

Data Extraction and Label Assignment for Web Databases

Jiying Wang
Computer Science Department
University of Science and Technology
Clear Water Bay, Kowloon
Hong Kong
cswangjy@cs.ust.hk

Frederick H. Lochovsky
Computer Science Department
University of Science and Technology
Clear Water Bay, Kowloon
Hong Kong
fred@cs.ust.hk

Abstract

Many tools have been developed to help users query, extract and integrate data from web pages generated dynamically from databases, i.e., from the Hidden Web. A key prerequisite for such tools is to obtain the schema of the attributes of the retrieved data. In this paper, we describe a system called, *DeLa*, which reconstructs (part of) a “hidden” back-end web database. It does this by sending queries through HTML forms, automatically generating regular expression wrappers to extract data objects from the result pages and restoring the retrieved data into an annotated (labeled) table. The whole process needs no human involvement and proves to be fast (less than one minute for wrapper induction for each site) and accurate (over 90% correctness for data extraction and around 80% correctness for label assignment).

Categories & Subject Descriptors

H.3.3 [Information Systems]: Information Search and Retrieval;
H.5.4 [Hypertext/Hypermedia]: architectures

General Terms

Algorithms, Measurement, Experimentation

Keywords

Hidden Web, Web information extraction, HTML forms, Automatic wrapper induction, Information integration, Data annotation

1. Introduction

While search engines provide some help in locating information of interest to users on the World Wide Web, a large number of the web pages returned by filling in search forms are not indexable by most search engines today as they are generated dynamically by querying a back-end (relational or object-relational) database. The set of such web pages, referred to as the Deep Web [3] or Hidden Web [15], is estimated to be around 500 times the size of the “surface web” [3].

Consider, for example, a user who wants to search for information such as configuration and price of a notebook computer before he/she buys on the Web. Since such information only exists in the back-end databases of the various notebook vendors, the user has

to go to the web site of each notebook vendor, send his/her queries, extract the relevant information from the result web pages and compare or integrate the results manually. Therefore, there arises the need for information integration tools that can help users to disseminate the queries, extract the corresponding results from web pages and integrate the retrieved information.

Information integration has been a hot topic for several years [6] and [12] (see [9] and [16] for survey). A lot of work has focused on schema mapping, query reformulation and optimization. However, little attention has been paid to the key prerequisite for information integration, namely, obtaining the schema of every single web site. In the conventional information integration scenario, it is always assumed that the schemas of all web sites are known before the integrated view is defined. Previous research assumed either that each web site cooperatively provides the schema information [6] or that the user can specify the relational schema for each web site [12]. However, it is not guaranteed either that cooperation from web sites is always provided to the integration system or that every user is a database expert and able to specify the schemas.

To integrate data from different web sites, some Information Extraction systems, i.e., wrappers and agents, have been developed to extract data objects from web pages based on their HTML-tag structures [1], [2], [4], [5], [7], [8], [11], [13], [14], [17] and [18]. However, while the extraction of plain-structured data, i.e., data objects with a fixed number of attributes and values, is no problem, few of the current approaches can extract nested-structured data, i.e., data objects with a variable number of attributes and possibly multiple values for each attribute. Moreover, most current work needs human involvement and is deficient in providing users the meaning of the attributes of the extracted data.

In this paper, we address the problem of automatically extracting data objects from a given web site and assigning meaningful labels to the data. In particular, we concentrate on the web sites that provide a complex HTML search form, other than keyword searching, for users to query the back-end databases. Solving this problem will allow the data both to be extracted from such web sites and its schema to be captured, which makes it easier to do further manipulation and integration of the data. This problem is challenging for three reasons. First, the system needs to deal with HTML search forms, which are designed for human use. This makes it difficult for programs to identify all the form elements and submit correct queries. Second, the wrapper generated for each web site needs to be complex enough to extract not only

plain-structured data, but also nested-structured data. Third, the generated wrapper is usually based on the structure of the HTML tags, which may not reflect the real database structure, and the original database field names are generally not encoded in the web pages. In addition, for scalability, the solution to this problem needs to be automatic and fast.

In this paper, we describe the *DeLa* (Data Extraction and Label Assignment) system that sends queries through HTML forms, automatically extracts data objects from the retrieved web pages, fits the extracted data into a table and assigns labels to the attributes of the data objects, i.e., the columns of the table. Our approach is based on two main observations. First, data objects contained in dynamically generated web pages share a common HTML tag structure and they are listed continuously in the web pages. Based on this, we automatically generate regular expression wrappers to extract such data and fit them into a table. Second, the form contained in a web page, through which users submit their queries, provides a sketch of (part of) the underlying relational database of the web site. Based on this, we extract the labels of the HTML form elements and match them to the columns of the data table, thereby annotating the attributes of the extracted data.

The *DeLa* system consists of four components: a form crawler, a wrapper generator, a data aligner and a label assigner. For the first component, we adopt the existing hidden web crawler, *HiWe* [15], to collect the labels of the web site form elements and send queries to the web site. In the remaining three components, we make the following contributions. First, we develop a method to automatically generate regular expression wrappers from data contained in web pages. Second, we employ a new data structure to record the data extracted by the wrappers and develop an algorithm to fill a data table using this structure. Third, we explore the feasibility of heuristic-based automatic data annotation for web databases and evaluate the effectiveness of the proposed heuristics.

The rest of the paper is organized as follows. Section 2 presents the data model and the system overview of *DeLa*, including its four components. Sections 3, 4, and 5 introduce details of, respectively, the wrapper generation, data alignment and label assignment components. Experimental results are provided in Section 6, followed by the related work in Section 7. Finally, Section 8 presents the conclusions and future work.

2. System Overview

Given a web site with a HTML search form, our goal is to send queries through the form, automatically extract data objects from the result web pages, fit the extracted data into a table and assign meaningful labels to the attributes of the data objects (the columns of the table). In this section, we first present the data model employed in this paper and then introduce the system architecture for *DeLa*. We use a simple example to illustrate how the four components of the system work.

2.1 Data model

In this paper, we are only concerned with web sites that generate their web pages by querying the data stored in a back-end database. We use the *nested type* as an abstraction to model the data objects contained in such web pages.

- If O_1, O_2, \dots, O_n are all nested types, then their ordered list $\langle O_1, O_2, \dots, O_n \rangle$ is also a nested type.
- If O is a nested type, then the set, $\{O\}$, is also a nested type.
- If O is a nested type, then $(O)?$ represents the optional type O .
- If O_1 and O_2 are nested type, then $(O_1|O_2)$ represents the disjunction of O_1 and O_2 .

