Master Thesis

Bootstrapping Ontologies from the Web

by

Jun Cai

University of the Saarland
Faculty 6 - Natural Sciences and Technology I
Dept. 6.2 - Computer Science

Max-Planck-Institute for Informatik
Databases and Information Systems Group
Prof. Dr. Gerhard Weikum
Advisors: Dr. Ralf Schenkel and Dipl.-Inf Jens Graupman

May 2004
Bootstrapping Ontologies from the Web

Abstract

Instance Extraction (Web Information Extraction) is crucial to many application area, such as ontology mapping, bootstrapping Semantic Web etc. Observation shows that instances are normally backed by underlying relational databases in almost all Web sites. Based on this observation, two approaches for Instance Extraction are illustrated in this thesis. The first approach is to extract instances from HTML tables. In this approach, machine learning techniques are used to detect genuine tables and several heuristics are applied to detect table headers. As more and more Web sites encode fetched database records into complex template engines, instances information in these Web sites would not be listed as they are in relational database—in rigid table format. So another more reliable approach based on the similarity of spatial locality of instances’ items is introduced. In this approach, an XPath clustering algorithm is used to generalize text item with similar path structure and apply a suffix tree algorithm to detect repeated sequence patterns among generalized path structure. The goal of this thesis is to bootstrap instances-rich ontologies from Web documents. API and GUI are provided to allow users navigate in the ontologies.
Acknowledgments

Starting at the beginning, I would like to thank Prof. Dr. Gerhard Weikum for introducing me to the exciting interdisciplinary research in Information Retrieval and Data Mining. He gave me the opportunity to be a part of a very active Databases and Information Systems Group.

I have to thank especially my supervisor Dr. Ralf Schenkel, Jens Graupman, who spend a lot of their valuable time by correcting this thesis at various stages of its development, with an amount of patience that I still find incredible. I am more than grateful for the constructive criticism and helpful remarks they gave me. Needless to say, all remaining errors are my own. Martin Theobald kindly discussed with me and gave me a lot of suggestion to my work for many times.

Finally, I always had and have the support of my family. Without them, everything is not possible.

JUN CAI

University of the Saarland
May 2004
## Contents

Abstract

Acknowledgments

List of Tables

List of Figures

Chapter 1 Introduction and Motivation

1.1 Challenge from the Internet

1.2 Ontology

1.3 Application scenarios

1.3.1 From Keyword-based to Concept-based Search

1.3.2 Automated form filling

1.3.3 From Web to Semantic Web

1.3.4 Ontology matching

1.4 Overview of contribution

Chapter 2 Extracting instances for ontology from HTML Table

2.1 Introduction

2.2 Contribution and related work

2.3 Table related elements and attributes

2.4 Structure analysis

2.5 Table Detection

2.5.1 Genuine table classification with decision tree

2.6 Table header detection
List of Tables

2.1 Crawled HTML pages for different domains. The numbers of genuine tables are detected by the system. . . . . . . . . . . . . . 31
2.2 Performance measure for each table understanding module. . . 31
2.3 Occurrences of properties of Car and Job domains . . . . . . . 32

3.1 Sample XPaths for given Text node. . . . . . . . . . . . . . . . 44
3.2 Properties of XPath Cluster. . . . . . . . . . . . . . . . . . . . . 45
3.3 Performance of SLIE system. . . . . . . . . . . . . . . . . . . . . 59
# List of Figures

1.1 Two ontologies about computer science department in different university with instances information. (taken from [17]) .... 7

2.1 A table describing cars(yahoo.com) ......................... 9
2.2 Part of the ontology created from the HTML table in Figure 2.1. 10
2.3 Rendering of *rowspan* and *colspan* examples ............ 14
2.4 Rendered table in a Web browser and its HTML source code .. 16
2.5 Corresponding pseudo-table of table in Figure 2.4 .......... 17
2.6 Algorithms: *GeneratePseudoTable* .......................... 18
2.7 Function: *AdjustColumnIndex* ............................... 20
2.8 Compute cell length consistency for each cell: Equation 2.7 .. 24
2.9 Sample HTML table header ....................................... 25
2.10 Merge example for the table in Figure 2.4 .................. 26
2.11 Table understanding process workflow ....................... 27
2.12 Generated decision tree of the test dataset ............... 28
2.13 Analysis result for sample dataset(training data) ......... 29

3.1 A sample Amazon search result page ......................... 34
3.2 DOM tree representation of Amazon page ..................... 37
3.3 Type System ..................................................... 40
3.4 Structure Partition Tree of Amazon page ..................... 42
3.5 The UML diagram for the implementation of the XPath clustering algorithm ..................................................... 46
3.6 Procedure: *ShouldGroup* ....................................... 47
3.7 Procedure: *CreateClusters* ..................................... 48
3.8 Algorithm: Compare each pair of two nodes and to create clusters 49
3.9 DOM tree structure of Amazon search page with primitive type. 51
3.10 Algorithm: build the Structure Partition Tree . . . . . . . . . . . 55
3.11 Algorithm: Discover sequence pattern . . . . . . . . . . . . . . 56
3.12 Structure Partition Tree building process. . . . . . . . . . . . . . 57
3.13 A sample search result page from jobs.com. . . . . . . . . . . . 60
3.14 Table(s) inferred from a sampling page of jobs.com in Figure 3.13. 61

4.1 A high level view of the Hibernate architecture . . . . . . . . . . 63
4.2 Data model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
4.3 Sample ontology model for Car domain. . . . . . . . . . . . . . . 66
4.4 Sample GUI for search in ontology database. . . . . . . . . . . . . 67
Chapter 1

Introduction and Motivation

1.1 Challenge from the Internet

During the last ten years, the Internet has been rapidly growing large: it doubles approximately every 8 months in size; there are currently more than 4 billion Web documents, more than 200 million Internet users internationally, and more than 20 million topic categories to facilitate users browsing (data taken from [2] by Fensel et al), etc.

Comparing to the exponentially growing information on the Internet, however, the way people access such vast information does not change that much: information on the Internet is written, organized and structured in HTML or WML (Wireless Markup Language); people have to read them by HTTP or WAP (Wireless Application Protocol) enabled devices or browsers; people find the information by following some hyper-links or searching them in ad-hoc search engines. In fact, because of the lack of novel and promising ways to access data, we have already experienced that it is difficult to exactly locate the information we want, while the Internet is flooded with data. For example, you might experience spending hours in finding the best prices of products like digital cameras or holiday gifts.

You might intuitively think this can be done by the use of a search engine,
such as Google. However, there are three major issues remaining unresolved.

1. Keyword-based search engines often return a too huge amount of data and users might easily get lost (by following the links, navigating across Web sites) in such huge amounts of irrelevant material and may often miss the relevant information.

2. Existing keyword-based search engines retrieve irrelevant information that uses the keyword in a context other than the one in which the user is interested. For example, you are looking for the cheapest price of a java coffee and you type “java” into a search engine. The search engine will for sure return hundreds of pages about the Java programming language.

3. Existing keyword-based search engines may miss relevant information that semantically employs equivalent words (synonyms) other than the keyword submitted to the search engine.

These issues pose a major challenge to research, “the quality of search results, in terms of completeness, accuracy, authenticity, and timeliness needs to be drastically improved and should ideally be guaranteed.” (The third QoS in paper [1]).

Many researchers want to use intelligent software to help people easily find and locate information on the Internet. However, the first problem they meet is that Web pages on the Internet are written in a natural language for human beings. Thus, “there is a gap between the information available for computer applications to process and the information kept in human readable form on those Web pages” [2]. Recently many research works have been proposed to entirely or partially fill in this gap that causes problems in accessing and processing Web information.

To fill in the gap described above, this thesis shows how to build an instances-rich ontology database. We organized this chapter as follows: Section 1.2 provides a basic introduction to ontologies; Section 1.3 illustrates several application scenarios where instances-rich ontology databases could be useful and thus motivates us to do a deep study in this area; Section 1.4 briefly introduces the contribution and outlines the structures of this thesis to give you a whole picture about what we are going to talk about.
1.2 Ontology

Ontologies originated from artificial intelligence communities, which includes knowledge engineering, natural language processing, and knowledge representation. Recently, they have widely spread in the fields of research for intelligent information integration, cooperative information systems, information retrieval, electronic commerce, and knowledge management, etc. The reason that ontologies are becoming so popular for investigation is because of what ontology promises:

A shared and common understanding of some domain that can be communicated among people and application systems. Because ontologies aim at consensual domain knowledge, their development is often a cooperative process involving different people, possibly at different locations. People who agree to accept an ontology are said to “commit” themselves to that ontology. (By Fensel et al [2])

In one words, ontology is used to make knowledge sharing and reusing easy. A widely accepted definition for an ontology is the one by Gruber and Giarino [30]: “An ontology is a formal, explicit specification of a shared conceptualization.” We could summarize several properties about an ontology as follows:

• It is an abstract model of some real world. This model uses concepts to identify the real world. For example, a sample ontology might consist of the concepts of red and white wine, grape varieties, vintage years, wineries and so forth that characterise the domain of “wine”.

• There might be some constraints or relationships among these concepts to reflect the real world. These constraints will be explicitly defined in the ontology. In “wine” ontology for example, there are relationships such as “wineries produce wines”, “wines have a year of production”.

• The model is formally defined in a standard ontology language in order to be understandable by computer applications;

• It is not only used by the people who build it, but also well accepted by other organizations. This wine ontology might be developed initially for a particular application, such as a stock-control system at a wine warehouse.
As such, it may be considered similar to a well-defined database schema. So having been developed for one purpose, it can be published and reused for other purposes. For example, a given winery may use the wine ontology to link its production schedule to the stock system at the wine warehouse. Alternatively, a wine recommendation program may use the wine ontology, and a description (ontology) of different dishes to recommend wines for a given menu.

- The concepts in the ontology might be optionally associated with instances. In this thesis, we define such an ontology as instances-rich ontology. For example, concept “vintage years” may have instance like “1960”, concept “grape varieties” may have all kinds of grapes.

In one words, ontology allows a programmer to specify, in an open, meaningful way the concepts and relationships that collectively characterise some domain.

1.3 Application scenarios

1.3.1 From Keyword-based to Concept-based Search

One application scenario for using an instances-rich ontology is to facilitate concept-based search [3] over data on the Internet. Concept-based search utilizes a kind of query that consists of one or more concept-value pairs. Consider an example taken from [3]: there is a user who is looking for the movie called “The Caine Mutiny” but forgot the exact title. He can only remember the word “Caine”, so he submits this word to a search engine. A search engine or some movie portal with search capability such as IMDB would probably return many movies starring the actor Michael Caine, leading to poor precision of the search result, and the user would have a hard time finding the requested information in this very large result set. However, if there is a search engine that can support concept-based queries such as “title=Caine”, i.e., that treats the whole Web as a huge database, and performs SQL-like queries, we could get much more exact and relevant answers. An instances-rich ontology provides a back-end database for such a search engine to perform concept-based queries.
1.3.2 Automated form filling

Many commercial Web sites are backed by traditional database and use template engines such as CGI, JSP (Java Server Page), ASP (Active Server Page), etc to generate dynamic content. These information repositories are hosted by so-called Web Portals. Web Portals use HTML forms to let users specify search criteria, generate corresponding SQL statements to query underlying relational databases and finally fill in the dynamic part of the template to produce the final pages to the users.

However, little of this dynamic content is being crawled and indexed by popular search engines currently. Today’s search and categorization services cover only a portion of the Web called the publicly indexable Web, which is the set of Web pages reachable by only following hyper-links, ignoring pages that are generated by filling and submitting HTML forms. As the hidden web contains a lot of high quality information (e.g., census data, patents, telephone numbers and trademarks), many researchers are now studying how to crawl the hidden Web.

