Doc} , and
Node G_S of a Spanning Graph G_{Span} where
 $G_S.TagName = T_D.TagName$.

$LCS_1, \dots, LCS_k \leftarrow \text{FindLCS}(G_S.ChildList,$
 $T_D.ChildList)$

for $i = 1$ to k **do**

Let N_t be the first child node of T_D where
 $N_t.TagName = LCS_i.TagName$

Let N_g be the first child node of G_S where
 $N_g.TagName = LCS_i.TagName$

foreach child node N_t^l of T_D that are
left-sibling nodes of N_t **do**

InsertLeftSibling(G_S, N_t^l, N_g)

$\langle G_S, N_g \rangle.EdgeIDList \leftarrow \langle G_S, N_g \rangle.EdgeIDList$
 $\cup T_D.NID$

$N_g.NodeIDList \leftarrow N_g.NodeIDList \cup N_t.NID$
LCSMerge(N_t, N_g)

foreach $N_t^r \leftarrow$ remaining child nodes of T_D **do**

InsertRightSibling($G_S, N_t^r, G_{Span}.RightMostChild$)

Figure 5: Procedure: Merging the Child Nodes of a node in the Document Tree with the Child Nodes of a node in a Spanning Graph

Example 4: Figures 3(c) and (d) show the process of constructing the Spanning Graph from two Document Trees of the documents “us.xml” and “sn.xml”. The first step in constructing the Spanning Graph is to merge the Document Tree of “sn.xml” with the initial Spanning Graph which is a graph with only a root node. We obtain the first intermediate Spanning Graph, G_{Span}^1 . Next, the Document Tree of another document “us.xml” is merged into G_{Span}^1 to form the second intermediate Spanning Graph G_{Span}^2 . Since there are no more Document Trees to merge, G_{Span}^2 becomes the final Spanning Graph. Note that this approach can be extended to merge an arbitrary number of Document Trees.

5. FINAL CONSTRUCTION OF DTD

In this section, we outline the process for the discovery of a DTD from a Spanning Graph. The process of discovering the DTD involves a set of *heuristic rules* to be applied on the Spanning Graph to suggest the DTD. Note that the heuristic rules are applied to the edges of the Spanning Graph to determine additional information such as whether a sub-element is an optional or mandatory element of the parent element. The rules are not applied to the nodes although it may be intuitive to do so. For example, in the Spanning Graph shown in Figure 6, the element `note` represented by a node of the same

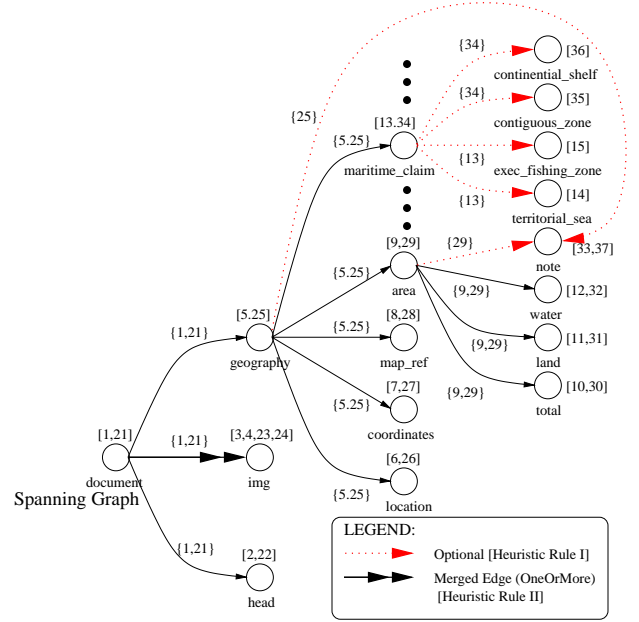


Figure 6: Example - Applying the Heuristic Rules to the Spanning Graph

name is both a sub-element of the elements `geography` and `area` in the document `sn.xml`. However, we see that `note` is an optional sub-element of `area` but a mandatory one of `geography`. If we apply the heuristic rules to the nodes in this case, confusion would arise as to whether `note` is optional or mandatory.