For example, Figure 1 shows a web page of an online bookstore¹ with a search form in its left side for users to query its back-end database of books. If we type a word, such as “Harry Potter”, in the textbox with label “Title”, we get a result page, shown in Figure 2, containing four book objects with “Harry Potter” appearing in their title. In this page, each book has a book title, zero or one author and one or more edition information, e.g., the first book has no author and the third book with the title “A Guide to the Harry Potter Novels” has two editions, a “hardcover” in “Apr, 2002” and a “paperback” in “Apr, 2002”. Therefore, we can model the book data contained in this page as the following nested type:

Book < Title,
(*Author* < Name >)?,
{ *BookEdition* < Format,
Publish Date,
Publisher > } >



Figure 1. An example web site with an HTML search form.

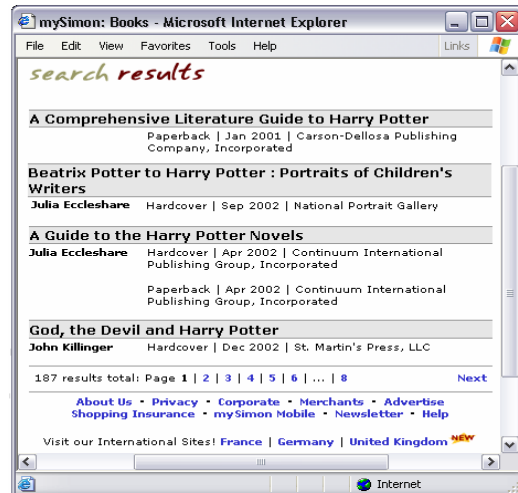


Figure 2. An example query result page.

¹ <http://www.mysimon.com/category/index.jhtml?c=bookisbn>

We consider the data objects contained in web pages as string instances of the implied nested type of its back-end database, where these instances are encoded in HTML tags. Thus, a *regular expression* can be employed to model the HTML-encoded version of the nested type. Given an alphabet of symbols Σ and a special token “text” that is not in Σ , a *regular expression* over Σ is a string over $\Sigma \cup \{text, *, ?, |, (,)\}$ defined as follows:

- The empty string ϵ and all elements of $\Sigma \cup \{text\}$ are regular expressions.
- If A and B are regular expressions, then AB , $(A|B)$ and $(A)^?$ are regular expressions, where $(A|B)$ stands for A or B and $(A)^?$ stands for $(A|\epsilon)$.
- If A is a regular expression, $(A)^*$ is a regular expression, where $(A)^*$ stands for ϵ or A or AA or ...

In Figure 3 we show the HTML code for the first book and the third book contained in the web page in Figure 2 and the corresponding regular expression wrapper to extract book instances from the page.

After we extract data objects from a web page, we choose to rearrange the extracted data from the web page in a table manner such that each row of the table represents a data instance and each column represents a data attribute. We do this so that the data table can be easily translated into a relational schema or an XML DTD for use by other applications. Note that it is not necessary to represent the data in only one table where some attributes are duplicated; we can easily build more tables or relations for multiple-value attributes according to the regular expression wrapper. Here, we choose to unify the data representation of each web site as a single table, because it is easier for the later data integration.

HTML code of the embedded data:

```
<TR><TD><B> A Comprehensive ... </B></TD></TR>
<TR>
  <TD><B> Paperback | Jan 2001 | Carson-Dellosa ... </B></TD>
</TR>
...
<TR><TD><B> A Guide to the ... </B></TD></TR>
<TR>
  <TD><B> Julia Eccleshare </B></TD>
  <TD> Hardcover | Apr 2002 | Continuum ... <BR>
    Paperback | Apr 2002 | Continuum ... <BR> </TD>
</TR>
...
```

Corresponding regular expression wrapper:

```
<TR><TD><B> text </B></TD></TR>
<TR>
  <TD> (<B> text </B>)? </TD>
  <TD> (text <BR>)* </TD>
</TR>
```

Figure 3. HTML code for the example data and the corresponding wrapper.

2.2 System architecture

The system consists of four components as shown in Figure 4.

Form Crawler. Given a web site with a HTML search form, the form crawler collects the labels of each element contained in the form and sends queries through the form elements to obtain the result pages containing data objects. We adopt the hidden web crawler, *HiWe* [15] for this task. *HiWe* was built on the observation that “most forms are usually associated with some

descriptive text to help the user understand the semantics of the element.” It is equipped with a database that stores some values of the task-specific concepts and assigns those values as queries to the form elements if the form labels and the database labels match. Similarly, our system, *DeLa*, further utilizes the descriptive labels of the form elements by matching them to the attributes of the data extracted from the query-result pages. Readers can refer to [15] for more details about the form crawler.

Wrapper Generator. The pages collected by the form crawler are output to the wrapper generator to induce the regular expression wrapper based on the pages’ HTML-tag structures. Since pre-defined templates generate the web pages, the HTML tag-structure enclosing data objects may appear repeatedly if the page contains more than one instance of a data object. Therefore, the wrapper generator first considers the web page as a token sequence composed of HTML tags and a special token “text” representing any text string enclosed by pairs of HTML-tags, then extracts repeated HTML tag substrings from the token sequence and induces a regular expression wrapper from the repeated substrings according to some hierarchical relationships among them. The wrapper generator is inspired by previous work on *IEPAD* [5]. We present in [20] a more technical comparison between our work and theirs.

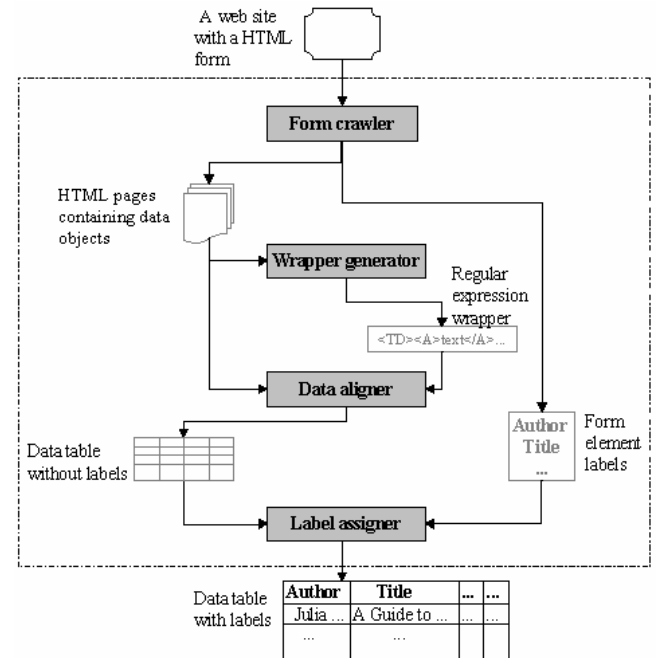
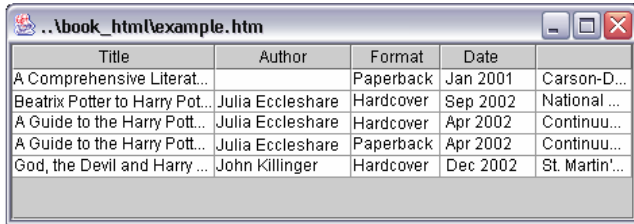


Figure 4. The *DeLa* Architecture.