Crawling the hidden Web requires automated form filling. But we can’t fill the form with all possible value combinations. For example, consider a search form of a Car portal containing two drop-down boxes (two fields): one box labeled with “make” containing car manufacture companies such as “Audi”, “Ford”, etc, and a second box labeled with “model” containing car models. When users select a specific “make” in the first drop-down box, this portal will fill in a range of car models specific to this “make” and therefore forces users to select “model” in this range. Such a client-side constraint is very common in many search portals by using Javascript. As many crawlers have problems to directly understand constrains of semantic relationship among fields exposed by Javascript, it would be easy to let the crawler understand the constraints through description logics in ontologies, for example, “A4” is a model of “Audi”, it does not make sense for crawler to build a query with “A4” as “model”, but “Ford” as “make”. Even worse, if the “model” field is not a drop-down box but a text box, without the knowledge of the constraint, the crawler will not even be able to fill the form.

But constraints like this will only be available after we build a ontology database with rich instances inside. Again, instance extraction will be need to
build such database.

1.3.3 From Web to Semantic Web

Instances-rich ontology also facilitates the building of the Semantic Web [20]. As said in the beginning of this chapter, a vast amount of information on the Internet is presented for human beings but not for machines to process. To fill in this gap, researchers of the W3C proposed the Semantic Web. Documents of the Semantic Web contain meta data to express the meaning of their content, so computer applications can effectively and efficiently understand and query information by processing the associated meta data. Many applications ranging from information integration to Web services have benefited from Semantic Web.

Tools such as SHOE [33] and OntoBroker [34] allow users to easily annotate HTML documents with semantic markups (i.e., meta data in the Semantic Web). However, a vast amount of legacy data in the Internet (such as product descriptions) before or even after the Semantic Web has been invented is still being encoded in “plain” HTML without meta data.

Instances-rich ontology can be able to bootstrap such legacy HTML pages with meta data, or to give HTML-to-Semantic Web translator some recommended meta data as markups.

1.3.4 Ontology matching

The Semantic Web provides a vision that computer applications can understand and query information data through meta data. Such meta data forms part of an ontology. In other words, an ontology has been used to describe the semantic meaning of the data. Now there is another problem exposed. The Semantic Web is typically built decentralized. One organization could build its Semantic Web with entirely or partially different meta data (terminologies) from another organization for the similar domain, which leads to different ontologies. For example, Figure 1.1 shows two ontologies about computer science departments in different universities with instances information.

This challenges applications that rely on Semantic Web to query or organize information. It is necessary for these applications to find a semantic mapping among ontologies. With the knowledge of semantic correspondences among
their elements (i.e., a semantic mapping among ontologies), we can integrate data from disparate ontologies.

Doan et al [17] develop a system called “GLUE” to assist in the ontology mapping process. This system employs machine learning techniques to find such a mapping. It uses well-founded probabilistic definitions for several practical similarity measures. But such a probabilistic calculation is based on instances information retrieved from HTML pages.

In this case, ontologies with rich instances come into play. The instances-rich ontology facilitates the calculation of probabilistic functions and eventually facilitates the computation of an ontology mapping.

1.4 Overview of contribution

The aim of this thesis is to bootstrap an instances-rich ontology from Web documents. One of the major steps in building such an ontology is to extract data instances. Instance extraction from HTML is usually performed by software modules called wrappers. Wrappers were built manually by early approaches,
so it is usually a difficult and labor-intensive task. As Web resources might change frequently, manually built wrappers are hard to maintain. This thesis proposes two automated instance extraction approaches:

1. HTML table extraction. Because most of data on the Web is coming from relational databases, an intuitive assumption can be made that it will also be presented in HTML table format. However, Web developers may use HTML tables for different purposes (for layout or for content), in different ways (column-wise or row wise) with flexible HTML syntax (colspan and rowspan), it deserves a full table understanding process for instance extraction. Chapter 2 illustrates the process of retrieving instances from HTML tables that contain a list of records (such as product description, address record, etc).

2. Spatial Locality based Instance Extraction (SLIE). Observation shows that not all Web sites will present their list of records in rigid table format. Therefore, the first approach is likely to fail in such situation. The SLIE approach utilizes an assumption—all instance items and their attributes are located at relatively regular positions inside HTML pages. This assumption is according to a premise—Web pages from one Web site are often dynamically generated by a template engine fed with dataset from underlying database. Based on this assumption, an XPath clustering algorithm is used to group semantically related instances (i.e., belonging to one concept) together; a Structure Partition Tree algorithm is used to put instances that are properties of one instance to one partition.

This thesis is organized as follows: Chapter 2 and 3 illustrate two instance extraction techniques respectively: one is about the table extraction process; another is about Spatial Locality based Instance Extraction (SLIE). Chapter 4 describes the way we store the extracted concepts and their corresponding instances to form the ontology, and also the way to access the ontology. Chapter 5 outlines future work, summarizes and concludes this thesis.
Chapter 2

Extracting instances for ontology from HTML Table

2.1 Introduction

<table>
<thead>
<tr>
<th>Photo</th>
<th>Make/Model</th>
<th>Year</th>
<th>Price</th>
<th>Mileage</th>
<th>Location</th>
<th>Sold By</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>no photo</td>
<td>BMW 3-Series</td>
<td>2001</td>
<td>$39500</td>
<td>18900</td>
<td>New York, NY 0.0 miles</td>
<td>Owner</td>
<td>12/03/02</td>
</tr>
<tr>
<td><img src="image" alt="Lexus SC430" /></td>
<td>Lexus SC430</td>
<td>2002</td>
<td>$45000</td>
<td>8208</td>
<td>New York, NY 0.0 miles</td>
<td>Owner</td>
<td>02/03/03</td>
</tr>
<tr>
<td><img src="image" alt="Audi TT" /></td>
<td>Audi TT</td>
<td>2001</td>
<td>$32995</td>
<td>17000</td>
<td>New York, NY 0.0 miles</td>
<td>Audi Certified</td>
<td>05/15/03</td>
</tr>
<tr>
<td><img src="image" alt="Audi S4" /></td>
<td>Audi S4</td>
<td>2000</td>
<td>$31995</td>
<td>29640</td>
<td>New York, NY 0.0 miles</td>
<td>Audi Certified</td>
<td>05/15/03</td>
</tr>
</tbody>
</table>

Figure 2.1: A table describing cars(yahoo.com).

As an intuitive and traditional way to present relational information, tables are used frequently in Web documents to describe schedules, experiment results, search results, etc. Since HTML tables contain such rich information, many
research areas or applications including knowledge management, information retrieval, information extraction, Web mining, etc have been working around HTML tables, extracting and learning knowledge behind tables. Among these areas and applications, ontology bootstrapping is one of them.

A table is often used to describe related objects by property-value pairs. The properties specify the important information that we need to know for identification and utilization of the described objects. For example, in Figure 2.1, we can identify each car by its values for the “Make/Model”, “Year”, “Price” ... properties. Property-value pairs are good candidates for concept-instance pairs in the ontology maybe after proper separation and alignment of property-value pairs. In Figure 2.2 for example, the property-value pair “Make/Model” has been correctly separated and the concepts “make” and “model” have been inferred. To save writing space, the other property-value pairs that have a direct mapping to concept-instance pairs in the ontology are not shown in this figure. We have seen that, to bootstrap instances rich ontologies, HTML tables serve a good resource for us to discover.

Figure 2.2: Part of the ontology created from the HTML table in Figure 2.1.

However, for a computer application to automated bootstrap and construct ontologies from a wide range of tables, there are many potentially difficult problems to face. We call the process to solve these problems as table understanding process because a computer application needs to learn what the table creator wants to express through the table. In the following list, we describe some problems that might arise during the table understanding process:
1. **Tables for layout rather than for information.** The ability to spatially arrange visual elements in relation to each other is fundamental to Web design. HTML tables have been the only reliable way to define layout grids before the advent of style sheets with positioning support. Even with CSS being well supported nowadays, tables still remain the most commonly used technique in Web design. Observation shows that in typical e-commerce Web page, the ratio of tables for layout purpose against tables for relational data representation is about 10 : 1. Experiments in Section 2.8 will show this observation. However, our goal is to extract data instances from data-rich HTML tables, but not from tables for layout purpose. Distinguishing between these two kinds of tables is the key point to the success of the table understanding process. We call the step to solve this problem *table detection*, which will be discussed in detail in Section 2.5.

2. **Relaxation of HTML tables.** HTML tables do not come with a rigid schema as tables in relational databases. Almost any forms of two dimensional tables are acceptable according to HTML syntax. This relaxation puts additional burden on the process of table understanding. We further categorize this relaxation into the following two folders:

- **HTML table specific attributes.** The **rowspan** and **colspan** attributes play a significant role in determining an *is-a* relationship, i.e., which data instance belongs to which concept. This problem is about how to correctly align a data value to its corresponding property. We give the fundamental knowledge to these two attributes in Section 2.3, and solve the problem of property-value pair alignment in Section 2.4.

- **Some tables are column-wise, i.e., the table contains at least one heading row at the top of the table. Some tables are row-wise, i.e., the table contains at least one heading column at the left side of the table. Other than these two types of tables, a large number of tables do not use HTML table specific elements to indicate the heading. This diversity requires an additional module in our table understanding process; we call this module *table header detection*. It
will be discussed in Section 2.6.

- Violates W3C HTML specification. Many HTML source codes on Internet violate HTML specification while modern browsers such as Internet Explore still be able to display it as normal. Our table understanding process has to tolerant these errors too.

In this chapter, an automated approach to extract data instances and corresponding concepts for ontologies from HTML tables is proposed. This approach is a table understanding process (i.e., understand what the table creator wants to express through an HTML table). Section 2.2 outlines related work. Section 2.3 gives a short introduction to HTML table specific elements and attributes and provides a foundation knowledge to the discussion of the table understanding process in Sections 2.4—2.7. Section 2.8 shows the experiment results. Section 2.9 outlines the limitation of our work and concludes this chapter.

### 2.2 Contribution and related work

In this chapter, an automated table understanding process is contributed. It combines a sequence of steps including table structure analysis, table detection, table header detection.

The work presented in this thesis has been inspired from several works. Lim et al [5] propose an automated approach for retrieving hierarchical structured data from HTML tables. The approach first maps an HTML table into a so-called pseudo-table to solve the alignment problem of property-value pairs and then constructs the content-tree of the HTML table from the pseudo-table, which captures the intended hierarchy of the data content of the table. But it does not mention how to deal with tables that are only used for layout purpose. Wang et al [6] propose a machine learning based approach to classify each given HTML table to either layout or content table.

The structure analysis module in our work employs the notion of pseudo-table from Lim et al [5] but generates the pseudo-table somehow differently from [5], especially the way to deal with rowspan. In the structure analysis module, we also gather the information of a HTML table for features computation. These features are used by classification methods for table detection module, which is
inspired from Wang et al [6]. Our goal is to extract instances and corresponding concepts to fill in ontologies.

2.3 Table related elements and attributes

According to the HTML specification [21], the simplest form of an HTML table is to divide a rectangle into row and column cells and place information inside these cells. Some cells, usually located on a table’s top or side, are the headers of the table. The other cells (i.e., most cells of a table) contain data. HTML and XHTML represent a basic table using four elements. In markup, a table tag pair, \texttt{<table>...</table>}, contains an optional \texttt{<caption>} element, followed by one or more rows, \texttt{<tr>...</tr>}. Each row contains cells holding a heading, \texttt{<th>...</th>}, or data, \texttt{<td>...</td>}.