Heuristic Rules

The heuristic rules are:

Rule I (Define Optionality):

To define optionality simply means to determine for each element N_p , how frequent does the relationship $N_c = \text{Child}(N_p)$ occurs for every child element N_c of N_p . To examine the frequency of occurrence of each element, we will make use of the `NodeIDList` attribute of the node N_p and compared them to the `EdgeIDList` attribute of the edge $\langle N_p, N_c \rangle$ to determine the relative frequency of occurrence of the relationship $N_c = \text{Child}(N_p)$ with respect to the frequency of occurrence of the node N_p .

For each child node N_c of a parent node N_p ,

- If $\langle N_p, N_c \rangle.EdgeIDList \equiv N_p.NodeIDList$, then the element represented by N_c is a mandatory sub-element of that represented by N_p .
- Else, we conclude that the element represented by N_c is an optional sub-element of the element represented by N_p .

A new attribute `OptTag` is assigned to each edge to capture the optionality of the relationship.

Example 5: In the Spanning Graph example in Figure 3, edge $\langle \text{area}, \text{note} \rangle$ is marked as `ZeroOrOne(?)` i.e., optional since $\langle \text{area}, \text{note} \rangle.\text{EdgeIDList} (= \{29\}) \subset$ of the `NodeIDList` of the node with `TagName` of `area` ($= \{9, 29\}$). This implies that the child node `note` of `area` does not always occur when the node `area` appears and hence is considered an optional child element. Consequently, the element `note` will be declared as an optional child element of `area` in the DTD i.e., `<!ELEMENT area (total, land, water, note?)>`. The same applies to the edge $\langle \text{maritime_claim}, \text{exec_fishing_zone} \rangle$ and others.

Rule II (Merge Repeat):

For each pair distinct adjacent sibling edges $E_i (= \langle N_p, N_c \rangle)$ and $E_j (= \langle N_p, N_{c+1} \rangle)$ in the Spanning Graph, if $N_c = N_{c+1}$ (i.e., E_i and E_j are adjacent parallel edges), we will collapse the two edges into a single edge E_k where $E_k.\text{EdgeIDList} \equiv E_i.\text{EdgeIDList} \cup E_j.\text{EdgeIDList}$.

- If $E_k.\text{EdgeIDList} \equiv N_p.\text{NodeIDList}$ then we consider N_c as an `OneOrMore` sub-element of N_p .
 - Else N_c will be said to be an `ZeroOrMore` sub-element of N_p .
-

Example 6: In the Spanning Graph in Figure 3, there are two adjacent edges $\langle \text{document}, \text{img} \rangle$. As defined in Heuristic Rule 1, we can merge these two edges into a single edge the resultant edge's `EdgeIDList` will be the union of the `EdgeIDList` of the two edges. Furthermore, since the result edge $\langle \text{document}, \text{img} \rangle$ has `EdgeIDList` of $\{1, 21\}$, which is equivalent to the `NodeIDList` of the parent node (root node) of this relationship, $\langle \text{document}, \text{img} \rangle.\text{OptTag} = \text{OneOrMore}$. and `img` will be considered as an `OneOrMore` (+) sub-element of `document`. The DTD generated will hence look like `<!ELEMENT document (..., img+, ...)>`

Rule III (Define Group):

In this rule, we attempt to identify *repeating groups* in the sequence of child edges of a parent node. The condition that must be met before a subsequence of child edges can be defined as a group is that all the edges in the group must have identical `EdgeIDList`. Hence, we use the `EdgeIDList` to group adjacent edges together i.e., group elements together if they have identical `EdgeIDList`.

After partitioning a sequence of “child” edges of a parent node according to the `EdgeIDList`, we then proceed to find adjacent groups that have identical elements i.e., the edges in both groups points to the same sequence of child nodes (in order). We first begin by examining the first two partitions to be merged and attempt to find in each of the partition, a sequence of edges pointing to identical elements. We will

use an example to illustrate the process of finding the common elements between two partitions and combining them to form a repeating group. For example, if we are given two partitions of elements $\{a, b, c, d, e\}$ and $\{b, c, d, e, f\}$, we will attempt to find two identical sequence of elements that begins with the first element of the second partition i.e., `b` and ends with the last element of the first partition i.e., `e`. Here, we see that the sequence $\{b, c, d, e\}$ fits this description and hence we merge the two partition into $\{a, b, c, d, e, f\}$ and the corresponding DTD fragment that will be generated will be of the form $\{a, (b, c, d, e), f\}$.