Data Aligner. Given the induced wrapper and the web pages, the data aligner first extracts data objects from the pages by matching the wrapper with the token sequence of each page. It then filters out the HTML tags and rearranges the data instances into a table similar to the table defined in a relational DBMS, where rows represent data instances and columns represent attributes. Note that the extracted data objects may have optional or multi-valued attributes, e.g., the first book in Figure 2 has no author listed and the third book in Figure 2 has two editions. Furthermore, sometimes several attributes of the data object are encoded together into one text string that is not separated by HTML tags, e.g., the format, publish date and publisher information of the

books in Figure 2. Therefore, the data aligner needs to distribute multiple values of one data attribute into several rows and separate the attributes encoded in one string to several columns, if possible. Figure 5 shows the table representation of the four books contained in Figure 2, where the third book with two editions is re-arranged into two rows (the third and the fourth) and the last three attributes are separated.

Label Assigner. The label assigner is responsible for assigning labels to the data table by matching the form labels obtained by the form crawler to the columns of the table. The basic idea is that the query word submitted through the form elements will probably reappear in the corresponding fields of the data objects, since the web sites usually try their best to provide the most relevant data back to the users. For example in Figure 2, the web page is generated to answer the query “Harry Potter” submitted through the form element labeled by “Title”. Therefore, the first column of the data table in Figure 5, with “Harry Potter” appearing in all five rows, can be marked as “Title”, the label of the form element. Similarly, the second and the third columns are marked as “Author” and “Format”, which also comes from the form element labels in Figure 2. Note that the mappings between form elements and data attributes are usually not exactly one-to-one. The label assigner sometimes needs to employ other information, such as the data format, as clues to understand the semantics of the data objects. For example, in Figure 5, the fourth column is marked as “Date”, since the data it contains are in a date format.



Title	Author	Format	Date	
A Comprehensive Literat...		Paperback	Jan 2001	Carson-D...
Beatrix Potter to Harry Pot...	Julia Eccleshare	Hardcover	Sep 2002	National ...
A Guide to the Harry Pott...	Julia Eccleshare	Hardcover	Apr 2002	Continuu...
A Guide to the Harry Pott...	Julia Eccleshare	Paperback	Apr 2002	Continuu...
God, the Devil and Harry ...	John Killinger	Hardcover	Dec 2002	St. Martin'...

Figure 5. Table representation of the extracted data.

3. Wrapper Generation

In this section, we introduce the techniques employed in the wrapper generator. Readers can refer to [20] for more details.

3.1 Data-rich section extraction

Template-generated commercial web sites usually contain advertisements, navigational panels and so on. Although these parts of a web page may be helpful for user browsing, they can be considered as “noisy data” that may complicate the process of extracting data objects from web pages. When dealing with web pages containing both data objects and “noisy data”, the “noisy data” could be wrongly matched as correct data resulting in either inefficient or even incorrect wrappers. Consequently, given a web page, the first task is to identify which part of the page is the data-rich section, i.e., the section or frame that contains the data objects of interest to the user.

It has been observed that pages from the same web site usually have a similar structure to organize their content, such as the location of advertisements and navigational menus. Based on this observation, we employ the *DSE* (*D*ata-rich *S*ection *E*xtraction) algorithm [19] to identify data-rich sections by comparing two web pages from the same web site. Its basic idea is to build DOM

trees [21] for the two pages, traverse them using a *depth-first* order, compare them node-by-node and discard those nodes with identical sub-trees that appear at the same depth.

3.2 C-repeated pattern

We assume that data objects contained in HTML pages are generated by some common templates and the structure of embedded data objects may appear repeatedly if the HTML page contains more than one data object instance. Accordingly, we iteratively discover continuous repeated (C-repeated) patterns as the wrapper candidates from the token sequences representing HTML pages. Note that some web site may show only one data object on each page. For that case, once we correctly identify the data-rich sections of each page, we can reorganize them by combining multiple pages into one single token sequence that will contain multiple data objects.

Definition: Given an input string S , a *C-repeated substring (pattern)* of S is a repeated substring of S having at least one pair of its occurrences that are adjacent.

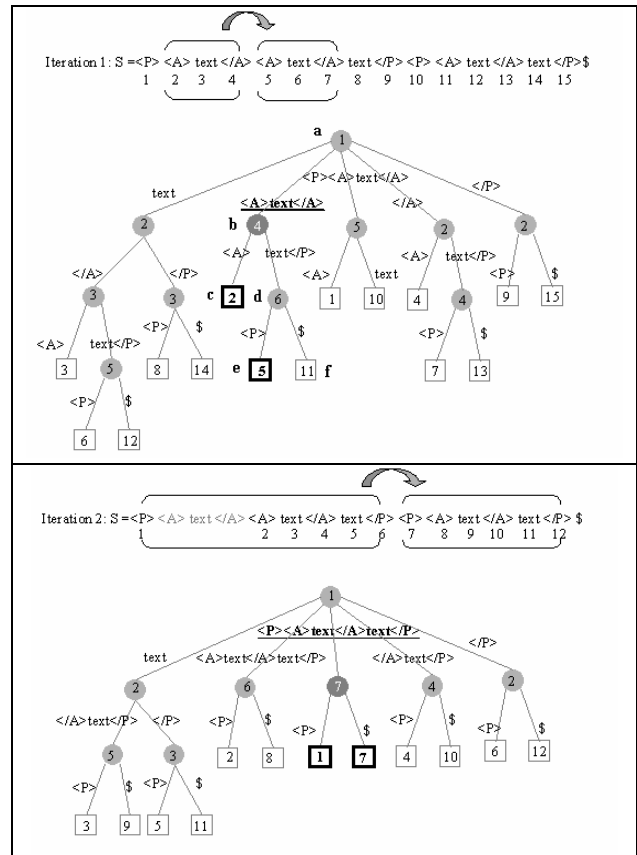


Figure 6. Iteratively discovering C-repeated patterns.