A table is made up of rows enclosed within \texttt{<tr>...</tr>}. The number of rows in the table is determined by the number of occurrences of the \texttt{tr} element. What about columns? In the first case, the number of columns in a table is determined by the maximum number of data cells in one row indicated by \texttt{<td>...</td>}, or headings indicated by \texttt{<th>...</th>} within the table. The headings for the table are set using the \texttt{th} element. The actual cells of the table are indicated by the \texttt{td} element. Both the \texttt{td} and \texttt{th} elements can enclose an arbitrary amount of data of just about any type. In the second case, when table cells need to be larger or smaller, the markup that creates tables is more complicated. The following markup showed in Figure 2.3 creates tables that are somewhat more complicated. By adding the \texttt{rowspan} and \texttt{colspan} attributes to the table elements, it is possible to create data cells that span a given number of rows or columns.

The basic idea of the \texttt{rowspan} and \texttt{colspan} attributes for \texttt{<td>} and \texttt{<th>} is to extend the size of the cells across two or more rows or columns, respectively. To set a cell to span three rows, use \texttt{<td rowspan="3">}; to set a heading to span two columns, use \texttt{<th colspan="2">}.

In general, setting the value of \texttt{colspan} or \texttt{rowspan} to more than the number of columns or rows in the table should not extend the size of the table. However, some HTML pages violate this rule while modern browsers will tolerate such an
**ROWSPAN Example**

<table>
<thead>
<tr>
<th>Element 1</th>
<th>Element 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element 3</td>
<td></td>
</tr>
</tbody>
</table>

**COLSPAN Example**

<table>
<thead>
<tr>
<th>Element 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element 2</td>
</tr>
</tbody>
</table>

Figure 2.3: Rendering of *rowspan* and *colspan* examples.
error and still be able to display table correctly. This puts more burden on our table understanding process.

The other attributes of HTML tables like `border`, `align`, `valign`, `width`, `bgcolor`, `background`, `cellpadding`, `cellspacing` etc, are often used for layout. So they are not relevant to structure analysis module. However, these attributes provide visual cues for header detection module in the table understanding process. We will discuss them in Section 2.6.

### 2.4 Structure analysis

As discussed in previous sections, to capture the *is-a* relationship for ontology from HTML tables, it is important to solve the alignment problem of property-value pairs. In order to solve this problem, we map every HTML table into a *pseudo-table* (inspired from [5]). The pseudo-table can be considered as a special type of HTML table that has the same number of columns in each row and the same number of rows in each column. To make this mapping correctly, we have to deal with two attributes of `th` and `td`, namely `rowspan` and `colspan` correctly. The algorithm presented in this section is just for this purpose and is called *structure analysis* among the other modules of the table understanding process. Once the pseudo-table is generated, the structure of the HTML table is clear to the computer application.

Let’s consider the table and its HTML source code showed in Figure 2.4 as an example. The table has four rows (i.e., four `tr`s). However, the numbers of columns for each row are different. We use the following rules to map a HTML table into a pseudo-table:

**colspan** If a `th` or `td` element contains `colspan = n`, at the same row of the cell that represents this element, we insert `n − 1` cells to right of the this cell and replicate the data content of this cell `n − 1` times to the new cells since the data content of the current cell is meant to be the same over the next `n − 1` cells in the same row. For example, we insert a new cell next to the cell that contains “Price” as data content and replicate its content.

Figure 2.5 shows the corresponding pseudo-table.

**rowspan** If a `th` or `td` element in a HTML table contains `rowspan = n`, at the
<table>
<thead>
<tr>
<th>Price</th>
<th>engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>retail</td>
<td>invoice</td>
</tr>
<tr>
<td>73%</td>
<td>27%</td>
</tr>
<tr>
<td>16%</td>
<td>84%</td>
</tr>
</tbody>
</table>

1.7 l 141 HP

Figure 2.4: Rendered table in a Web browser and its HTML source code.

same column of the cell that represents this element, we insert $n - 1$ cells beneath the current `th` cell in the current column, and replicate the data content of the current cell to the underneath rows $n - 1$ times. The cells that contain “engine” and “1.7 l 141 HP” as data content are examples for this rule.

Note, very important, whenever we replicate the data content of a cell, we also replicate the visual style such as font, background color, etc of the cell. This is necessary to master the semantic meaning of the table through its visual cues, especially for table header detection discussed in Section 2.6.

Many HTML source codes miss closing tags, contain unnecessary tags etc, which aggravates the difficulty of structure analysis of tables. We use jtidy [24] to clean the HTML source code and generate a tree structure to facilitate table processing. In fact, the tree structure that is generated by jtidy is a DOM [22] tree. DOM (Document Object Model) specifies a set of programming objects
and functions that let you work with the data in an XML document (the DOM objects are implemented in programming languages like Java).

Before going through the algorithm, we first introduce two classes, namely CellLocator and RowSpanHelp, that will be used in the algorithm of structure analysis:

**CellLocator**

An instance of type CellLocator represents a cell inside a table. Basically, it has three properties:

- **cindex** the index of the column that the cell resides in;
- **rindex** the index of the row that the cell resides in;
- **content** the content of the cell.

In the computer application, we use a list of CellLocators to represent a HTML table. By sorting against rindex or cindex, we can quickly get the content of a certain row or column respectively.

---

Figure 2.5: Corresponding pseudo-table of table in Figure 2.4.
Algorithm GeneratePseudoTable(table)
input
  table: a table document as DOM element
begin
1. numCol=findColumnNumber(table);
2. rindex = 0;
3. for each row tr do
4.  cindex = 0;
5.  for each column thtd in row tr do
6.    rowspan = getRowspan(thtd);
7.    adjustColumnIndex(rindex, cindex, helps);
8.    if (rowspan > 1)
9.      RowSpanHelper help = new RowSpanHelper(cindex, rindex, rowspan);
10.     helps.add(help);
11.    for each effect row effectRow of help do
12.      CellLocator cell = new CellLocator(cindex, effectRow, content);
13.      cells.add(cell);
14.    endfor
15.  endif
16.  colspan = getColspan(thtd);
17.  if ((0 == colspan) || (1 == colspan))
18.    CellLocator cell = new CellLocator(cindex, rindex, content);
19.    cells.add(cell);
20.  elsif (colspan > 1)
21.    for s=0; s < colspan; s++ do
22.      CellLocator cell = new CellLocator(cindex, effectRow, content);
23.      cindex++;
24.    endfor
25.  endif
26.  endfor
27.  rindex++;
28.endfor
end

Figure 2.6: Algorithms: GeneratePseudoTable
RowSpanHelp

The class RowSpanHelp is used to help identifying the correct column index for CellLocator. An instance of type RowSpanHelp will be created and put into help list when the program meets the attribute rowspan with the value larger than 1 (only the value larger than 1 will need additional care). This class has the following methods:

**getEffectRows** Get all the effective rows that this rowspan affects. In Figure 2.4 for example, the th element with “engine” as content is located on row 0 (i.e., rindex = 0). As it has rowspan = 2 as its attribute, its effective row index is 1. If it has rowspan = 3 as its attribute, its effective row indices are 1 and 2.

**getEffectRowUntil** Get the maximum row index this rowspan affects. Again in the example above, this function will return 1, because the row with index 1 is the maximum row this rowspan will affect. But another td element with “1.7l L4 118 HP” as content is located on row 2, its effective row is 3 and getEffectRowUntil function will also return 3.

**removeEffectRow** Remove the effective rows.

The algorithm to generate pseudo-tables is shown in Figure 2.6. The algorithm takes a table (an element of a DOM tree) as input and generates a list of CellLocator to represent a pseudo-table.

Because not every row of a table has the same number of columns, the findColumnNumber function is used in line 1 to find the maximum number of columns in a given table. Applied on the table showed in Figure 2.4, the function will return 3.

In line 4 (before iterating every cell in one row), cindex is initialized to 0. Because of the influence of Rowspan, cindex will not be correct if we simply add one after passing every cell in the row and return to 0 after finishing one row. We need to adjust cindex in line 7. This function is shown in Figure 2.7. cindex will be adjusted only when the current cell is affected by some RowSpanHelp, namely help’s effect rows contains current cell’s row and current cell’s column index is smaller or equal than help’s column index. A rowspan can only affect
Function AdjustColumnIndex(rindex, cindex, helps)
input
  rindex: the current row index.
  cindex: the current column index, maybe wrong.
  helps: a list of RowspanHelper.
begin
1. for each RowspanHelper helper do
2.   if helper.getEffectRows().contains(rindex)
3.     if helper.cindex <= cindex
4.       cindex++
5.     helper.removeEffectRow(rindex)
6.   endif
7. endif
8. endfor
end

Figure 2.7: Function: AdjustColumnIndex

the current row once, because after adjusting cindex, cindex will be correct for subsequent cells in the row. So we need to remove the current row as effect row in help.

Line 8-15 illustrates how to insert \( n - 1 \) cells to the right of the current cell and replicate the data content of the current cell \( n - 1 \) times to the new cells since the data content of the current cell is meant to be the same over the next \( n - 1 \) cells in the same row. Line 21-23 illustrates how to insert \( n - 1 \) cells beneath the current TH cell in the current column, and replicate the data content of the current cell to the underneath rows \( n - 1 \) times.

2.5 Table Detection

As said in the beginning of this chapter, HTML tables have the tradition to be used for defining layout grid. Related visual elements can be clustered together and put into a cell of the grid. Such HTML tables are not used for displaying relational information. The presence of the \(<\text{TABLE}>\) tag does not necessarily indicate the presence of a relational table. To increase the accuracy of our table understanding process, we have to check each table whether it is used for displaying relational information and we simply drop it if it is not. Wang et
al [6] uses machine learning techniques to do this and their work is presented in this section. First, the definition of table classification is given below.

**Definition 1 Table classification**

**Genuine table**: HTML tables where a two dimensional grid is semantically significant in conveying the logical relations among the cells.

**Non-genuine table**: HTML tables where `<TABLE>` tags are used as a mechanism for grouping contents into clusters for easy viewing only.

Therefore, table detection is a technique, which can classify a given HTML table enclosed by the `<TABLE>`/<TABLE> tag as either genuine or non-genuine table.

### 2.5.1 Genuine table classification with decision tree

Our work is inspired from Wang et al [6] and uses machine learning to do classification. Most machine learning techniques are designed to learn which are the most appropriate features to use for making their decision. For example, decision tree methods choose the most promising feature to split on at each point, and should—in theory—never select irrelevant features. In the case of table classification problem, we have to find a combination of features that together provide significant separation between genuine and non-genuine tables while at the same time constrain the total number of features to avoid the curse of dimensionality.

**Notions**

Some notions for features are listed as following. Given a table $T$:

- $rowNum$: the number of rows;
- $colNum$: the number of columns;
- $c_i$: the number of cells in row $i, i = 1, \ldots, rowNum$;
- $r_i$: the number of cells in column $i, i = 1, \ldots, colNum$;
- $cellNum$: the total number of cells in $T$;
- $cl_i$ or $cl$: the length of cell $i, i = 1, \ldots, cellNum$. 
• $m_i$: the mean cell length,
  if $i = 1, \cdots, \text{rowNum}$, $m_i$ denotes mean cell length of certain row $i$.
  if $i = 1, \cdots, \text{colNum}$, $m_i$ denotes mean cell length of certain column $i$.

