Wrapper Induction based on Nested Pattern Discovery

Jiying Wang
Department of Computer Science
University of Science and Technology
Clear Water Bay, Kowloon
Hong Kong
(+852)-23588836
cswangjy@cs.ust.hk

Frederick H. Lochovsky
Department of Computer Science
University of Science and Technology
Clear Water Bay, Kowloon
Hong Kong
(+852)-23586996
fred@cs.ust.hk

Abstract
One of the most difficult issues in information extraction from the World Wide Web is the automatic generation of wrappers that can extract relevant data objects embedded in semi-structured HTML pages. Wrapper generation requires that (a) the relevant data for extraction (data-rich section) in a web page be identified and (b) a pattern be constructed that represents the structure of the data objects in the data-rich section. This pattern can then be used to extract the data objects in a web page for subsequent querying. To address the first problem, a novel algorithm, called data-rich section extraction (DSE), is employed to identify the data-rich section of HTML pages. The DSE algorithm acts as a pre-processing “clean-up” step that improves the accuracy of the generated wrappers. To address the second problem, a new concept, C-repeated pattern, is utilized to identify plain or repeated nested data structures 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, we build a pattern-tree, which represents the hierarchical relationship between discovered patterns, and obtain a (regular expression) wrapper that is used to extract both plain- and nested-structured data from HTML pages. Our approach is fast, fully automatic with no human involvement, and experiments show that the discovered patterns can achieve high accuracy and retrieval rates when matching them in web pages to extract data object instances.

Keyword: Data-rich HTML page, Information Extraction, Wrapper Induction
1. Introduction

The amount of information available on the World Wide Web is large and growing rapidly. While search engines can be employed to find information on the Web, the search result is often not very selective and it is usually not possible to exploit the structure, e.g., nested hierarchical relationships that may exist among the retrieved data. Database technology has long been employed to address these two problems (i.e., selectivity of retrieval and exploitation of relationships among data). Information Extraction (IE) systems are web search systems that apply database search technology to web pages by identifying and extracting data objects\(^1\) present in the pages and then using a query language to query the extracted data objects ([4], [5], [6], [7]). A key component of an information extraction system is the wrapper, which contains a set of extraction rules that can extract data objects from HTML pages and return them in a structured format. Generating the wrappers for web sites presents a challenge because HTML pages are designed to benefit human browsing and not for the ease of computer applications.

Fortunately, most data-rich web pages (i.e., web pages that contain data objects) are dynamically generated. That is, after receiving a user request, a server script program fetches data from a back-end database and fills the data into a predefined HTML template. Such pages usually carry multiple instances of data objects that follow a particular alignment and format. Currently, most of the work on inducing wrappers for web pages often requires that a programmer understand 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 ([1], [2], [3], [6], [11], [12], [13], [14], [15]).

Previous approaches to wrapper construction either need human involvement ([1], [2], [11], [14], [15]), or have mainly focused on extracting plain-structured data objects with a fixed number of attributes and values ([6], [12], [13]). Nested-structured data objects, whose instances may have a variable number of values on their attributes, usually cannot be handled. For example, in pages from an Internet bookstore, some book data objects may only have one author, while others may have more than one author. Our research is motivated by the need to develop a method to automatically discover not only data objects with plain structures, but also those with nested structures.

In this paper we address the problem of constructing wrappers for data-rich web pages with nested-structured data objects. Since querying back-end databases generates such pages, our goal is, in fact, to reconstruct (part of) the structure of the back-end database given the generated web pages. In this paper,

\(^1\) A data object is any data in an HTML page that a user is interested in querying. For the template-generated web pages that we deal with here, the data objects often have a hierarchical nested structure (e.g., book data, product data, etc.).
we deal only with the problem of identifying the (database) structure and extracting the data objects in data-rich web pages. Once this structure is identified and the data objects extracted, there still remains the problem of assigning semantics to the extracted data objects. This problem is beyond the scope of this paper, but it is the subject of our on-going research.