To expose the internal structure of a string, we adopt a data structure called a *token suffix-tree* [10]. An example token sequence and its token suffix-tree are shown in Figure 6 (Iteration I). Each leaf is represented by a square with a number that indicates the starting token position of a suffix. A solid circle represents each internal node with a number that indicates the token position where its children nodes differ. Sibling nodes sharing the same parent are put in alphabetical order. Each edge between two internal nodes has a label, which is the substring

between two token positions of the two nodes. Each edge between one internal node and one leaf node has a label, which is the token at the position of the internal node in the suffix starting from the leaf node. For example in Figure 6 (Iteration I), the edge label between node a and node b is “<A>text”, i.e., the substring starting from the first token up to, but not including the fourth token. The concatenation of edge labels from the root to node e is “<A>texttext</P><P>”, which is the unique prefix indicating the fifth suffix string “<A>texttext</P><P><A>texttext</P>”. Since a token suffix-tree is a special suffix-tree, it can be constructed optimally in $O(n)$ time [10]. If pointers to the original string represent the labels of all edges, then the space complexity is $O(n)$.

Path-labels of all internal nodes and their prefixes in the token suffix-tree are our candidates to discover C-repeated patterns. For each candidate repeated pattern, it is a C-repeated pattern if any two of its occurrences are adjacent, i.e., the distance between the two starting positions is equal to the pattern length. For example, “<A>text” is a repeated pattern in Figure 6 (Iteration I), since node b is an internal node with its path-label containing the pattern as a prefix. Its leaf nodes c , e and f indicate that the pattern appears three times in the sequence with starting position 2, 5 and 11. Specifically, this pattern is a C-repeated pattern because two of its occurrences are adjacent ($5-2=3$). An algorithm that discovers all C-repeated patterns from a suffix tree in $O(n \log n)$ time is presented in [20].

Our final objective is to discover nested structures from the string sequence representing an HTML page. We use a hierarchical pattern-tree to handle this task. The basic idea is based on the iteration of building token suffix-trees and discovering C-repeated patterns. For example in Figure 6 (Iteration I), we first build a token suffix-tree and find “<A>text” as a C-repeated pattern. Next, we mask the occurrence from S_2 to S_4 and form a new sequence in Figure 6 (Iteration II). Then, we build a suffix-tree for the new sequence and find a new C-repeated pattern “<P><A>texttext</P>”. Finally, we can obtain a wrapper from these two iterations, “<P>(<A>text)*text</P>” that correctly represents the structure for the two data objects in the given string, where “*” here means appears zero or more times.

In this iterative discovery process, patterns form a hierarchical relationship, since some pattern’s discovery is *dependent* on the discovery of some other patterns. For example, in Figure 6, “<P><A>texttext</P>” cannot be extracted unless “<A>text” is found. On the other hand, for each iteration phase, we may find several C-repeated patterns that are *independent* of each other. For example, in the string “texttexttexttext<P>text<P>”, the discovery of C-repeated pattern “text” is not dependent on the discovery of “text<P>” and vice versa. A *pattern tree* can be used to represent both the dependence and independence between discovered C-repeated patterns and to record the entire iterative discovery process. In a pattern-tree:

- pattern P_i is a *child* of pattern P_j , if P_i is discovered in the iteration phase right after the phase where P_j is found.
- pattern P_i is a *sibling* of pattern P_j , if P_i is discovered in the same iteration phase where P_j is found.

Initially, we set an empty pattern as the root of the suffix-tree. The iterative discovery process proceeds as follows. For each discovered pattern we retain the last occurrence in each repeated

region of that pattern and mask other occurrences in the current sequence (either the original one or the one formed by masking some tokens in the last phase) to form a new sequence. Then, we build a suffix-tree for the new sequence, discover new C-repeated patterns, and insert those patterns into the pattern-tree as children of the current pattern. We repeat this process for each newly found pattern. Once we finish processing the children patterns of the current one, we go back up one level and continue to process siblings of the current pattern. When we go back to the root again, we are finished. For each node in the pattern tree, we record its repeated occurrences for later processing.

Figure 7 is an example of a complete pattern-tree with each character representing a token in the HTML sequence. In Figure 7, the string is “ABACCABBABBBCCC”, where each letter represents a token, and the structure covering two data objects in this string is “(AB*A)C*”, where “*” means the substring may appear zero or more times. In such a pattern tree, every leaf node represents the discovered nested structure given by the path from the root to the leaf node. Although such a tree can clearly reflect the iterative discovery process, it is time-consuming to construct, especially when we handle real HTML pages with hundreds of tokens on average since in each iteration we need to reconstruct a suffix-tree ($O(n)$ time) and retrieve it to discover C-repeated patterns ($O(n \log n)$ time).

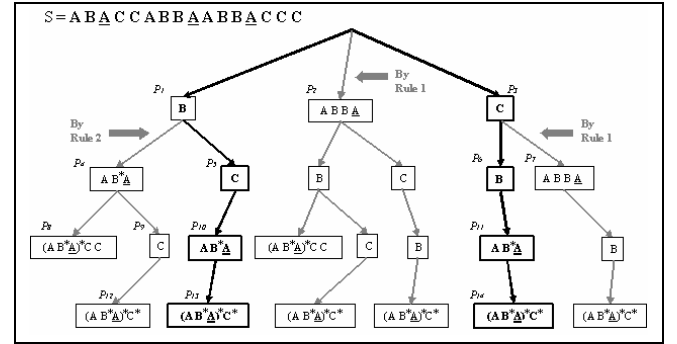


Figure 7. An example of the pattern-tree.

To lower the complexity, we employ a heuristic to filter out patterns that cross pairs of HTML tags, e.g., “</TR><TR>text”, and use three rules to prune some branches of the pattern-tree.

- Prune Rule 1:** A pattern is discarded if it contains one of its siblings.
- Prune Rule 2:** A pattern is discarded if it contains its parent while one of its siblings does not.
- Prune Rule 3:** Define the *coverage* of a pattern to be its string length multiplied by the number of its occurrences. In each processing iteration, if more than one child pattern has been found after applying rules 1 and 2, we only retain the one with the largest coverage and discard the others.