Features for table detection

• Average number of columns, computed as the average number of cells per row:
  $$c = \frac{1}{\text{rowNum}} \sum_{i=1}^{\text{rowNum}} c_i$$  \hspace{1cm} (2.1)

• Standard deviation of number of columns:
  $$dC = \sqrt{\frac{1}{\text{rowNum}} \sum_{i=1}^{\text{rowNum}} (c_i - c) \times (c_i - c)}$$  \hspace{1cm} (2.2)

• Average number of rows, computed as the average number of cells per column:
  $$r = \frac{1}{\text{colNum}} \sum_{i=1}^{\text{colNum}} r_i$$  \hspace{1cm} (2.3)

• Standard deviation of number of rows:
  $$dR = \sqrt{\frac{1}{\text{colNum}} \sum_{i=1}^{\text{colNum}} (r_i - r) \times (r_i - r)}$$  \hspace{1cm} (2.4)

Through observation, we notice that the majority of non-genuine tables contain graphics in gif, jpeg, etc formats or even nested tables, because they are used for defining layout grid. On the contrary, genuine tables should contain relational data. To design features for distinguishing them, we compute three more layout features based on cell length in terms of the number of characters:

• Average overall cell length:
  $$cl_{av} = \frac{1}{\text{cellNum}} \sum_{i=1}^{\text{cellNum}} cl_i$$  \hspace{1cm} (2.5)

• Standard deviation of cell length:
  $$dcl = \sqrt{\frac{1}{\text{cellNum}} \sum_{i=1}^{\text{cellNum}} (cl_i - cl_{av}) \times (cl_i - cl_{av})}$$  \hspace{1cm} (2.6)
• Average cumulative length consistency, $CLC$.

The last feature is designed to measure the cell length consistency along either row or column directions. It is inspired by the fact that most genuine tables demonstrate certain consistency either along the row or the column direction, but usually not both, while non-genuine tables often show no consistency in either direction. First, the average cumulative within-row length consistency, $CLC_r$, is computed as follows. Let the set of cell lengths of the cells from row $i$ be $R_i$, $i = 1, \ldots, r$

1. Compute the mean cell length, $m_i$, for certain row $R_i$

2. Compute cell length consistency for each cell in the row $R_i$

$$LC_{cl} = 0.5 - \min\{\frac{\|cl - m_i\|}{m_i}, 1.0\}$$

$L_{cl}$ is used to measure the length consistency of a certain cell. Depending on the length of the cell, we can see there are three cases for $L_{cl}$ and Figure 2.8 plots this linear function:

(a) Extreme consistency, if the cell with cell length $cl = m_i$, $L_{cl} = 0.5$;

(b) Extreme inconsistency, if the cell with cell length $cl \geq 2m_i$, $L_{cl} = -0.5$;

(c) Otherwise, $L_{cl}$ ranges between $(-0.5, 0.5)$.

3. Compute cumulative length consistency within this row $R_i$:

$$CLC_i = \sum_{cl \in R_i} LC_{cl}$$

$CLC_i$ measures the cumulative length consistency for this row $R_i$. When most cells within $R_i$ are consistent, the cumulative measure $CLC_i$ is positive, indicating a more or less consistent row.

4. Take the average across all rows:

$$CLC_r = \frac{1}{r} \sum_{i=1}^{r} CLC_i$$
5. After the within-row length consistency $CLC_r$ is computed, the within-column length consistency $CLC_c$ is computed in a similar manner. Finally, the overall cumulative length consistency is computed as

$$CLC = \max(CLCL_r, CLC_c)$$  \hspace{1cm} (2.10)
training subset into two subsets and generates two child nodes. The process is repeated at each newly generated child node until a stopping condition is satisfied, and the node is declared as a terminal node based on a majority vote. The maximum impurity reduction, the maximum depth of the tree, and minimum number of samples are used as stopping conditions.

At the testing stage, a feature vector is the input to a decision tree, a decision is made at every nonterminal node as to what path the feature vector will take. This process is continued until the feature vector reaches a terminal node of the tree, where the associated class is assigned to it.

2.6 Table header detection

<table>
<thead>
<tr>
<th>City</th>
<th>Sheffield</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make</td>
<td>BMW</td>
</tr>
<tr>
<td>Model</td>
<td>320d SPORT</td>
</tr>
<tr>
<td>Year</td>
<td>2003</td>
</tr>
<tr>
<td>Price £</td>
<td>23,500</td>
</tr>
<tr>
<td>Mileage</td>
<td>6500</td>
</tr>
<tr>
<td>Fuel</td>
<td>Diesel</td>
</tr>
<tr>
<td>Doors</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 2.9: Sample HTML table header.

For any detected genuine table, we generate a pseudo-table. The next step in the table understanding process is to detect the table header. In general, table headers are property names that correspond to concepts in the ontology. Correctly identifying headers is very important in bootstrapping ontologies. However, because of the flexibility of HTML, the table header could be column-wise or row-wise. For example, the left table of Figure 2.9 is row-wise and the right is column-wise. Several heuristics have been employed in order to detect table headers.

1. If there are th elements, according to the HTML specification, these elements must be header elements.

2. If there is thead element, according to the HTML specification, the elements that are contained inside of the thead element must be header
elements.

3. Check the visual style of td elements, if one row or one column of td elements contains elements like <b> or attributes such as style="bold", these elements are header elements.

4. Compare the visual style of the first row to the other rows. If there exist consecutive rows with the same visual style as the first row, merge them together with the first row to form a new row. Figure 2.10 gives such merge example for the table in Figure 2.4. Comparing this new row to the other rows in the table, if there exists difference in visual style, we say this new row is a header row. Do the same thing in column direction. This assumes that the visual cues (background, fonts, etc) of the header are different from the other text data inside the HTML table.

5. Check cell length consistency. If some row or some column is length consistent, it is not considered as table header. For example, the “see” column in Figure 2.9 is not header column.

<table>
<thead>
<tr>
<th>Price-retail</th>
<th>Price-invoice</th>
<th>engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>73%</td>
<td>27%</td>
<td>1.7 l L4 118 HP</td>
</tr>
<tr>
<td>16%</td>
<td>84%</td>
<td>1.7 l L4 118 HP</td>
</tr>
</tbody>
</table>

Figure 2.10: Merge example for the table in Figure 2.4.

### 2.7 Whole table understanding process

Now it is the time to put all together. Figure 2.11 shows the table understanding and extraction process flow. Experiments in Section 2.8 shows performance measure for each different module.
2.8 Experiments

2.8.1 Experiment of genuine table classification

Data collection

In order to test the performance of the table detection algorithm, some test dataset is needed. For this thesis, we collected two datasets (i.e., sample HTML pages) manually from two domains: car selling and job search Web sites.

For each of these Web sites, 5-10 web pages are downloaded. Then these sample web pages are loaded into Microsoft Frontpage. By using Microsoft Frontpage’s WYSIWYG feature, one can easily identify genuine tables among all other tables. For each of these identified genuine tables, the attribute-value pair genuine="true" is added to the table element.

We used the following Web sites for our experiments.

- “Car” domain: (25 HTML pages downloaded)

http://www.carsearch.com
http://classifieds.autos.yahoo.com/class/browse.html
http://www.channel4.com/4car/
http://www.usedcarmart.co.uk/

- “Job” domain: (25 HTML pages downloaded)
  
  http://careers.yahoo.com/
  http://www.careerbuilder.com/
  http://jobsearch.usajobs.opm.gov/
  http://www.jobs.net/

Result analysis

Figure 2.12: Generated decision tree of the test dataset.

Figure 2.12 is the pruned decision tree that is generated for the test dataset. To generate this decision tree, we use 518 table instances as training set and one used for test. The size of the tree is 11, it contains 5 internal nodes and 6 leaves. Each internal node specifies the criteria to split the tree. The meaning of the node names are listed as follows:

- “anr” stands for “average number of rows”;
- “anc” stands for “average number of columns”;

28
• “aclsl” stands for “standard deviation of average cell length”;

• “clc” stands for “average cumulative length consistency”;

• “Gen” stands for genuine table;

• “Non-gen” stands for non-genuine table.

Each leaf is classified into either genuine or non-genuine table. The numbers in parentheses at the end of each leaf tell us the number of examples in this leaf.

=== Stratified cross-validation ===

Correctly Classified Instances 518 99.8073 %
Incorrectly Classified Instances 1 0.1927 %
Kappa statistic 0.984
Total Number of Instances 519

=== Detailed Accuracy By Class ===

<table>
<thead>
<tr>
<th>TP Rate</th>
<th>FP Rate</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Measure</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.971</td>
<td>0</td>
<td>1</td>
<td>0.971</td>
<td>0.985</td>
<td>genuine</td>
</tr>
<tr>
<td>1</td>
<td>0.029</td>
<td>0.998</td>
<td>1</td>
<td>0.999</td>
<td>non-genuine</td>
</tr>
</tbody>
</table>

=== Confusion Matrix ===

a b <-- classified as
33 1 | a = genuine
0 485 | b = non-genuine

Figure 2.13: Analysis result for sample dataset (training data)

Because the dataset is relatively small, we use 10-folder cross validation on the dataset (i.e., we split the dataset into 10 folders equally, run the test 10 times, for each run we use one folder for training and the other 9 folders for testing). This is a typical test scheme in machine learning literature when dataset is small.

The stratified cross validation (Figure 2.13) paints a more realistic picture of our classification scheme. The kappa statistic measures the agreement of the prediction with the true class—1.0 signifies complete agreement. The confusion matrix is more commonly named contingency table. In our case we have two classes (genuine and non-genuine), and therefore a 2x2 confusion matrix. The
The number of correctly classified instances is the sum of diagonals in the matrix; all others are incorrectly classified. We have totally 519 instances (tables); 518 instances (99.8073% of total) are correctly classified and 1 instance (0.1927% of total) is genuine table but incorrectly classified as non-genuine table.

The Truth Positive (TP) rate is the proportion of sample instances which were classified as class x, among all examples which truly have class x, i.e., how much part of the class was captured. For genuine class, we have 0.971 TP rate; for non-genuine class, we have 100% TP rate. Truth Positive is equivalent to Recall. In the confusion matrix, this is the diagonal element divided by the sum over the relevant row.

The False Positive (FP) rate is the proportion of sample instances which were classified as class x, but belong to a different class, among all examples which are not of class x. For genuine class, we have 0 FP rate; for non-genuine class, we have 0.029 FP rate. In the matrix, this is the column sum of class x minus the diagonal element, divided by the rows sums of all other classes.

The Precision is the proportion of the sample instances which truly have class x among all those which were classified as class x. For genuine class, we have 100% precision rate; for non-genuine class, we have 99.8% precision rate. In the matrix, this is the diagonal element divided by the sum over the relevant column.

The F-Measure is a combined measure for precision and recall by using the following formula. For either genuine or non-genuine class, we get very high F-Measure (98.5% and 99.9% respectively).

\[ F = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \] (2.11)

2.8.2 Experiment of whole table understanding process

Data collection

For this thesis, we collected datasets from two domains: car selling and job search Web sites (URL addresses are listed in the last section) by a multi-threaded crawler (called spiderman in source code). We choose starting URLs that list instances information in HTML tables to feed to the crawler, because until now we can only deal with HTML tables. Obviously, our table understand-
ing process has low accuracy and efficiency on Web sites that do not list records in rigid HTML table format. The crawled HTML documents are saved into the database. On the computer with Intel Pentium-4 2.6 GHZ, 512M RAM, 40GB harddisk configuration, we measure the performance of each module of the table understanding process in Table 2.2.

<table>
<thead>
<tr>
<th>Modules</th>
<th>Car domain</th>
<th>Job domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML pages #</td>
<td>204</td>
<td>118</td>
</tr>
<tr>
<td>Table #</td>
<td>3769</td>
<td>3320</td>
</tr>
<tr>
<td>Genuine table #</td>
<td>300</td>
<td>311</td>
</tr>
</tbody>
</table>

Table 2.1: Crawled HTML pages for different domains. The numbers of genuine tables are detected by the system.