Before we conclude the grouping of elements, we must also determine the optionality of the entire group of elements as well as each individual element that was merged. To determine the `OptTag` of the group of elements, we examine the union of the `EdgeIDList` of both groups of elements:

- If the unioned `EdgeIDList` of the group of elements is equivalent to the `NodeIDList` of the parent node, then the group of elements is considered as `OneOrMore`.
- Else if the unioned `EdgeIDList` is a subset of the `NodeIDList` of the parent node, then the group of elements is considered as `ZeroOrMore`.

For each individual elements in the group, the `OptTag` is `Required`. The exception is when one or both the elements to be merged has an `OptTag` of `ZeroOrMore` or `OneOrMore` (from the `MergeRepeat` rule). If only one of the element is `OneOrMore` or `ZeroOrMore` (repeating), then the merged element will be `OneOrMore` or `ZeroOrMore` respectively. In the case when both elements to be merged are repeating, then we will use the following:

- If both the elements have the same `OptTag` value, then the merged element will inherit that `OptTag` value.
- Else the merged element will have an `OptTag` value of `ZeroOrMore`.

Note that each edges can only be a member a single group. Each group is assigned a `GroupID` to enable us to determine the groups of elements in the DTD that are to be generated.

Example 7: As the documents on the “CIA World Factbook” do not provide any examples for applying Heuristic Rule III, we shall provide an independent example in Figure 7. In this sub-graph structure rooted at node `A`, we can identify two identical subsequences of edges $\{B, C, D\}$. As every edge of each of the subsequence has the same `EdgeIDList`, they can be considered as a group. In the example, since the two groups are identical and adjacent, we will merge them together. The resultant `EdgeIDList` of the edges in the new group will be the union of the `EdgeIDList` of the edges in the two component groups i.e., $\{1, 2, 3\}$. Since the `NodeIDList` of node `A` is $\{1, 2, 3\}$, the group will be considered as an `OneOrMore` group. The corresponding DTD segment will be `<!ELEMENT A ((B, C, D)+, B)>`.

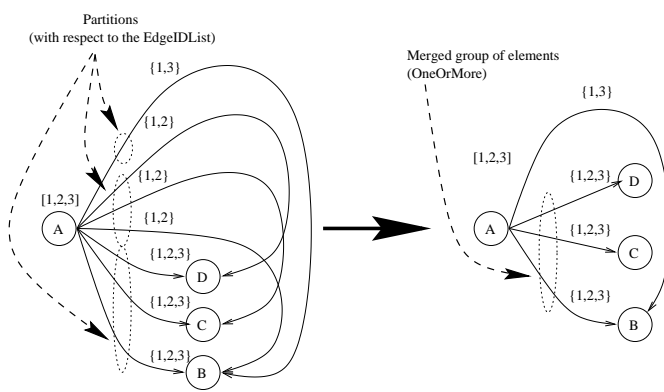


Figure 7: Example - Grouping of Elements

6. EXPERIMENTAL RESULTS

For the purpose of empirical verification, we have implemented and tested the our proposed on the collection of XML documents for the play “Life of Henry the Fifth” by William Shakespeare (converted into XML by Jon Bosak). The DTD (known as “play.dtd”) used by the collection is shown in Figure 8. Our primary aim here is to verify that the DTD generated is correct. Furthermore, by comparing the generated DTD and the original one, we would like to identify the pitfalls of our algorithms and to suggest some improvement.

The DTD generated by our algorithms is shown in Figure 9. Using a validating XML parser, we verified that the generated DTD can be used to parse the XML collection. Most of the simple element declaration in the generated DTD are quite similar to the original ones. Nevertheless, we observe that the generated DTD element declaration contains too much repetitive information. For example, the elements PGROUP and PERSONA appeared multiple times as child elements of the element PERSONAE in the generated DTD in Figure 9. This makes the DTD too complicated and lengthy to be useful and hinders its readability. In the original DTD in Figure 8 that the declaration for PERSONAE is simply `<!ELEMENT PERSONAE (TITLE, (PERSONA | PGROUP)+)>`. This suggests that our DTD generation method can be further improved to derive more readable and simpler DTDs. In the next Section, we present some further refinement to our DTD generation method to achieve the goal.