The first two rules enforce an order from inner levels to outer levels to form a nested structure for our discovery process. The reason for requiring this ordering is that the discovery of outer structures is dependent on the discovery of inner structures. If outer-level patterns are discovered before inner-level ones, we may get some wrong patterns. For example, in Figure 7, after masking newly discovered pattern P_4 , we get a sequence “ABACCABBABBBCCC”. If we extract outer-level patterns first, we will obtain P_8 “(AB*A)C*”, which misses the last “C” token in

the sequence, while if we extract P_9 “C” first, we will obtain a correct pattern P_{12} “(AB^{*}Δ)^{*}C^{*}” representing the nested structure in S. Applying the rules to Figure 7, pattern P_2 and P_7 are discarded by rule 1 and P_4 is discarded by rule 2. Thus, only the two paths $P_1P_5P_{10}P_{13}$ and $P_3P_6P_{11}P_{14}$ are left, which are shown in bold font. These two paths are both correct, but they overlap with each other in their iteration 3 and iteration 4. In fact, either of these two paths is sufficient, because pattern “B” and pattern “C” are independent of each other, i.e., they are in the same inner level of the sequence and it does matter which one of them is extracted first.

Consequently, we introduce the third rule so that each level of the pattern tree has only one child, which is the minimal size by turning a tree into a list. Note that it is not necessary to choose patterns by their coverage. They can be chosen simply by their length or by the number of occurrences or even randomly. However, processing larger-coverage patterns first will mask more tokens in the sequence and thus get a shorter sequence for later processing. After introducing these three rules, we will get a list-like pattern-tree for a string sequence representing the web page. In [20] we show that the time complexity of the pattern extractor as a whole is $O(n \log n)$.

The pattern with the highest nested-level in the pattern tree is chosen to be the nested schema we infer from the specific web page. Note that the nested-level of the extracted pattern may be less than the height of the pattern tree, since some patterns at different heights of the tree may be in the same level of the structure, such as P_1 “B” and P_5 “C” in Figure 7. Similarly, there may not be one pattern with the highest nested-level in the pattern tree. If we enclose lower-level patterns in parentheses followed by the symbol “*”, the pattern becomes a union-free regular expression (without disjunction, i.e., union operators). However, to make sure a lower-level pattern in the tree is nested inside a higher-level one, we need to verify that the former is a substring of the latter and they have overlapping occurrences. Otherwise, we may introduce unnecessary nested-levels into the patterns.

3.3 Optional attributes and disjunction

Up to this point, we have assumed that the patterns in a nested structure appear contiguously and thus we can infer regular expressions with “*” from them. However, we may not correctly obtain a nested structure for data objects that have an optional attribute because the attribute may only appear in some, but not all, occurrences of the pattern. For example, for the string “ABCΔ ABCCΔ ACCΔ ACAΔ” with “B” as the optional attribute, we will discover two patterns with the same nested-level, “ABC^{*}Δ” and “AC^{*}Δ”, each of which has only two occurrences in the string. However, if we generalize these two patterns, we can obtain “AB[?]C^{*}Δ”, where “?” means appears once or zero times, which matches all four data objects in the string.

When the form crawler sends out queries, it downloads K pages for each web site (K is relevant to the number of elements the form contains) and the wrapper generator extracts repeated patterns with the highest nested-level from them as the wrapper candidates. However, there may be more than one pattern with the highest nested-level for each page. Therefore, we may get more than K wrapper candidates for each web site. It has been observed that data objects embedded in web pages from the same web site usually share a common data structure. Consequently, the

wrapper candidates we discover should be similar to each other and they should all be similar to the real data structure of the web site’s data objects, possibly with some optional attributes missing or some attributes with disjunction values. Therefore, we try to construct a generalized wrapper from the multiple discovered patterns.

The discovered patterns can be “merged” into a single generalized pattern using a *string alignment* algorithm. An alignment of two strings S_1 and S_2 is obtained by first inserting spaces (or dashes), either into or at the ends of S_1 and S_2 , and then placing the two resulting strings one above the other so that the characters of the resulting strings can be put in one-to-one correspondence to each other as in the following example where the symbol ‘ ’ is used to represent an inserted space (see chapter 11 in [10] for details). String alignment can be done in $O(nm)$ time where n and m are the size of S_1 and S_2 . For example, the alignment for patterns P_1 = “ABCDXF” and P_2 = “BCEXF” will be:

P_1 :	A	B	C	D	X	F
P_2 :	–	B	C	E	X	F

Based on token comparison, we apply this algorithm to construct a generalized wrapper. Additionally, rather than inserting spaces into the alignment, we insert the corresponding tokens either in S_1 or S_2 into the alignment along with the symbol “?” representing zero or one occurrence, or the symbol “[” representing disjunction of the different tokens in S_1 and S_2 . For instance the generalized wrapper for P_1 and P_2 will be “A[?]BC(D|E)XF”.

Our discovered patterns may cross the boundary of a data object. Given a string sequence “ABCABCAX” and the pattern “ABC” representing a data object starting with A and ending with C, the given string sequence contains two instances of this data object. However, after applying our pattern extractor algorithm, we obtain the C-repeated patterns “ABC” and “BCA” from the string sequence. While the former pattern correctly represents the boundary of the data object, the latter pattern crosses its boundary.

To deal with this problem, we choose one pattern from all wrapper candidates as the correct indication of the data object’s structure. Specifically, we select the pattern with the largest *page-occurrence*, i.e., the largest number of web pages in which the pattern is discovered. Since the selected pattern appears in more pages than do other patterns, we believe that it correctly represents the boundary of the data object². Therefore, when aligning the discovered patterns, rather than comparing them pair by pair, we compare the patterns one by one to the chosen pattern. This alignment process needs $O(Mn^2)$ time when dealing with M patterns. Note that in this case inserting tokens at the left end of a pattern is not allowed during alignment so that the data object boundary is preserved. In addition, when aligning other patterns to the one with the largest page-occurrence, we discard those patterns that need a large number of inserted tokens, i.e., if the number of inserted tokens is larger than half of the pattern length³.

4. Data Alignment

The data aligner works in two phases. The data-extraction phase first extracts data from the web pages according to the wrapper

² [20] shows the experimental results on the distribution of the page-occurrence of discovered patterns.

³ [20] shows the experimental results on the number of discarded patterns.

induced by the wrapper generator and fills the extracted data into a table. Then the attribute-separation phase examines the table's columns and separates those attributes that are encoded into one text string into several newly added columns.