<table>
<thead>
<tr>
<th>Modules</th>
<th>Car - Time(msec.)</th>
<th>Job - Time(msec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clean table</td>
<td>1351</td>
<td>1082</td>
</tr>
<tr>
<td>Generate pseudo table and compute features</td>
<td>430</td>
<td>531</td>
</tr>
<tr>
<td>Genuine table detection</td>
<td>190</td>
<td>220</td>
</tr>
<tr>
<td>Header detection</td>
<td>191</td>
<td>201</td>
</tr>
<tr>
<td>Persist retrieved data</td>
<td>45626</td>
<td>30803</td>
</tr>
<tr>
<td>Overall</td>
<td>47788</td>
<td>32837</td>
</tr>
</tbody>
</table>

Table 2.2: Performance measure for each table understanding module.

**Noise elimination**

The extraction results for Car and Job domains are listed in table 2.3. We can see than the system introduced some noise. In order to eliminate the noise, we make the following assumption: a property that is candidate for concept of ontology typically appears many times. We can see for Car domain, “make”, “model”, “year”, etc appear very often, but “search”, “phone number” etc that might not relevant to Car domain appear not very often. So we can setup a threshold and cut occurrences of properties lower than this threshold (for 200 html documents, set occurrences threshold to 200).

Because we can not eliminate noise totally, this system does not perform well on arbitrary crawled HTML collections that actually do not contain many regular HTML tables. The system performs well on select HTML collections that describe list of records, such as online catalogs.
Table 2.3: Occurrences of properties of Car and Job domains

<table>
<thead>
<tr>
<th>Property</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>376</td>
</tr>
<tr>
<td>model</td>
<td>342</td>
</tr>
<tr>
<td>photo</td>
<td>90</td>
</tr>
<tr>
<td>mileage</td>
<td>380</td>
</tr>
<tr>
<td>area</td>
<td>360</td>
</tr>
<tr>
<td>title</td>
<td>90</td>
</tr>
<tr>
<td>mg</td>
<td>50</td>
</tr>
<tr>
<td>state</td>
<td>20</td>
</tr>
<tr>
<td>year</td>
<td>393</td>
</tr>
<tr>
<td>search</td>
<td>27</td>
</tr>
<tr>
<td>phone number</td>
<td>32</td>
</tr>
<tr>
<td>city</td>
<td>380</td>
</tr>
<tr>
<td>email</td>
<td>20</td>
</tr>
<tr>
<td>vw</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>email</td>
<td>153</td>
</tr>
<tr>
<td>a job for you</td>
<td>200</td>
</tr>
<tr>
<td>date</td>
<td>100</td>
</tr>
<tr>
<td>job summary</td>
<td>75</td>
</tr>
<tr>
<td>title</td>
<td>100</td>
</tr>
<tr>
<td>pay</td>
<td>100</td>
</tr>
<tr>
<td>company</td>
<td>100</td>
</tr>
<tr>
<td>job title</td>
<td>64</td>
</tr>
<tr>
<td>contact</td>
<td>153</td>
</tr>
<tr>
<td>opening date</td>
<td>125</td>
</tr>
<tr>
<td>agency</td>
<td>125</td>
</tr>
<tr>
<td>location</td>
<td>239</td>
</tr>
<tr>
<td>ads04-r5lpmp-1138dp</td>
<td>6</td>
</tr>
<tr>
<td>04-136 bhs</td>
<td>18</td>
</tr>
<tr>
<td>gs04b0225</td>
<td>6</td>
</tr>
<tr>
<td>hq-oig-de-2004-0016</td>
<td>24</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

2.9 Summary

This chapter describes a full table understanding process for instances extraction, which includes structure analysis (dealing with the flexibility of HTML syntax), table detection (dealing with genuine or non-genuine table), table header detection (dealing with column-wise or row-wise table). To eliminate noise, future work might use percentage threshold (e.g., eliminate property-value pairs with occurrences lower than 5% of the total occurrences).

The approach described in this chapter has a potential problem. Many Web sites will not present their relational information in a rigid table format. Table analysis will not work in this situation totally. To solve this problem, we describe a new approach in the next chapter.
Chapter 3

Spatial Locality based Instance Extraction (SLIE)

In Chapter 2, we extract instances from Web documents based on table analysis. However, current Web sites (such as online stores or online auctions) do not always use rigid HTML tables (i.e., as regular as underlying database tables) to present their instances information. Figure 3.1 gives a sample Web document retrieved from Amazon.com by querying the keywords "java programming". Each product instance listed on the Web page visually spreads its properties over three rows. The title, author and format properties of a product are on the first row. Price property is on the third row. How to retrieve instances on such Web documents will be examined in this chapter.

Fortunately, most instances-rich Web documents (i.e., Web documents that contain multiple records such as products) are dynamically generated: data are stored in a back-end database, and HTML pages are produced using scripts or templates, fed by the dataset from the database. Although there are many template engines and languages to write scripts, Crescenzi et al [4] generalize and abstract the page-generation process as the result of two separated activities: (1) a number of queries are executed on the underlying database to retrieve a source dataset, i.e., a set of database records that represent products to be published on the site pages; (2) the dynamically fetched dataset is encoded into static HTML code to produce the actual pages. The static HTML code might
contain links, images, and other HTML elements for layout purpose. We will see later in this chapter that pages generated by the same content management system or template engine usually carry multiple instances of a concept following a particular alignment and format, i.e., properties of instances exhibit consistent and regular spatial locality inside the HTML pages. We define:

**Definition 2 Page Class** [4]

A collection of pages that are generated by the same script in a site is called page class.

Currently, most data extraction works for HTML are usually performed by software modules called *wrappers*. Early approaches to wrapping Web sites were based on manual techniques. A key problem with manually coded wrappers is
that writing them is usually a difficult and labor intensive task, and that by their nature wrappers tend to be brittle and difficult to maintain.

The Spatial Locality based Instance Extraction (SLIE) approach described in this thesis can generate wrappers in an automated process, which analyzes one sampling page from a page class. When the wrapper is generated, it is straightforward to annotate data items of that page class or to extract instances or concepts from that page class. It is also possible to improve and verify the wrapper by using more than one sampling pages.

SLIE is an automated wrapper generation process that contains the following steps:

1. First, we discover and maintain a *type system*. All data items that have similar spatial locality will be assigned to the same type. A type system contains all these types discovered during the process of analyzing a sampling page and it contains utility methods for testing a given data item that belongs to which type, etc. A type system characterizes the regularity (alignment and format) of HTML codes around instances (book instances, car instances, etc) in Web documents. Such type system can be discovered and maintained by the analysis of a single sampling Web document from a page class. This is done only once, for instances annotation or extraction in Web documents from the same page class, we can reuse this type system.

2. Second, by depth-first walking through the DOM tree representation of an HTML document and according to the type system generated in the first step, we can build a structure partition tree that contains partitioned information and type information of the original DOM tree. A typical partition will contain all the information for an instance (i.e., a record of the original database) associated with type information.

3. Third, wrapper is generated by selecting types for a record (instance) and labeling those types. Several heuristic rules or systematic ways can be used to do this work. It is also possible to let users label and select types on demand when heuristic rules fail.

This chapter is organized as follows. Section 3.1 presents an overview of
the approach. Especially, how type system characterizes the regularity of the HTML codes around instances. Section 3.2 mentions some related work. Section 3.3 and Section 3.4 present the detail of type system generation algorithms. Section 3.5 discusses the heuristic rules for labeling the types. Section 3.6 shows the experiment results. Section 3.7 summarizes and concludes this chapter.

3.1 System Overview

Currently, most commercial web pages, especially those pages that carry lots of records, are maintained by content management systems. Such HTML documents are created by populating templates from a back end database. Therefore, their formats (i.e., layouts) are regular. Such pages often carry multiple instances that follow a particular alignment and format. Moreover, the structure of embedded instances may appear repeatedly if the HTML page contains more than one instance of a particular concept (e.g., book).

Through observation, the regularity characteristic of a page class can be summarized as following:

1. *Similar alignment*: We can see a Web document in Figure 3.1 that continuously lists multiple book instances, and those book instances share similar alignment and format. For example, each book instance has a title followed by the information about its authors, format, price, etc.

2. *Repeated tags sequence*: Since multiple instances of books are continuously listed in these pages, the HTML tags that enclose the book instances appear repeatedly.

Therefore, the basic idea of our approach is to induce wrappers by examining repeated HTML tag sequences in the instances-rich sections of the web pages. However, the difficulty in identifying repeated HTML tag sequences is that the HTML-tag structures of the instances are not necessarily exactly the same. The reason is that the original database column may contain null values for some attributes or the returned dataset is fetched from several database tables, which makes some attributes contain multiple values. For example, some book instances may only have one author, while others may have several authors.
Therefore, we need a mechanism to discover not only those data objects with a fixed number of attributes and values (plain structure), but also those with a variable number of attributes and values (nested structure).

The approach to extract structured data objects from Web documents is based on the above two observations. This idea gets clearer if we present a HTML page in tree structure. For example, Figure 3.2 shows a partial DOM tree representation of the HTML document in Figure 3.1. All branch nodes in the DOM tree represent HTML tags inside HTML documents and the value beside the tag name is the index of this tag node among all the children of its parent node (i.e., among all its siblings).

![Figure 3.2: DOM tree representation of Amazon page.](image)

From this figure we can see that if we don’t care about the index beside the tag name, all product names have the same path from the root to the leaves:

```
table\tbody\tr\td\ldots\tr\td\b\a\b
```

In the same manner, all author names share the same path from root to the leaves:

```
table\tbody\tr\td\ldots\tr\td\span
```
Furthermore, we can see that because multiple instances of products are continuously listed in this pages, the HTML tags that enclose the product instances appear repeatedly.

In order to detect and maintain such a regular pattern for subsequent Web documents in the same page class, we propose a type system to capture this consistency of spatial locality of semantically related items (i.e., attributes of an instance). Formally, we give the definition of a type system as follows:

First, the definition of a tag is given below. A tag is actually an HTML tag, but has some additional information (i.e., the index property).

**Definition 3 Tag**

*A tag is used to represent a HTML tag with the following properties,*

*tagName* *the name of the tag, such as* `<table>`, `<tr>`, or `<td>`, *etc.*

*index* *the relative index among the children of its corresponding HTML tag’s parent. It is also possible to use* $\alpha$ *to denote an arbitrary index. Such tag is called* GeneralTag. *

For example, in Figure 3.2 there are four *tr* underneath *tbody* tag. All *tr*s have the same tag name (i.e. *tr*), but have different index (i.e., 1, 3, 7, 9 for each of them respectively). So these four *tr*s are actually four *Tag* instances in our definition. But these four *tr*s can also be generalized as *GeneralTag* with an arbitrary index $\alpha$.

**Definition 4 Type**

*Given a DOM tree,*

*Primitive type* *Let* $T_k$ *be a path of the DOM tree from the root to a leaf node. $T_k$ contains a sequence of Tags. Such* $T_k$ *is called a primitive type. A primitive type must contain one and only one general tag.*

*Composite type* *If* $T_1,T_2,T_3,\ldots,T_k$ *are types, then an ordered list*

\[<T_1,T_2,T_3,\ldots,T_k>\]

*is called composite type.*
No type

No type is a kind of type that represents noise or an irregular text item in DOM tree.

Intuitively, a primitive type encodes the presentation style (including location and visual cues such as font type and size) of a piece of text that corresponds to a leaf node in a DOM tree (i.e., the first observation in the beginning of this section). For example, in Figure 3.2, all titles, “Effective Java Programming Language Guide”, “Developing Games in Java”, “Microsoft Visual Studio .NET Professional”, etc., have the same primitive type. Let’s denote it as type $T_1$. The same holds for author, let’s denote the type as $T_2$. Later, we can assign labels “product (book) name” to $T_1$ and “author” to $T_2$. The XPath clustering algorithm described in Section 3.3 is subject to generate primitive types.