7. FURTHER REFINEMENT

In order to overcome overly complex element declaration generated by our DTD generation method, we have adopted an interactive solution that allows a user to specify how much the DTD should be simplified or *relaxed*. The relaxation of DTD is achieved by reducing the ordering constraint on the sub-elements in the element declaration so as to group approximately similar groups of elements together, hence reducing complexity.

One simple example is the sequence of sub-elements $\{A, B, C, D, A, D, A, C\}$, which can be declared in the DTD as `<!ELEMENT Root (A, B?, C,?, D?)>` or `<!ELEMENT Root (A|B|C|D)+>` instead of `<!ELEMENT Root (A,B,C,D,A, D,B,C)>`.

```

<!-- DTD for Shakespeare J. Bosak ... -->
<!-- Revised for case sensitivity 1997.09.10 -->
<!-- Revised for XML 1.0 conformity 1998.01.27 ... -->

<!ENTITY amp "&#38;">
<!ELEMENT PLAY (TITLE, FM, PERSONAE, SCNDESCR, PLAYSUBT,
INDUCT?, PROLOGUE?, ACT+, EPILOGUE?)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT FM (P+)>
<!ELEMENT P (#PCDATA)>
<!ELEMENT PERSONAE (TITLE, (PERSONA | PGROUP)+)>
<!ELEMENT PGROUP (PERSONA+, GRPDESCR)>
<!ELEMENT PERSONA (#PCDATA)>
<!ELEMENT GRPDESCR (#PCDATA)>
<!ELEMENT SCNDESCR (#PCDATA)>
<!ELEMENT PLAYSUBT (#PCDATA)>
<!ELEMENT INDUCT (TITLE, SUBTITLE*, (SCENE+|(SPEECH|
STAGEDIR|SUBHEAD)+))>
<!ELEMENT ACT (TITLE, SUBTITLE*, PROLOGUE?, SCENE+,
EPILOGUE?)>
<!ELEMENT SCENE (TITLE, SUBTITLE*, (SPEECH|STAGEDIR|
SUBHEAD)+)>
<!ELEMENT PROLOGUE (TITLE, SUBTITLE*, (STAGEDIR|SPEECH)+)>
<!ELEMENT EPILOGUE (TITLE, SUBTITLE*, (STAGEDIR|SPEECH)+)>
<!ELEMENT SPEECH (SPEAKER+, (LINE|STAGEDIR|SUBHEAD)+)>
<!ELEMENT SPEAKER (#PCDATA)>
<!ELEMENT LINE (#PCDATA | STAGEDIR)*>
<!ELEMENT STAGEDIR (#PCDATA)>
<!ELEMENT SUBTITLE (#PCDATA)>
<!ELEMENT SUBHEAD (#PCDATA)>