4.1 Data extraction

Given a regular expression pattern and a token sequence representing the web page, a nondeterministic, finite-state automaton can be constructed and employed to match its occurrences from the string sequences representing web pages. The algorithm can be found in [10] and its complexity is $O(nm)$ with n being the target string length and m being the size of the regular expression. Recall that when inducing the wrapper, any text string enclosed by pairs of HTML tags is replaced by the special token “*text*”. Therefore, once we obtain the occurrences of the regular expression in the token sequences, we need to restore the original text strings. Thus, each occurrence of the regular expression represents one data object from the web page.

We employ a *data-tree* for a given regular expression to record one occurrence of the expression. A data-tree is defined recursively as follows:

- If the regular expression is atomic, then the data-tree is a single node and the occurrence of the expression is the node label.
- If the regular expression is $E_1E_2\dots E_n$, then the data-tree is a node with n children and the i^{th} ($1 < i < n$) child is a data-tree that records the occurrence of E_i .
- If the regular expression is $(E_1|E_2)$, then the data-tree is a node with one child that records the occurrence of either E_1 or E_2 .
- If the regular expression is $(E)^*$ and there are m occurrences of E , then the data-tree is a node with m children and the i^{th} ($1 < i < m$) child is a data-tree that records the m^{th} occurrence of E .

In the left part of Figure 8, we show an example of a data-tree for the regular expression “ $A(B(C|F))^*D^*$ ” that records the occurrence of “ABCBFDDDD”. We use different subscripts to distinguish different occurrences of the same sub-pattern, such as “ B_1 ” and “ D_3 ”. We can see that the root of the data-tree has three children because the wrapper is a concatenation of “ A ”, “ $(B(C|F))^*$ ” and “ $(D)^*$ ”. Node 3 is the data-tree for “ $(D)^*$ ” and it has three children because there are three occurrences of “ D ” in the data. Similarly, node 2 has two children for the two occurrences “ B_1C_1 ” and “ B_2F_1 ” of the expression “ $(B(C|F))^*$ ”.

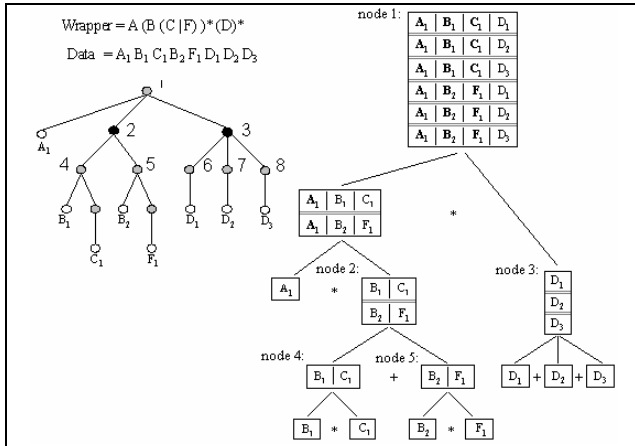


Figure 8. Example data-tree and its derived table.

From the definition of a data-tree, we can classify its internal nodes into two categories: *star-type* and *cat-type*. The *star-type* internal node is for $(E)^*$ -like sub-expressions of the wrapper and its children all record the occurrences of E , i.e., its children record multiple values of the same attribute of the data object. The *cat-type* internal node is for $(E_1|E_2)$ -like or E_1E_2 -like sub-expressions of the wrapper and its children record the occurrences of different sub-expressions, i.e., its children record the values of different attributes of the data objects. For example, in Figure 8, nodes 2 and 3 are both star-type internal nodes (in black color); nodes in gray color, such as nodes 1 and 4, are cat-type internal nodes. We separate these two kinds of internal nodes because when we fill a table with the data occurrences recorded by the data-tree, different methodology is used for star-type and cat-type internal nodes.

To construct a table of data from a data-tree we traverse the tree in a *depth-first* order and fill the table for one node by “adding” the tables of its child nodes (star-type internal nodes) or “multiplying” the tables of its child nodes (cat-type internal nodes). Given table T_1 and table T_2 , with n representing the number of rows of T_1 and m representing the number of rows of T_2 ,

- The result table of T_1 **adds** T_2 is constructed by taking the *union* of T_1 and T_2 , i.e., we insert all rows of T_1 and all rows of T_2 . Thus for the result table, its number of columns is the same as the number of columns of T_1 and T_2 and its number of rows equals n plus m .
- The result table of T_1 **multiplies** T_2 is constructed by a *Cartesian product* of T_1 and T_2 , i.e., we insert new rows that are built by repeating each row of T_1 m times and individually concatenating them with all rows of T_2 . Thus for the result table, its number of columns is equal to the number of columns of T_1 plus the number of columns of T_2 , and its number of rows is equal to n multiplied by m .

For example, the right part of Figure 8 shows how a table is constructed from the data-tree. The table for node 2 is the result of “adding” the tables for nodes 4 and 5, so it has 2 ($=1+1$) rows. The table for node 1 is constructed first by “multiplying” the table for node “ A_1 ” with the table for node 2, and then “multiplying” the result table with the table for node 3, so it has 4 ($=1+2+1$) columns and 6 ($=1*2*3$) rows.

Using the above algorithm, we can construct a table of data that is similar to a table in a relational DBMS where multiple values of one attribute are distributed into multiple rows of the table while the single value of other attributes are repeated.

4.2 Attribute separation

Before we do attribute separation we remove all HTML tags contained in the table as well as columns of the table with no text string inside to make sure every cell of the data table contains a text string as its content. The basic assumption of the attribute separation phase is that if several attributes are encoded into one text string, then there should be some special symbol(s) in the string as the separator to visually support users to distinguish the attributes. Thus, the problem here is to find the right separator for each web site.

To separate attributes of the data objects, for each column we search all its cells for characters neither a letter nor a digit and record their occurrences and corresponding positions in the cells. If the discovered characters have the same number of occurrences

in all cells for one column, then the character is recognized as the separator candidate for this column. The separator candidates are validated by several heuristics. For instance, “@” is not a valid separator since it appears in any text string representing email, and the following “.” after “\$” is not a valid separator either since it appears in the text string representing price. When several separators are found to be valid for one column, the attribute strings of this column are separated from the beginning to the end in the order of the occurrence positions of each separator.

5. Label Assignment

To assign labels to the columns of the data table containing the extracted data objects, i.e., to understand the meaning of the data attributes, we employ the following four heuristics:

Heuristic 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. Recall that the form crawler collects all labels of the form elements and sends out queries through these elements to obtain web pages. Therefore, 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.

Heuristic 2: search for voluntary labels in table headers

The HTML specification [22] 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) the contained data objects is examined for possible voluntary labels.

Heuristic 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.

Heuristic 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 the 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. In *DeLa*, we also allow users to add a label to unassigned attributes and to modify the assigned labels.