A compound type essentially summarizes the structural recurrence information at a subtree rooted at an internal node (i.e., the second observation in the beginning of this section). Continuing the above example, we see $T_1T_2\ldots$ repeats for several times depending on how many products are inside this Web document. Section 3.4 describes how to generate a compound type.

Some attributes of a record can have multiple values and some attributes of a record can be of null value. This introduces some “noise” into HTML code and puts additional burden for us to detect regularity. We say “noise” here, it doesn’t mean that Text nodes that classify as noise are real noise that can be skipped during the extraction process. We use “noise” because it makes it difficult to discover sequence patterns (i.e., compound type). We will come to this issue in Section 3.4.

Based on the type system that we generated from a sampling HTML page, we can construct a structure partition tree for all remaining Web documents in the same page class as the sampling. The structure partition tree presented in Figure 3.4 is another DOM tree. Every element (nodes) in the tree has a type information. After the structure partition tree has been generated, we use some heuristic rules to determine instance partitions\(^1\). For example, in Figure 3.4, all product instances are under the element named “SPTP” with type “CT27”. All “CT27” are collapsed together into a new element with name “SPTP” with type “CT27” and with “recordPartition” attribute set to “true”. Some parti-

\(^1\)A partition that contains all properties for a instance is called instance partition.
tions such as “CT24” are sub-partitions of an instance partition with consistent information (e.g., “CT24” always contains title, author, format information, which give us a chance here to detect attribute with multiple values). After labeling some element (type), we can generate a wrapper for instance extraction. This wrapper is based on type information. For example, “PT1” contains title for product instance, “PT2” contains author information for product instance, “PT3” contains format information for product instance, etc.

Figure 3.3 shows the UML diagram of the type system.

We end this section by summarizing the automated wrapper generation work flow as following:

- In the type generation phase:
  1. Given a sampling page, generate the type system.
  2. For the type system generated in first step, generate the structure partition tree.
3. Select the instance partition and label elements in the instance partition with type information.
4. Generate the wrapper based on the above information.

- In the instances extraction phase:
  1. Given a subsequent Web document in the same page class as the sampling page, generate the structure partition tree according to the type system.
  2. Using the wrapper for instances extraction.

### 3.2 Related Work

Currently, most works on wrapper inducing for Web pages require that a programmer understands the specific presentation layout or contents of the web pages. Furthermore, since even a small change to a Web page may prevent the wrappers from working correctly, labor-intensive wrapper maintenance is often required. Therefore, how to automatically generate or induce the delimiter-based extraction rules to extract data objects from dynamically generated web pages has become an important research problem.

Early approaches to generate wrappers for web sites were mainly based on hand coding, which needs experts to analyze each data source, extract its specific data object structures and manually construct wrappers. This approach is not scalable and is expensive to maintain, especially for frequently changing web sites.

Crescenzi et al. [4] develop a wrapper induction system that generates a wrapper based on a comparison of the similarities and differences between web pages. Once a mismatch is found when comparing two web pages, they try to resolve it by generalizing the wrapper to have an optional sub-expression or a nested sub-expression. Therefore, this approach can identify nested structures in an HTML page. However, this approach requires at least two pages for comparison.

The approach presented in this report is inspired by [15] and [9]. Wang et al. [15] propose a system that uses data-rich extraction (DSE) to identify the
Figure 3.4: Structure Partition Tree of Amazon page.
data-rich section of HTML pages and uses C-repeated pattern to identify plain or repeated nested data structured in HTML pages. A web page is considered as a token sequence and repeated substrings are discovered from it by building a token suffix-tree from the token sequence. By iteratively applying the extraction process, a pattern-tree is built, which represents the hierarchical relationship between discovered patterns, and a (regular expression) wrapper is obtained that is used to extract both plain- and nested-structured data from HTML pages. But this system does not use explicit type system for noise elimination.

Mukherjee et al [9] build structure partition tree for HTML pages based on a type system nearly the same as our work. The main difference is that they use semantic analysis including some heuristic rules to solve the “noise” problem, but it seems to be too complex and it is likely to fail. We solve the “noise” problem during the generation of the structure tree by information exposed from the tree structure.

3.3 Primitive Type Generation with XPath Clustering Algorithms

3.3.1 XPath clustering algorithm

XPath [23] is a standard, confirmed by the World Wide Web Consortium (W3C) as a common language to access XML documents in tree representation. The XPath specification defines how a specific item within an XML document can be located. This is accomplished through referencing specific nodes in the XML documents; here “node” refers to any piece of XML data, including elements, attributes, or textual data. In XPath specification, an XML document is considered a tree of these nodes, where each node can be accessed by specifying the location in the tree at which it is located. In one word, a XPath can be used to quickly locate some node(s) in DOM tree.

We may use some special XPath to query the corresponding Text node (Text node is a jargon in DOM [22] to present textual data). On the contrary, if we already have the reference of a Text node, it is very easy to generate the corresponding XPath for this node by recursively looking for its parent and
ancestors and concatenating them together. As discussed in the beginning of this chapter, we need to find out the spatial locality of semantically related text nodes. So it is very intuitive to think about the similarity among XPaths of text nodes. For example, Table 3.1 shows some Text nodes in the DOM tree of Figure 3.2 with corresponding XPaths. We can see semantically related items, such as product title—“Effect...” and “Microsoft...”, these two nodes’ corresponding XPaths differ in only one number, one is 1 and another is 7. If we replace this difference number by an arbitrary index number $\alpha$, the following generalized XPath can be used to group such semantically related items into together. Returning to our definition, we use a primitive type to denote a generalized XPath. In programming detail, we use a sequence of tags (instances of Tag class) with one instance of GeneralTag to represent a primitive type.

$$\text{string}(/html/body/table[2]/tbody/tr/td/table[2]/tbody/tr[\alpha]/td/b[1]/a/b)$$

To generate the primitive type defined above, an XPath Clustering algorithm

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Content</th>
<th>XPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Effect...</td>
<td>string(/html/body/table[2]/tbody/tr/td/table[2]/tbody/tr[2]/td[3]/table[2]/tbody/tr[1]/td[2]/table/body/tr[1]/td/b[1]/a/b)</td>
</tr>
<tr>
<td>T1</td>
<td>Microsoft...</td>
<td>string(/html/body/table[2]/tbody/tr/td/table[2]/tbody/tr[2]/td[3]/table[2]/tbody/tr[7]/td[2]/table/body/tr[1]/td/b[1]/a/b)</td>
</tr>
</tbody>
</table>

Table 3.1: Sample XPaths for given Text node.
Table 3.2: Properties of XPath Cluster.

<table>
<thead>
<tr>
<th>Name</th>
<th>Differ Position</th>
<th>Center Path</th>
</tr>
</thead>
</table>

is presented in Section 3.3.1. An XPath cluster is used to group Text nodes of a HTML page with similar path structures together. More precisely, an XPath cluster is defined as follows:

**Definition 5 Primary same**

Two XPaths are primary same, if and only if

1. two sequences of tags are exactly the same, and
2. two sequence of tags must have one and only one index on the same position different.

**Definition 6 XPath Cluster**

An XPath cluster contains a set of XPaths, whose

1. all XPaths are considered to be similar if and only they are primary same,
2. For every two XPaths in a cluster their difference position are the same.

Table 3.2 shows sample clusters for Figure 3.2. The left most column is the cluster’s name. For another XPath to be group into a cluster, it has to match all sequence tag names and indices (in the third column) except the index of the GeneralTag (indicated by a differ position in the second column). We can define that one cluster is one instance of Primitive Type. They have a one-to-one association relationship. So if two Text nodes belong to the same cluster, they have the same type information.
3.3.2 Implementation of the XPath clustering algorithm

Figure 3.5 shows the UML diagram for implementation of the XPath clustering algorithm.

![UML diagram](image)

Figure 3.5: The UML diagram for the implementation of the XPath clustering algorithm.

The code in Figure 3.6—`shouldGroup()` method—checks whether two XPaths are primary same. If they are primary same, then it finds out the diff position and returns it, otherwise it returns -1.

The code in Figure 3.7—`createClusters()` method—decides whether to create a cluster based on the criteria of two XPaths. If two paths are primary same, it will look into already created clusters. If there is a cluster that (1) has the same diff position as two XPaths’ diff position, (2) already contains two paths or its center path is exactly the same as the center path from input parameters, then the `found` flag is set to true. Otherwise, a new cluster is created. If two paths are not primary same, this method simply returns and does nothing.

Finally, the algorithm presented in Figure 3.8 gathers all Text nodes in a DOM tree (line 1, all information we are interested inside a Web document is such textual data) and compares each pair of nodes to create clusters in line 6. The input parameter `minArea` is the minimum number of nodes that could be inside a cluster for a given page. This parameter is used to strip off some noise textual data inside the page so that the algorithm presented later in this chapter can better capture the regularity of Web documents.
Procedure ShouldGroup(path1, path2)
input
    path1: A given XPath
    path2: A given XPath
begin
1. if path1’s length not equal path2’s length
2. return -1;
3. endif
4. diffPos = -1;
5. for (int i = 0; i < path1.size(); i++) do
6. Tag tag1 = (Tag)(path1.get(i));
7. Tag tag2 = (Tag)(path2.get(i));
8. if tag1.getName().equals(tag2.getName())
9.   if (tag1.getIndex() != tag2.getIndex())
10.      if diffPos == -1
11.         diffPos = i;
12.      else
13.         return -1;
14.   endif
15. endif
16. else
17.   return -1;
18. endif
19. endfor
20. if diffPos == -1
21. return -1;
22. endif
23. return diffPos;
end

Figure 3.6: Procedure: ShouldGroup
Procedure CreateClusters(clusters, xpathCenter, center, xpathAround, around)

input
clusters: A given clusters list
xpathCenter: A given XPath
center: A Text node corresponding to xpathCenter
xpathAround: A given XPath
around: A Text node corresponding to xpathAround

begin
1. diffPos = shouldGroup(xpathCenter, xpathAround);
2. if (-1 != diffPos)
3. //whether there already exist a cluster
4. found = false;
5. for (int i = 0; i < clusters.size(); i++) do
6. XPathCluster cluster = (XPathCluster)(clusters.get(i));
7. if (cluster.getDiffIndex() == diffPos)
8. if ((cluster.contains(xpathCenter)) && (cluster.contains(xpathAround)))
9. found = true;
10. break;
11. elsif cluster.getCenterXPath().exactSame(xpathCenter)
12. found = true;
13. cluster.add(xpathAround, around);
14. break;
15. endif
16. endfor
17. endif
18. if (found == false)
19. //create new cluster, and add into clusters list
20. create new XPathCluster(diffPos, xpathCenter, center, xpathAround, around);
21. and add into clusters
22. endif
23. endif // end of shouldGroup
24. end

Figure 3.7: Procedure: CreateClusters
Algorithm $\text{Clustering}(\text{doc}, \text{minArea})$

input
\begin{itemize}
  \item $\text{doc}$: a DOM document
  \item $\text{minArea}$: min area of a cluster
\end{itemize}

begin
1. $\text{nodes} = \text{gatherTextNodes}(\text{doc})$
2. for each node $\text{n}$ in $\text{nodes}$ do
3. \hspace{1em} $\text{nPath} = \text{generateXPath}(\text{n})$
4. \hspace{1em} for each node $\text{m} = \text{n} + 1$ in $\text{nodes}$ do
5. \hspace{2em} $\text{mPath} = \text{generateXPath}(\text{m})$
6. \hspace{2em} CreateClusters($\text{clusters}$, $\text{nPath}$, $\text{n}$, $\text{mPath}$, $\text{m}$)
7. \hspace{1em} endfor
8. endfor
end

Figure 3.8: Algorithm: Compare each pair of two nodes and to create clusters

3.3.3 Discussion

XPath Clustering is used to generate \textbf{Primitive types}, which is part of the type system. Once generated, it is fixed and used for subsequent Web documents in the same page class as the sampling page. Subsequent Web documents are analyzed as follows: (1) given a new document, recursively walk through the DOM tree and gather Text nodes; (2) by using the type system, we can determine which type a given Text node belongs to. In other words, based on already generated primitive types of the type system, we can assign Text nodes of a new Web document primitive type information.