```

Figure 8: Original DTD for “Life of Henry the Fifth”

```

<!-- DTD-Miner Version 1.4 -->
<!-- CAIS, NTU, Singapore -->
<!-- Copyright 1999 Moh Chuang Hue -->
<!-- Parameters specified: -->
<!DOCTYPE PLAY [
<!ELEMENT PLAY ( TITLE, FM, PERSONAE, SCNDESCR,
PLAYSUBT, INDUCT?, PROLOGUE?, ACT+ )>
<!ELEMENT TITLE ( #PCDATA )>
<!ELEMENT FM ( P+ )>
<!ELEMENT P ( #PCDATA )>
<!ELEMENT PERSONAE ( TITLE, PERSONA?, PGROUP?,
PERSONA?, PGROUP?, PERSONA?, PGROUP?, PERSONA*,
PGROUP*, PERSONA*, PGROUP*, PERSONA+, PGROUP?,
PERSONA*, PGROUP?, PERSONA*, PGROUP?, PERSONA*,
PGROUP?, PERSONA?, PGROUP*, PERSONA*, PGROUP?,
PERSONA* )>
<!ELEMENT PERSONA ( #PCDATA )>
<!ELEMENT PGROUP ( PERSONA+, GRPDESCR )>
<!ELEMENT GRPDESCR ( #PCDATA )>
<!ELEMENT SCNDESCR ( #PCDATA )>
<!ELEMENT PLAYSUBT ( #PCDATA )>
<!ELEMENT INDUCT ( TITLE, STAGEDIR*, SPEECH?,
STAGEDIR?, SCENE* )>
<!ELEMENT STAGEDIR ( #PCDATA )>
<!ELEMENT SPEECH ( SPEAKER+, LINE*, STAGEDIR?, LINE?,
STAGEDIR?, LINE?, STAGEDIR?, ... (not shown), LINE* )>
<!ELEMENT SPEAKER ( #PCDATA )>
<!ELEMENT LINE ( #PCDATA | STAGEDIR )*>
<!ELEMENT SUBHEAD ( #PCDATA )>
<!ELEMENT SCENE ( TITLE, STAGEDIR?, SUBHEAD?, SPEECH*,
..., SPEECH? )>
<!ELEMENT PROLOGUE (TITLE, STAGEDIR*, SPEECH, STAGEDIR?)>
<!ELEMENT ACT ( TITLE, PROLOGUE?, SCENE+, EPILOGUE? )>
<!ELEMENT EPILOGUE ( TITLE, STAGEDIR?, SUBTITLE?, SPEECH,
STAGEDIR? )>
<!ELEMENT SUBTITLE ( #PCDATA )>
]>

```

Figure 9: DTD Generated for “Life of Henry the Fifth”

In designing the interactive process, we ensured that the amount of user intervention required is minimized so as to make the generation of DTD a less tedious task for the user.

Consider the sequence of sub-elements $\{A, B, C, D, A, D, A, C\}$. The **Repetition Factor** for each sub-element is defined as the number of times the same sub-element (of the same `TagName`) appears in the sequence. In this example, the repetition factor of A is 3 and that of B is 1. We define **Maximum Repetition Factor (MAXREP)** to be the maximum repetition factor one can find in an element declaration. To reduce the complexity of the generated DTDs, we allow users to impose constraints over the maximum repetition factor. If the repetition factor of a sub-element in a DTD element declaration exceeds **MAXREP**, the sequence of sub-elements in the declaration should be merged to reduce the repetition factor. For example, for a **MAXREP** of 1, the sequence in the example can be merged to form $\{A, B?, C?, D?\}$. The merging strategy that we used is an incremental strategy which merges two subsequences starting from the first extraneous sub-element e.g., we merge the two sub-sequences $\{A, B, C, D\}$ and $\{A, D\}$ to get the sequence $\{A, B?, C?, D\}$. Then we further merge this sequence with $\{A, C\}$ to get $\{A, B?, C?, D?\}$.

We employ the concept of LCS here to merge subsequences of elements together to achieve our goal of relaxing the DTD generated. Due to space constraint, we will only consider in this paper only cases where there are no groups in the DTD. Starting from the first edge in the sequence of “child edges”, we will partition the entire sequence of edges into smaller sequences. Each of these smaller sequences will start with edges that point to a sub-element with repetition factor greater than **MAXREP**. For example, if we have the sequence $\{A, C, D, E, A, D, F, A, B, E\}$ and A has a repetition factor larger than **MAXREP** (say = 2), we will then have the partitions $\{A, C, D, E\}$, $\{A, D, F\}$ and $\{A, B, E\}$. These partitions are to be merged to reduce the repetition factor of A.

To merge the partitions, we will find the LCS between the two smaller sequences and proceed to merge the elements, using those elements in the LCS as pivotal elements. In the example given above, we see that the LCS between the first two partitions of elements is $\{A, D\}$ and the merged sequence will be $\{A, C, D, E, F\}$. The merging of partitions proceeds incrementally until **MAXREP** is satisfied. Here, if the **MAXREP** is 2, then we will stop since the number of partitions is 2 and the final sequence of child elements is $\{A, C, D, E, F, A, B, C\}$. However, if the **MAXREP** is 1, then we will merge $\{A, C, D, E, F\}$ with $\{A, B, C\}$ to get the final sequence of $\{A, B, C, D, E, F\}$.