6. Experiments

We conducted experiments to study and measure the performance of the *DeLa* system. The system is completely written in Java and

*JTidy*⁴, a library for HTML cleaning, is employed. While we employ *HiWe* [15] as the form crawler, we do not measure its performance, but rather focus on measuring the performance of the other three *DeLa* components. We examined three categories of web sites, book shopping, job advertisements and car advertisements, and collected nine web sites for each category from *invisibleweb.com*⁵. We did not use any specific criteria to select these web sites other than they all provided a form-based search.

Recall that in label assignment we match form element labels to data attributes; so the correctness of form element collection will affect the performance of label assignment. In order to better examine the effectiveness of our proposed heuristics for label assignment, we simulated the actions of the form crawler by manually collecting the labels of the form elements and manually sending out queries through the HTML forms. We also manually made sure that every query we sent out had some results returned back. Table 1 shows the number of form elements contained by each web site (columns labeled by “FE”) and the number of queries we sent out (columns labeled by “Q”). Note that we only sent queries for each *SELECT* and *TEXTBOX* element, while ignoring other elements like *CHECKBOX* and *RADIO*. For each *SELECT* element, the query contained one of its option values and for each *TEXTBOX* element, the query contained one typed keyword. Therefore, the number of queries sent out for each web site is equal to the number of its *TEXTBOX* elements plus the number of its *SELECT* elements.

Table 1. Number of form elements and queries sent out.

	Book		Job		Car	
	FE	Q	FE	Q	FE	Q
Site 1	13	7	6	3	2	2
Site 2	8	7	3	2	15	10
Site 3	4	3	2	2	6	5
Site 4	6	4	3	2	4	2
Site 5	5	5	8	2	8	8
Site 6	6	3	4	3	4	4
Site 7	15	8	13	3	15	5
Site 8	8	7	5	2	7	7
Site 9	6	5	4	2	9	7

The columns labeled by “S” in Table 2 show the special symbols of “?”, “[” and “*” contained in the induced wrapper (the regular expression) for each web site, from which we can see that web sites do model their data objects both in plain-structure and nested-structure. We also show in Table 2 the precision of the induced wrapper for each web site (columns labeled by “P”), where the precision is measured by the ratio of the number of occurrences of the wrapper in the web pages to the number of data objects contained in the web pages. The result indicates that our method of automatic wrapper induction can obtain a very high precision. Other than the precision, we also measured the number of non-data objects that was wrongly extracted by the induced wrapper. Encouragingly, there was no information that was wrongly taken as data objects, i.e., the recall of the induced wrappers was 100% for each web site.

⁴ <http://sourceforge.net/projects/jtidy>

⁵ <http://www.invisibleweb.com/>

We show in Table 3 the performance of the attribute separation phase of the data extractor. Columns labeled by “Non” list the number of columns in the result table without separating attributes encoded in one text string. Columns labeled by “S” list the number of columns after the separation. The cells shaded in gray are the web sites that encode several attributes into one text string where the attribute separation phase succeeded. The cells in bold font are the web sites where we still have some attributes un-separated or mis-separated. The average correctness of the attribute separation is 95%. We examined the web sites that have encoded their attributes into one text string. The separating symbols they employ vary from site to site, for example, “[”, “~” and “/”. The main reason for the failure to separate some attributes is that the web sites use spaces as their separators.

Table 2. Precision of the induced wrappers and special symbols contained in them.

	Book		Job		Car	
	S	P	S	P	S	P
Site 1		1.00	?	1.00	?	1.00
Site 2	?,	0.90	?	1.00		0.95
Site 3		1.00		1.00		1.00
Site 4		1.00		0.98		1.00
Site 5		1.00		0.93	*, ?	0.97
Site 6	*	1.00	?	1.00		0.98
Site 7	?	0.97	?	1.00		1.00
Site 8		1.00		1.00		1.00
Site 9		1.00		1.00		1.00

Table 3. Precision of column separation.

	Book			Job			Car		
	Non	S	C	Non	S	C	Non	S	C
Site 1	7	8	11	5	5	5	7	7	7
Site 2	5	8	8	3	3	3	8	8	8
Site 3	2	3	3	2	3	3	5	7	8
Site 4	4	4	5	6	6	6	7	7	7
Site 5	3	5	5	6	8	9	7	8	9
Site 6	3	5	5	6	6	6	2	5	5
Site 7	4	7	7	6	6	6	5	6	7
Site 8	4	8	8	4	4	4	7	7	7
Site 9	5	6	6	4	4	4	3	4	3

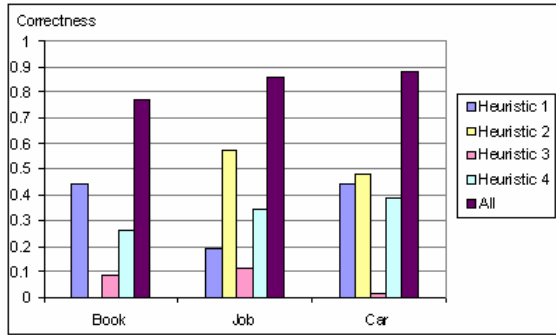


Figure 9. Correctness of the label assignment heuristics.

Finally, we measured the performance of the label assigner. Recall that we employ four different kinds of heuristics in the label assigner (see Section 5). Figure 9 shows the average

correctness of the four heuristics and the combination of all of them for the three categories of web sites, where correctness is measured by manually checking whether the meaning of the assigned label matches the meaning of the corresponding attribute. From this figure, we can see that the correctness of each heuristic varies with the type of web site. For example, most of the web sites for job advertisements list their data objects in HTML tables with table headers, so Heuristic 2 worked well for them. Similarly, Heuristic 1 worked well for the web sites for book shopping and car advertisement because book and car data objects tend to have a clear data structure. The performance of Heuristic 4 is quite stable, because the three categories of web sites all have data attributes, such as price and date. We note that even though no individual heuristic had a high correctness, the correctness rate for the combined approach is satisfactory (around 80%) given that we do not have any *a priori* knowledge about the page contents or any user involvement in the process.

7. Related Work

Early approaches to generate wrappers for web sites were mainly based on hand coding, which needs experts to analyze each web site, extract its specific data object structures and manually construct wrappers. Later approaches for wrapper generation are mainly based on unsupervised learning of user labeled examples to extract data objects from web pages [1], [11], [14] and [17]. The semantics of the extracted data are given in the user labeled examples. Another category of Information Extraction systems [2], [13] and [18] uses supervised wrapper generation, providing a GUI to interact with users during the wrapper generation process and data labels are provided by users during the interaction.