To improve the quality of primitive types, we must select a Web document that contains as many instances as possible. For every instance, it is better to have as many values as possible for each attribute. This would give us more chances to discover nested data types (i.e., an attribute contains multiple values.) For future development, we can consider to use additional sampling pages to improve the quality of primitive type generation.

The time complexity for XPath clustering algorithm is $O((\#\text{text node})^2)$. But this time consuming task is only performed in wrapper generation stage. For instances extraction based on already generated wrapper, this task will not needed.
3.4 Structure Partition Tree

With the XPath Clustering algorithm, we are able to gather semantically related items into one cluster, such that “Effect...” and “Microsoft...” are instances of product title, “by Joshua Bloch” and “by Microsoft” are instances of author or publisher name. But one important information is still missing—instances relationship. How can we identify that the book “Effect Java Programming Language Guide” is written by “Joshua Bloch” with price “$27.99”? How can we identify that the product “Microsoft Visual Studio .Net Professional” is published by “Microsoft” with price “$260”? To identify an instance of a particular concept (e.g., book) and identify its related properties such as title, author, and price etc is essential to bootstrap the ontology. In this section, we propose an approach to capture instances relationships described above.

Again we need the help from XPath. The essential is to find groups of Text nodes with similar structure (i.e. a record) by looking for patterns in the occurrences of the XPaths of these Text nodes. Recall that in the last section, instances that belong to the same concept have been grouped together called primitive type. Figure 3.9 shows the DOM tree structure of an Amazon search page augmented with primitive type (square filled with blue color).

We can see that the primitive types $T_1, T_2, T_3, T_4$ together form an instance, where $T_1$ stands for book (product) title, $T_2$ stands for author (publisher) name, $T_3$ stands for “Buy new” price, $T_4$ stands for “Used & New” price. $T_3$ is an optional property for Microsoft Visual Studio product because it is missing in the Web page.

Now the problem of “finding groups of Text nodes with similar structure” can be reduced to the problem of sequential pattern analysis among type sequences. In our case, how can we discover the pattern $T_1 T_2 (T_3?) T_4$ in the sequence $T_1 T_2 T_3 T_4 T_1 T_2 T_3 T_4 T_1 T_2 T_3 T_4$.

3.4.1 Structure Partition Tree algorithm

Before we start to deal with the Structure Partition Tree algorithm, we first point out what challenges we are going to face at this point:

1. Detect sequential patterns among type sequences.
2. Deal with optional fields.

The Structure Partition Tree algorithms proposed in this section solves these two challenges.

Find Longest Repeated Subtypes

To solve the first challenge, i.e., detect sequential patterns among type sequences, we can use ideas from string search algorithms [18]. If we assume the type system as a finite alphabet, that is, each type instance is a distinct character inside the alphabet, we can apply all string search algorithms on type system. Among all string search algorithms, the most appropriate algorithm that is suitable for sequential pattern detection is to find Non-overlapped Longest Maximal Repeated Substring (in our case, it would be Non-overlapped Longest Maximal Repeated Subtypes). The definitions below are written in a step by step manner following the book [18].

Definition 7 Repeated Substring [18]
Given an input string $S$, a **repeated substring (pattern)** of $S$ is one having at least two matching occurrences at distinct positions within $S$, with the possibility that such occurrences may overlap.

**Definition 8 Maximal Repeated Substring** [18]
Given an input string $S$, a **maximal repeated substring** of $S$ is a repeated substring of $S$ that can not be extended further in the left and right direction.

**Definition 9 Longest Maximal Repeated Substring** [18]
Given an input string $S$, a **longest maximal repeated substring** of $S$ is a maximal repeated substring that contains largest number of characters. It has the following properties: (1)occurring at two or more distinct positions in $S$, (2)can not be extended further in the left and right direction,(3)possibly overlapping with itself.

**Definition 10 Non-overlapped Longest Maximal Repeated Substring**
Given an input string $S$, a **non-overlapped longest maximal repeated substring** of $S$ is a longest maximal repeated substring without overlapping.

For example, the string $PABCQRABCSABTU$ has $A$, $B$, $C$, $AB$, $BC$ and $ABC$ as its **repeated substrings**. Substrings $AB$ and $ABC$ are **maximal repeated substrings**, and $ABC$ is thus the **longest maximal repeated substrings**. Another example, conside string $ABCABCAD$, $ABCA$ is a **longest maximal repeated substring** with overlapped occurrences $(1, 4)$ and $(4, 7)$, and $ABC$ is a **non-overlapped longest maximal repeated substring** with adjacent occurrences $(1, 3)$ and $(4, 6)$. The above example shows the differences between definitions 5 to 8. In fact, we believe that Non-overlapped Longest Maximal Repeated Substring is more suitable for Web information extraction, especially, sequential pattern detection in type system.

To find such non-overlapped longest maximal repeated substrings, we can use a data structure called suffix tree (see Definition 11). A suffix Tree is a data structure that allows many problems on strings (sequences of characters) to be solved in linear time. When classical pattern matching algorithms are used to search for several patterns $P_1, P_2, \ldots, P_k$ in the string $S$, $O(|P_1| + |P_2| + \ldots + |P_k| + k|S|)$ time is taken. The suffix tree data structure that we are about to study reduces this complexity to $O(|P_1| + |P_2| + \ldots + |P_k| + |S|)$. 

52
Definition 11. A suffix tree $T$ for an $m$-character string $S$ is a rooted directed tree with exactly $m$ leaves numbered 1 to $m$. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of $S$. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf $i$, the concatenation of the edge-labels on the path from the root to leaf $i$ exactly spells out the suffix of $S$ that starts at position $i$. That is, it spells out $S[i..m]$.

Suffix Trees can be used to solve a large number of string problems that occur in text-editing, free-text search, computational biology, and other application areas. One important application for suffix trees is finding Longest maximal repeated substring.

Add a special “end of string” character, e.g. `$`, to $txt[1..n]$ and build a suffix tree; the maximal repeated substring of $txt[1..n]$ is indicated by the deepest fork node in the suffix tree, where depth is measured by the number of characters traversed from the root. The maximal repeated substring can be found in $O(n)$ time using a suffix tree.

The longest substrings represented by an internal node in the suffix tree $Y$ are the longest repeated substrings of the input string $y$. It therefore suffices to keep track of the maximum length of the internal-node substrings as the tree is being built, which may be accomplished in linear time.

Find subtypes in composite types level

We apply the algorithms to find non-overlapped longest maximal repeated substring finding on sequence of types. This raises two problems:

1. Text nodes without type (NoType). This is typical for attributes that contain multiple values or optional attributes for some instance. Since its locality recurrence is relatively small, it can to be dropped during the process of finding primitive type (XPath Clustering algorithm) by the parameter $min.Area$. These text node is typed as NoType. However, we can not drop NoType nodes when we infer all properties of instances.
2. Text nodes with **NoType** as their type information will introduce “noise” inside type sequence. It makes the algorithm difficult to detect sequence pattern and may even result in errors. For example, consider the following type sequence:

\[ T_1, T_2, T_3, T_4, T_1, T_2, T_3, T_4, T_1, T_2, T_3, T_4 \]

We assume that \( T_1, T_2, T_3, T_4 \) compose an instance record, \( T_4 \) is an optional field. The algorithm will detect \( T_1, T_2, T_3, T_4 \) as repeated pattern, which is correct. But how to detect that is \( T_1, T_2, T_3 \) is also an instance record with the optional field \( T_4 \) missing. Even worse, if we consider this sequence:

\[ T_1, T_2, T_3, T_4, T_1, T_2, T_4, T_1, T_2, T_3 \]

Here \( T_3 \) and \( T_4 \) are both optional fields. The algorithm will detect \( T_1, T_2 \) as repeated pattern, which is wrong.

In order to deal with the above two points that might introduce errors in the pattern discovery process, we perform type sequence pattern detection on composite types level. The algorithm processes a DOM tree in depth-first manner, it will eventually reach leaf nodes—Text Nodes. When it sees an Element node whose children are leave nodes, it collapses them together and forms a new element. This element has a composite type whose subtypes are the children’s types. We describe this procedure in more detail in the next section.

**Algorithm process**

Figure 3.10 shows the algorithm to build the structure partition tree. The structure Partition Tree algorithm first processes a DOM tree in depth-first manner and then discovers sequential patterns recursively bottom-up, at the same time a new DOM\(^3\) tree structure is generated in bottom-up manner. Such a tree structure contains partitioned information of the original DOM tree.

Let us now walk through the whole structure partition tree building process, which is illustrated in Figure 3.12. The algorithm starts with the `buildSPT()` method and visits the DOM tree in depth-first manner. When it sees an Element node, it goes one level deeper by invoking the `buildSPT()` method recursively.

\(^3\)In our implementation, this new DOM tree uses JDOM [25] representation.
Algorithm BuildSPT()
input
n: a Node in DOM tree
begin
1. SPTElement sptElem = null;
2. if (n is Text Node)
3. String data = n.getData();
4. Type type = typeSys.getType(n);
5. sptElem = new SPTElement(‘SPTC’,true,type);
6. sptElem.setData(data);
7. 
8. elsif (n is Element Node)
9. NodeList children = n.getChildNodes();
10. for (int i = 0; i < children.getLength(); i++)
11. Node tempNode = children.item(i);
12. SPTElement result = buildSPT(tempNode);
13. if (null != result)
14. results.add(result);
15. endif
16. endfor
17. if (results.size() == 1)
18. sptElem = (SPTElement) (results.get(0));
19. elsif (results.size() > 1)
20. sptElem = analysis(results);
21. endif
22. endif
23. return sptElem;
end

Figure 3.10: Algorithm: build the Structure Partition Tree
Algorithm Analysis()
input
  sptElements: a list of SPTElement
begin
  1. sptElements = replaceFlatten(sptElements);
  2. sptElements = collapse(sptElements);
  3. List typesList = getTypesFromList(sptElements);
  4. List sublist = discover sequence pattern by typesList
  5. while (null != sublist) and (sublist.size() > 1)
  6.    presentList = sublist;
  7.    sptElements = merge(sptElements, sublist);
  8.    sptElements = collapse(sptElements);
  9.    typesList = getTypesFromList(sptElements);
 10. sublist = discover sequence pattern by typesList
endWhile
12.
13. SPTElement sptParent;
14. if (sptElements.size() == 1) // means collapse performed.
15.   sptParent = (SPTElement)(sptElements.get(0));
16. else
17.   if (null != presentList)
18.     Type type = typeSys.getType(presentList);
19.     sptParent = new SPTElement('SPT', false, type);
20. else
21.     sptParent = new SPTElement('SPT', false, typeSys.getFlattenType());
22. endif
23. endif
24. for (int i = 0; i < sptElements.size(); i++)
25.   SPTElement e = (SPTElement) (sptElements.get(i));
26.   sptParent.addContent(e);
27. endfor
28. return sptParent;
end

Figure 3.11: Algorithm: Discover sequence pattern
When it sees a Text node, it returns an SPTElement with content of the Text node’s content and the recursive process jumps one level up.

Figure 3.12: Structure Partition Tree building process.