Finally, we also need to determine the optionality of the entire group of merged elements and also that of each individual elements that we have merged. For the group of merged elements, if the union of all the `EdgeIDList` of the elements in the group is equal to the `NodeIDList` of the parent node, then we consider the group of elements as `OneOrMore`. Otherwise, if the `EdgeIDList` of the elements in the group is a subset of the `NodeIDList` of the parent node, then the

group is considered as `ZeroOrMore`. We use another variable `RelaxOptTag` to represent the optionality of each individual element within the group of elements. This `RelaxOptTag` supercedes the `OptTag` in determining the optionality of elements in the final DTD generated and is only applicable to elements that are involved in the relaxation process. The rule to determine the `RelaxOptTag` is as follows:

- If the element in the group is not in the LCS, then its `RelaxOptTag` will be `Optional`.
- If the element appears in the LCS, then we will need to merge it with another identical element in the second partition. The `RelaxOptTag` of the merged element will be determined as follows:
 - If one of the element to be merge has `RelaxOptTag` = `Optional`, then the `RelaxOptTag` of the merged element will be `Optional`.
 - Else if one or both the element has an `OptTag` value of `Optional` or `ZeroOrMore`, then the merged element’s `RelaxOptTag` value will be `Optional`.
 - Else the merge element’s `RelaxOptTag` value will be `Required`.

After applying the method with the first element as the starting element, we then proceed to apply the method again with the second element as the starting element for the partitions and so on. The relaxation stops when **MAXREP** is satisfied.

Example 8: Figure 10 shows the result of generating the DTD with a **MAXREP** of 1. To illustrate the merging of the elements in the DTD, we will look at the declaration of `INDUCT`. In the DTD generated originally, the definition of this element is `<!ELEMENT INDUCT (TITLE, STAGEDIR*, SPEECH?, STAGEDIR?, SCENE*)>`. We first begin by partitioning the sequence of child elements with the first element as the start element and we obtain only a single partition since `TITLE` only appears once. No merging is required in this case. Next, we proceed to partition the sequence with `STAGEDIR` as the starting element and we obtain two partitions, namely $\{STAGEDIR, SPEECH\}$ and $\{STAGEDIR, SCENE\}$. Since the number of partitions exceeds the **MAXREP**, we will need to merge the two partitions. Here, the LCS that we derive from the two partitions is $\{STAGEDIR\}$ and the sequence $\{TITLE, (STAGEDIR, SPEECH, SCENE)\}$ is obtained by merging the partitions. The optionality of the entire group of merged element (as determined by the program automatically) is `OneOrMore`. The generated DTD fragment is `<!ELEMENT INDUCT(TITLE, (STAGEDIR?, SPEECH?, SCENE?)+)>`. We see that the elements `SPEECH` and `SCENE` are `Optional` since they are not found in the LCS. The element `STAGEDIR` is also `Optional` although it is found in the LCS since the original elements (before merging) are also `optional` (`ZeroOrMore` and `Optional` respectively).

With a **MAXREP** of 1, a new DTD was derived as shown in Figure 10. The complexity of the generated DTD was successfully reduced and is closer to the original DTD.

```

<!-- DTD-Miner Version 1.4 -->
<!-- CAIS, NTU, Singapore -->
<!-- Copyright 1999 Moh Chuang Hue -->
<!-- Parameters specified: [ MaxRep = 1 ] -->
<!DOCTYPE PLAY [
  <!ELEMENT PLAY ( TITLE,FM,PERSONAE,SCNDESCR,PLAYSUBT,
    INDUCT?, PROLOGUE?, ACT+ ) >
  <!ELEMENT TITLE ( #PCDATA ) >
  <!ELEMENT FM ( P+ ) >
  <!ELEMENT P ( #PCDATA ) >
  <!ELEMENT PERSONAE ( TITLE, ( PERSONA?, PGROUP? )+ ) >
  <!ELEMENT PERSONA ( #PCDATA ) >
  <!ELEMENT PGROUP ( PERSONA+, GRPDESCR ) >
  <!ELEMENT GRPDESCR ( #PCDATA ) >
  <!ELEMENT SCNDESCR ( #PCDATA ) >
  <!ELEMENT PLAYSUBT ( #PCDATA ) >
  <!ELEMENT INDUCT ( TITLE, ( STAGEDIR?, SPEECH?, SCENE? )+ ) >
  <!ELEMENT STAGEDIR ( #PCDATA ) >
  <!ELEMENT SPEECH ( SPEAKER+, ( LINE?, STAGEDIR?, SUBHEAD? )+ ) >
  <!ELEMENT SPEAKER ( #PCDATA ) >
  <!ELEMENT LINE ( #PCDATA | STAGEDIR ) * >
  <!ELEMENT SUBHEAD ( #PCDATA ) >
  <!ELEMENT SCENE ( TITLE, ( STAGEDIR?, SPEECH?, SUBHEAD? )+ ) >
  <!ELEMENT PROLOGUE ( TITLE, ( STAGEDIR?, SPEECH? )+ ) >
  <!ELEMENT ACT ( TITLE, PROLOGUE?, SCENE+, EPILOGUE? ) >
  <!ELEMENT EPILOGUE ( TITLE, ( STAGEDIR?, SUBTITLE?,
    SPEECH? )+ ) >
  <!ELEMENT SUBTITLE ( #PCDATA ) >
]>