More recently, approaches to automatically generate wrappers without human involvement have been proposed [4], [5], [7] and [8]. [4] and [8] employ several heuristics for identifying separator tags to discover data object boundaries in web pages, such as the number of occurrences for each tag and the text size between a start tag and the corresponding end tag. These heuristics are good for segmenting web pages into parts, possibly containing data object instances. However, how to precisely locate the data object instances in the separated parts and how to extract them by their specific structures are not addressed.

Crescenzi et al. develop in [7] a wrapper induction system, *ROADRUNNER*, which 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, when generalizing a wrapper each mismatch can be treated in two alternative ways, which results in the algorithm having an exponential time complexity. Moreover, *ROADRUNNER* assumes the wrappers are *union-free* regular expression, which cannot catch “the full diversity of structures presented in HTML pages.” ([7])

Chang et al. propose a system called *IEPAD* in [5] that generates extraction rules by coding the HTML page into a binary sequence and then mining maximal repeated patterns in the sequence by building a *PAT* tree (Patricia Tree). The discovered maximal repeated patterns are further filtered by the size regularity between two adjacent occurrences and their density. Finally, the users can choose one of the generalized patterns as an extraction

rule to extract data objects from web pages. This approach is deterministic and efficient for web pages containing plain-structured data objects. However, it cannot handle complex, nested-structured data objects, whose occurrences may have a variable number of values on their attributes and thereby do not necessarily share the same HTML tag strings.

As far as we know, no fully automatic wrapper generation approaches pay attention either to the semantics or to the post-processing and presentation of the extracted data objects.

8. Conclusion

In this paper, we described a system, *DeLa*, which re-constructs (part of) a “hidden” back-end web database. It does this by sending queries through HTML forms, automatically generating regular expression wrappers to extract data objects from the result pages and restoring the retrieved data into a table. Our experiments on several collections of web sites indicate that *DeLa* performs very well in automatically inducing wrappers, extracting data objects and assigning meaningful labels to the data attributes. The experimental results also demonstrate the feasibility of heuristic-based label assignment and the effectiveness of the employed heuristics, which we believe sets the stage for more fully automatic data annotation of web sites.

As future work, we plan to include an ontology into the system, which can help the label assignment if the data domain of the web site is known. We also plan to build an integration component into the system to help users integrate the data objects extracted from different web sites. Another possible direction for future work is the information discovery problem, that is, how to automatically locate the web site(s) from which users want to extract data objects.

Acknowledgements

This work was supported by the UGC Research Grants Council of Hong Kong under the Areas of Excellence—Information Technology program.

References

- [1] B. Adelberg. “NoDoSE – A tool for semi-automatically extracting structured and semistructured data from text documents,” *Proc. ACM SIGMOD Conf.*, 1998, 283-294.
- [2] R. Baumgartner, S. Flesca and G. Gottlob. “Visual web information extraction with Lixto,” *Proc. 27th VLDB Conf.*, 2001, 119-128.
- [3] BrightPlanet Corp. “The Deep Web: Surfacing hidden value.” <http://www.completeplanet.com/Tutorials/DeepWeb/>
- [4] D. Buttler, L. Liu and C. Pu. “A fully automated object extraction system for the World Wide Web,” *Proc. Intl. Conf. on Distributed Computing Systems*, 2001, 361-370.
- [5] C.H. Chang and S.C. Lui. “IEPAD: information extraction based on pattern discovery,” *Proc. 10th Intl. Conf. on World Wide Web*, 2001, 681-688.
- [6] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman and J. Widom. “The TRIMMIS project: integration of heterogeneous information sources,” *Proc. IPSJ Conference*, 1994, 7-18.
- [7] V. Crescenzi, G. Mecca and P. Merialdo. “ROADRUNNER: towards automatic data extraction from large web sites,” *Proc. 27th VLDB Conf.*, 2001, 109-118.
- [8] D. Embley, Y. Jiang and Y. K. Ng. “Record-boundary discovery in web documents,” *Proc. ACM SIGMOD Conf.*, 1999, 467-478.
- [9] D. Florescu, A. Y. Levy and A. O. Mendelzon. “Database techniques for the world-wide web: a survey,” *SIGMOD Record* 27(3), 1998, 59-74.
- [10] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge, 1997.
- [11] C. Hus and M. Dung. “Generating finite-state transducers for semi-structured data extraction from the web,” *Information Systems* 23(8), 1998, 521-538.
- [12] T. Kirk, A. Levy, Y. Sagiv and D. Srivastava. “The Information Manifold,” *Proc. the AAAI Spring Symp. on Information Gathering from Heterogeneous, Distributed Environments*, 1995, 85-91.
- [13] L. Liu, C. Pu and W. Han. “XWRAP: An XML-enabled wrapper construction system for web information sources,” *Proc. 16th Intl. Conf. on Data Engineering (ICDE)*, 2000, 611-621.
- [14] I. Muslea, S. Minton and C. Knoblock. “A hierarchical approach to wrapper induction,” *Proc. 3rd Intl. Conf. on Autonomous Agents*, 1999, 190-197.
- [15] S. Raghavan and H. Garcia-Molina. “Crawling the hidden web,” *Proc. 27th VLDB Conf.*, 2001, 129-138.
- [16] S. Raghavan and H. Garcia-Molina. “Integrating diverse information management systems: a brief survey,” In *IEEE Data Engineering Bulletin* 24(4), 2001, 44-52.
- [17] B. Ribeiro-Neto, A. Laender and A.S. da Silva. “Extracting semi-structured data through examples,” *Proc. Intl. Conf. on Information and Knowledge Management*, 1999, 94-101.
- [18] A. Sahuguet and F. Azavant. “WysiWyg web wrapper factory (W4F),” *Proc. 8th World Wide Web*, 1999.
- [19] J. Wang and F. Lochovsky. “Data-rich section extraction from HTML pages,” *Proc. 3rd Conf. on Web Information Systems Engineering*, 2002, 313-322.
- [20] J. Wang and F. Lochovsky. “Wrapper induction based on nested pattern discovery,” *Technical Report HKUST-CS-27-02*, Dept. of Computer Science, Hong Kong U. of Science & Technology, 2002 (submitted for publication). <http://www.cs.ust.hk/~cswangjy/paper/tr-27-02.pdf>
- [21] World Wide Web Consortium. Document Object Model Level 3 Core Specification, 2001.
- [22] World Wide Web Consortium. HTML 4.01 Specification, 1999.