Our approach to extract plain- or nested-structured data objects from web pages consists of the following steps. First, we employ a novel algorithm, called data-rich section extraction (DSE), to identify data-rich sections of the HTML page. This is the first contribution of our work. Next, we output the identified data-rich sections as a token sequence composed of HTML tags or elements, such as <TABLE> and <A>. Then, we employ a token suffix-tree to index the token sequence for the HTML page. A new concept of C-repeated (continuous repeated) substrings or sub-sequences is formally defined and discovered in the suffix-trees. We run the discovery process iteratively and build a pattern-tree to record the discovered C-repeated substrings and their hierarchical relationships. This is the second contribution of our work. Finally, a (regular expression) wrapper is generalized by multiple alignments of the discovered nested patterns and is used to match with web pages to extract data objects. Experiments show that our approach for automatically generating wrappers for HTML pages is fast and accurate requiring users neither to specify examples nor to interact with the system during the wrapper generation process. Therefore, our approach is applicable to some real-time applications such as web data integration or web comparison shopping services.

This paper is organized as follows. In Section 2, we present an overview of our approach. Section 3 gives the detailed description of how to identify data-rich sections of HTML pages. Section 4 introduces the notion of C-repeated pattern and the method to discover such patterns from a suffix-tree. Moreover, a pattern tree is utilized to record iteratively discovered patterns. Section 5 presents our experimental results. Section 6 discusses some related work. Finally, section 7 provides the conclusion and some future work.

2. System Overview

Currently, templates generate most commercial web pages, especially those pages that carry lots of data. Therefore, their formats, i.e., layouts, are regular. Such pages usually carry multiple instances of data objects that follow a particular alignment and format. Moreover, the structure of embedded data objects may appear repeatedly if the HTML page contains more than one instance of a data object. For instance, Figure 1 illustrates two query result pages from an online book store. Both of these pages continuously list multiple book data objects and those book data objects share a similar alignment and format, e.g., each book data object has a title followed by the information about authors, publication date, etc. Specifically, when we observe the HTML source of these two pages, the HTML tags that enclose the book data objects are almost the same. Since the multiple book data objects are continuously listed in these pages, the HTML tags that enclose the book data objects appear repeatedly. Therefore, the basic idea of our approach
is to induce wrappers by examining repeated HTML tag sequences in the data-rich sections of the web pages. The difficulty in identifying repeated HTML tag sequences is that the HTML-tag structures of the data objects are not necessarily exactly the same, since they may have multiple values on their attributes. For example, some book data objects 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).

![Query = “American Economy”  
Query = “China Economy”](image)

**Figure 1.** Query result pages from the same web site sharing the same layout.

In this paper, we use nested types as an abstraction to model the data objects contained in web pages. For example, in Figure 1 each book data object contained in the two pages has a title, a set of authors, a publisher, and so on. A book data object may have some optional attributes as well, for instance, some of them have an extraction on their content and some of them do not. Therefore, we can abstract the book data object as a nested type ([3], [9]):

\[
\text{Book} < \text{extract?}, \text{title}, \{\text{Author} <\text{name}\}\}, \text{publisher}, \text{publication date}, \text{format}, \text{list price} >
\]

where the symbols < > represent an unordered list tuple, the symbols { } represent a set and the symbol ? represents an optional attribute.

Note that a nested type may not fully capture the diverse structures of data objects embedded in HTML pages. Fortunately, nowadays, script programs, fetching data from back-end relational or object-relational databases, generate most data-rich HTML pages. Therefore, the structure of the data objects in such web pages is quite regular. It has been shown in [3] and [9] that the nested type is a good abstraction to describe the structure of data objects contained in fairly regular web sites. Thus, we consider the data objects embedded in HTML pages as string instances of the implied nested type of the script programs or templates where the instances are encoded in HTML tags.
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 $\Sigma$ and a special token text that is not in $\Sigma$, a regular expression (UFRE) over $\Sigma$ is a string over $\Sigma \cup \{\text{text}, *, ?, |, (, )\}$ defined as follows:

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

In this paper, we also employ a regular expression representing wrappers to extract data objects from web pages. For example, we list in Figure 2 the HTML code for the two book data objects in the web pages shown in Figure 1 and the corresponding wrapper as a regular expression over an alphabet of HTML tags and the token “text”. We can see that the two book data objects are two string instances of the book data object nested type defined above and they are encoded in some HTML tags. Thus, our goal is to induce the wrapper, or extraction rules, based on some HTML-encoded data object instances. After that, the induced wrapper can be employed to extract data object instances from the data-rich section of the web pages.

**Book instances encoded in HTML tags:**

```html
<TABLE>
  <TR>
    <TD> Understanding China’s Economy </TD>
  </TR>
  <TR>
    <TD> Gregory C. Chow </TD>
    <TD> ... </TD>
  </TR>
</TABLE>
<TABLE>
  <TR>
    <TD> China and the Asia Pacific Economy </TD>
    <TD> C. H. Chu </TD>
    <TD> Y. Y. Fang </TD>
    <TD> C. A. Wong </TD>
  </TR>
  <TR>
    <TD> ... </TD>
  </TR>
</TABLE>

**Wrapper in union-free regular expression:**

```html
<TABLE>
  <TR>
    <TD> text </TD>
  </TR>
  <TR>
    <TD> ( <A> text </A> )* </TD>
  </TR>
  <TR>
    <TD> text </TD>
  </TR>
</TABLE>
```

*Figure 2.* Example HTML source and the corresponding wrapper.

To address the problem of inducing wrappers to extract plain/nested-structured data objects from web pages, our approach contains three components: a data-rich section extractor, a pattern extractor and a wrapper generator. Note that a pattern is defined as the discovered substrings from the web page sequence string that represents a web page composed of HTML tags and the token “text”. Figure 3 shows the system architecture. Given some web pages from some site, the data-rich section extractor is responsible
for extracting the data-rich sections from the web pages and outputting them as token sequence strings to
the pattern extractor. The pattern extractor builds *token suffix-trees* on the token sequence strings and
discovers specific patterns (*C-repeated*) in the suffix trees. The discovered patterns are inserted into a
*pattern-tree*, which records the nested relationship between patterns. Finally, the pattern extractor outputs
the discovered patterns to the wrapper generator, which generalizes a final wrapper for the web pages
from the patterns. We briefly introduce the three components in the following sub-sections.

![Figure 3. System architecture.](image)

### 2.1 Data-rich section extractor

Web pages in HTML format are mainly designed for human browsing. 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 in the page 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 an algorithm [18], referred to as the DSE (Data-rich Section Extraction) algorithm, to identify data-rich sections by comparing two data-rich pages from the same web site. The algorithm is explained in detail in section 3. Its basic idea is to build tag-trees [19] for the two pages, traverse them using a depth-first order, compare them node-by-node and discard those nodes with identical tag names that appear at the same depth. For example, Figure 1 shows two query-result pages from an online bookstore that share the same layout. Their top and left bottom parts are identical, while their right bottom parts are different. After the tree construction and matching, the parts circled by the two dashed ellipses survive, having been identified as data-rich sections of these two pages. Note that we assume the HTML pages are well formed, for example, tags must appear in pairs. If they are not, we can use some tools such as JTidy2 to clean up the pages.

The result, represented as a token sequence of the identified part, is output to the pattern extractor. Since our final goal is to explore useful tag structures from HTML pages and extract data objects embedded in those structures, HTML tags such as <P>, <UL>, <TABLE> and so on, are taken as tokens with their attributes ignored. Additionally, the plain text enclosed by one pair of tags is considered to be one single token, “text”. Hence, in the later phase of pattern extraction, the following two data objects

“<TR><TD>Jiying Wang</TD><TD>Student</TD></TR>” and
“<TR><TD>Fred Lochovsky</TD><TD>Professor</TD></TR>”

contained in two web pages from the same site will be considered as having the same structure.

Based on the conclusion in [6] that “high-level abstraction has better performance average”, we discard text-level tags that define text display information, such as font style and size, because they contain little information on how the data object is structured. We only keep block-level tags that describe the page layout, such as lists and tables. For ease of later computation, when we output the token sequence, we adjust all tags to be a uniform length of 6 by adding spaces at the end of those tags whose length is less than 6 and deleting the extra characters of those tags whose length is greater than 6, while keeping the uniqueness of the tokens.