```

Figure 10: DTD Generated for “Life of Henry the Fifth” with MAXREP of 1

8. CONCLUSIONS

In this paper, the concept of re-engineering structures from Web documents has been introduced. Based on a structure re-engineering framework, we have developed some algorithm to construct a Spanning Graph that describes the structures of a set of similarly structured XML documents. We further proposed to generate the DTD for these XML document using a set of heuristic rules. For demonstration purposes, we have implemented our proposed technique into a prototype system known as *DTDMiner*. The Web interface for the system can be found at “<http://www.cais.ntu.edu.sg:8000/~chmoh/dtd-miner/>”. The system allows the user to supply some XML files and generates a DTD for them. It also supports relaxation of the generated DTDs.

As part of our future research, we plan to extend the re-engineering techniques in the following directions:

- **Discovering of attributes and attribute types:** The way that we have handled attributes so far is to simply assume that all the attributes are mandatory and of type CDATA. Attributes however, can be of various data types and may not always be required in the XML standard. As a result, we need to explore into more sophisticated ways of handling attributes to produce more accurate DTDs. Note that attributes can prove to be important to the structures of XML documents e.g., the XLink standard utilizes attributes to define the hyperlinks between XML documents.
- **Discovering inter-document structures:** The framework we have proposed is primarily used to discover the structures within Web documents i.e., intra-document structures. We see that such structures are not the only category of structures that can exist in Web documents. The hyperlinks that exist in almost all Web documents present an inter-document

structure (e.g., Web-site structure). Used in conjunction with the DTD discovered, the inter-document structures can provide a useful road-map to user query formulation.

REFERENCES

1. B. AdelBerg. NoDoSE - A Tool for Semi-Automatically Extracting Semi-Structured Data from Text Documents. In *ACM SIGMOD International Conference on Management of Data*, pages 283–294, 1998.
2. A. Apostolico and Z. Galil. *Pattern Matching Algorithms*. Oxford University Press, 1997.
3. N. Ashish and C. Knoblock. Wrapper Generation for Semi-structured Internet Sources. *ACM SIGMOD Record*, 26(4):8–15, 1997.
4. H.T. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1997.
5. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semi-Structured Databases. In *The 23rd VLDB Conference*, pages 436–445, 1997.
6. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *ACM SIGMOD Record*, 26(3):54–66, 1997.
7. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *11th International Conference on Data Engineering*, pages 251–260, 1995.
8. A. Sahuguet and F. Azavant. Web Ecology: Recycling HTML pages as XML documents using W4F. In *ACM Workshop on the Web and Database (WebDB)*, pages 31–36, 1999.
9. K. Shafer. Creating DTDs via the GB-Engine and Fred. In *SGML '95 Conference*, 1995.
10. K. Wang and H. Liu. Schema Discovery for Semistructured Data. In *3rd International Conference on Knowledge Discovery and Data Mining*, pages 271–274. AAAI Press, 1997.
11. K. Wang and H. Liu. Discovering Typical Structures of Documents: A Road Map Approach. In *21st ACM SIGIR Conference*, pages 146–154, 1998.