At point 1(td), two SPTElement child\(_1\) and child\(_2\) are created with the content “Effect...” and “by Joshua...”. These two SPTElements go into analysis method (Figure 3.11) and try to find the repeated sequence pattern. Because their types are both Primitive Types, they are collapsed into a Composite Type (\(T_{12}\)) with subtype sequence \(T_1T_2\) (line 2 of Figure 3.11). At the same time, it creates a parent SPTElement parent\(_1\) with type \(T_{12}\). In the same manner, process will create SPTElements parent\(_{i}(i=2,\ldots,8)\), parent\(_{i}(i=1,3,5,7)\) have the same type \(T_{12}\) because their subtype sequences are the same—\(T_1T_2\). parent\(_8\) has the subtype \(T_4\), but parent\(_6\) also have the same type \(T_{34}\) as parent\(_{i}(i=2,4,8)\). Why?

This is the way how we deal with optional fields: before the algorithm wants to create a new composite type, it will first iterate all already created composite types and see whether there is a composite type whose sub-types contains current sub-types.
At point 9, two previously created SPTElement $parent_1$ and $parent_2$ go into the $analysis()$ method. In the $analysis()$ method, because $parent_1$ and $parent_2$’s type sequence is $T_{12}T_{34}$, so there is no repeated sequence pattern found. When no repeated pattern found, the algorithm creates a parent element SPTElement called $parent_{a1}$ whose “flatten” flag is set to true (line 22 of Figure 3.11) and whose content is $parent_1$ and $parent_2$. In the same manner, $parent_{a2}$, $parent_{a3}$, $parent_{a4}$ are created at points 10,11,12 respectively. All these SPTElements have flatten flag set to true.

Until point 13, $parent_{a1}$,$parent_{a2}$,$parent_{a3}$, $parent_{a4}$ go into $analysis()$ method. Because their flatten flags are true, they are replaced by their children (line 1 of Figure 3.11), which are $parent_i (i = 2,\ldots,8)$. The type sequence is the following:

$$T_{12}T_{34}T_{12}T_{34}T_{12}T_{34}T_{12}T_{34}$$

Now we are able to find a repeated pattern. In this type list, $T_{12}T_{34}$ is the longest repeated substring. So after the first iteration of the analysis method, the original list becomes:

$$T_5T_5T_5$$

$T_5$ is a composite type of $T_{12}T_{34}$, which is actually a record in semantical meaning. At this point, the whole tree is partitioned.

### 3.5 Label discovery

To assign labels to the types for the structure partition tree, i.e., to understand the meaning of the data attributes, we employ the following four heuristics:

1. Match form element labels to data attributes. The search form of a web site through which users submit their queries provides a sketch of the underlying relational database of the web site. If we make the assumption that the web site designers try their best to answer user queries with the most relevant data, keyword queries submitted through one specific form element will re-appear in the corresponding attribute values of the data objects. For each form element with its keyword queries, if the keywords mostly appear in one specific column of the data table, then we assign the label of that form element to the column.
2. Search for voluntary labels in table headers. The HTML specification defines some tags such as `<TH>` and `<THEAD>` for page designers to voluntarily list the heading for the columns of their HTML tables. Moreover, those labels are usually placed nearby the data objects. Therefore, the HTML code near (usually on the top of or on the left of) the contained data objects is examined for possible voluntary labels.

3. Search for voluntary labels encoded together with data attributes. Some web sites encode the labels of data attributes together with the attribute values. Therefore, for each column of the data table we try to find the maximal-prefix and maximal-suffix shared by all cells of the column and assign the meaningful prefix to that column and the meaningful suffix to the column next to that column as the labels.

4. Label data attributes in conventional formats. Some data have a conventional format, e.g., a date is usually organized as “dd-mm-yy”, “dd/mm/yy”, etc., email usually has the symbol “@”, price usually has a currency symbol “$”, etc. Thus, such information is used to recognize the corresponding data attributes. Note that the form elements and the data attributes do not need to be perfectly matched. Therefore, the label assigner may not be able to assign meaningful labels to all of the data attributes.

### 3.6 Experiments

<table>
<thead>
<tr>
<th>Web sites</th>
<th># Pages</th>
<th># Type System</th>
<th>Precision(avg)</th>
<th>Recall(avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mysimon.com</td>
<td>10</td>
<td>1</td>
<td>100</td>
<td>88</td>
</tr>
<tr>
<td>AltaVista</td>
<td>10</td>
<td>1</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>Excite!</td>
<td>10</td>
<td>2</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Monster.com</td>
<td>10</td>
<td>1</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Amazon</td>
<td>10</td>
<td>1</td>
<td>80</td>
<td>70</td>
</tr>
<tr>
<td>MediaMarkt</td>
<td>10</td>
<td>1</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Jobs.com</td>
<td>10</td>
<td>1</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Ebay</td>
<td>10</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.3: Performance of SLIE system.

Figure 3.14 shows the inferred table from a HTML page of jobs.com in Fig.
ure 3.13. We see that all records about jobs are not listed as rigid as underlying database tables. So the approach described in Chapter 2 will not work. But SLIE described in this chapter can successfully extract instances (jobs). We also see that labels “Industry”, “Salary”, “Location” are successfully discovered. Of course, labels like “Date”, “State”, “Job summary” etc are not discovered.

We conducted experiments on 8 different input collections of pages. The pages in these collections are results of queries from well known search engines for books, jobs etc. The results are shown in Table 3.3. We measure the precision and recall using the following formula. The fourth and fifth columns in the table...
are average precision and recall respectively. The third column is the number of type system found for one Web site. For example, sample pages from Excite found two type systems, which leads to two wrappers.

\[
\text{Precision} = \frac{\# \text{ relevant partitions among all extracted partitions}}{\text{all extracted partitions}} \quad (3.1)
\]

\[
\text{Recall} = \frac{\# \text{ relevant partitions among all extracted partitions}}{\text{all relevant records of a Web page}} \quad (3.2)
\]

From Table 3.3, we see the system does not work for EBay Web site. This is because EBay presents their products in JavaScript arrays, and dynamically load them into Web pages in client side. When we checked the downloaded
Web page from EBay, we see many Javascript arrays. But our system is based on DOM tree structure that is used for both XPath clustering and Structure Partition Tree, so our system will not work in such situation.

3.7 Summary

In this chapter, we describe the SLIE approach to extract concepts, instances, and relationship among instances from Web pages without rigid tables. This approach is based on the observation that semantically related items (i.e., properties of instances) exhibit regularity in spatial locality. This approach can be operated on a single Web page or on a sequence of Web pages from the same page class.

Semantically labeling discovered instances is not well studid in this theis. Future development may use an already existing ontology, or systematic methods such as Hidden Markov Models to help labeling.
Chapter 4

Ontology database

There are two ways to talk to the ontology database. One is the direct way that is through the ontology database API (Section 4.1). Another way is though a standard Ontology language, such as DAML+OIL (Section 4.2).

4.1 Ontology database API

Figure 4.1: A high level view of the Hibernate architecture
The implementation of the Ontology database uses Hibernate [26] for its persistent layer. “Hibernate is a powerful, ultra-high performance object/relational persistence and query service for Java. Hibernate lets you develop persistent objects following common Java idioms - including association, inheritance, polymorphism, composition and the Java collections framework. Extremely fine-grained, richly typed object models are possible.” Hibernate also provides a so-called the Hibernate Query Language (HQL), which is full object-oriented, understanding notions like inheritance, polymorphism and association. Hibernate is now the most popular ORM solution for Java. Figure 4.1 shows a high level view of the Hibernate architecture.

4.1.1 Data model

![Diagram of Data model](image)

Figure 4.2: Data model

Database schemas are listed following:

```sql
CREATE TABLE html_property_value_pair (  
    pair_id BIGINT NOT NULL AUTO_INCREMENT,  
    property VARCHAR(255),  
    value VARCHAR(255),  
    record_fk BIGINT NOT NULL REFERENCES html_table_record(record_id),  
    PRIMARY KEY (pair_id)  
);
```

```sql
CREATE TABLE html_table_record (  
    record_id BIGINT NOT NULL AUTO_INCREMENT,  
    PRIMARY KEY (record_id)  
);
```
4.1.2 API

The implementation is packaged into a component and provide as Application Programming Interface (API) to other developers. The following lists the functions that the ontology database provides:

1. *Return all concepts inside the ontology database.* Using the following Hibernate query language:

   ```
   select distinct pvp.property from org.unisb.dbs.agirdm.htmlanalz.persist.PropertyValuePair as pvp
   ```

2. *Given a concept, eg:“model”, return all its instances.* Using the following Hibernate query language:

   ```
   select distinct pvp.value from org.unisb.dbs.agirdm.htmlanalz.persist.PropertyValuePair as pvp where (pvp.property=?) and (pvp.value is not null)
   ```

3. *Given a concept and its instance pair, eg:“model=A6”, return all the other concept and instance pairs, that composite a record (instance) eg: “area=NY, make=audi, year=2001” etc.* It is implemented by searching for “property=model and value=A6”. Some pairs will be returned by following HQL query:

   ```
   from org.unisb.dbs.agirdm.htmlanalz.persist.PropertyValuePair pvp where (pvp.property=?) and (pvp.value=?)
   ```

   For the pairs returned by above query, get the records. And for every record, lists all its property-value pairs.

4.2 Ontology language

There are many way to written down an ontology. In order to support a variety of common styles of use, we uses Jena [32] ontology API to write ontology using DAML+OIL language. Jena is, at heart, an RDF platform, we restrict ourselves
to ontology formalisms built on top of RDF. Specifically this means RDFS, the varieties of OWL and DAML+OIL. Figure 4.3 shows a sample car ontology.

![Sample ontology model for Car domain.](image)

Figure 4.3: Sample ontology model for *Car* domain.

### 4.3 Ontology navigation GUI

We also develop a simple ontology navigation GUI (Figure 4.4) based on a well known graph library—“touchgraph”. that allows the user to navigate in ontology database among instances and concepts in the following way: (1) first, the user submits a concept, for example “make” in a “car” ontology, GUI will draw all instances that belong to this concept, such as “nissan”, etc. (2) When the user clicks on the “nissan” node, it will draw all “records” (i.e., instances of “car”) with “nissan” as “make”. (3) When the user clicks on one record node, this node will be expanded and all properties about a ”car” instance will be drawed, such as “area” value, “model” value, “mileage” value etc.
Figure 4.4: Sample GUI for search in ontology database.
Chapter 5

Summary

In this thesis, we have explored the way to bootstrap ontology from Web documents. The premise is that the Internet is an enormous information resource, but lacks mechanism that intelligent software can be able to effectively, accurately access and process it. The work presented in this thesis—building ontologies with rich instances inside, provides the foundation for intelligent softwares that can perform complex logic computation on it and do arbitrary sophisticated tasks.

The challenge work in building such ontologies is to correctly identify concepts and corresponding instances from Web documents with flexible HTML syntax. We have proposed two approaches to extract instances and concepts.

The first approach is an automated table understanding process. It is based on the assumption that list of records such as product description, addresses, job advertisements are presented in rigid HTML table format.

The second approach is much more flexible in the sense that it can perform extraction work on the list of records that needn’t to be presented in rigid table format. This approach is based on the observation: records from the same Web site exhibit consistent and regular spatial locality inside the HTML pages of this Web site.

Finally, to persist the extracted instances and concepts while keeping their relationship, we can store them directly into relational database by well organized tables, or we can use a state-of-art ontology language such as DAML+OIL.
to describe the ontology model and persist the model into disk or database indi-
directly.
Bibliography


[22] www.w3.org/DOM/DOMTR. Document Object Model, W3C.


[31] http://140.115.156.71/iepad/. IEPAD


Simple HTML Ontology Extensions(SHOE)