2.2 Pattern extractor

The pattern extractor is the core component of our approach and is responsible for building token suffix-trees for token sequences and extracting repeated patterns with nested structures from them. In this subsection, we briefly discuss how a nested pattern is discovered using the simple example in Figure 4. In

2 http://sourceforge.net/projects/jtidy
Section 4 we will present some formal definitions and show in detail how a suffix-tree can be used to extract repeated patterns with nested structures.

The example page in Figure 4 is returned from a CS Bibliography site for the query “information extraction”. It is clear that this part of the HTML page contains 6 research papers relevant to the query organized into 3 groups according to the source from which they come. We can see that research papers may have multiple authors and paper groups may have multiple paper instances. For purposes of illustration, tag-strings of the first paper group are shown in the right part of Figure 4 and are arranged as indented sentences with line numbers (indicated by L_i in the rest of this section). Strings from L_6 to L_15 represent the first paper instance and strings from L_16 to L_24 represent the second paper instance in the group. However, if we match these two regions of strings line-by-line they are not exactly the same, since the former has two authors (L_{10} and L_{11}) while the latter has only one (L_{20}). Note that if we skip L_{10}, which is identical to its following line L_{11}, then the strings from L_6 to L_{15} and the strings from L_{16} to L_{24} can be matched exactly. Similarly, if we skip the additional paper instances so as to leave only one for each paper group, we can match all the paper groups with the same structure. Therefore, by iteratively applying the process of discovering and skipping continuous repeated patterns in a given sequence we can extract the multiple-level nested structures.
2.3 Wrapper generator

Recall that our basic assumptions are 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 discover continuous repeated patterns as the wrapper candidates from the token sequences representing HTML pages. However, there still exist some problems to resolve. The first problem is that we may discover more than one nested pattern from each token sequence and the number of discovered patterns may be even higher when we deal with several pages from the same web site. Usually not all of the discovered patterns correctly reflect the embedded data objects’ structure. Thus we need some method to generalize one wrapper from all the discovered patterns.

Another problem is that the HTML-tag tokens of data object instances are not necessarily exactly the same when they have optional attributes because the attribute may only appear in some, but not all, occurrences of the pattern. For example in Figure 4, paper data object instances one, two, three and six contain a hyperlink to “BibTeX”, while paper data object instances four and five do not contain such a hyperlink. This hyperlink can be considered as an optional attribute of the paper data objects embedded in this CS Bibliography site. Consequently, we may discover two different repeated patterns from the token sequence of this page, one of which contains a substring “<TD><A>text</A></TD>” as the HTML-tags of the “BibTeX” hyperlink, while the other pattern does not contain such a substring. Our technique for handling such cases is to align the validated patterns and identify the substrings that only appear in some of the patterns as the optional ones. Therefore, the wrapper generator has the responsibility for first validating the discovered patterns and then aligning them to generalize a final wrapper that may be nested and may contain some optional sub-structures.

Once the pattern extractor outputs the patterns as a regular expression (see Section 3 for details) representing the nested schema of the web pages, a pattern-matching algorithm can be applied to extract data objects from web pages. Since the extraction rule is expressed as a regular expression, a nondeterministic, finite-state automaton can be constructed and employed to match its string occurrences in string sequences representing web pages. The algorithm can be found in [10] and [16] and its complexity is \( O(nm) \) with \( n \) being the target string length and \( m \) being the size of the regular expression.

3. Data-rich Section Extraction

Our DSE (Data-rich Section Extraction) algorithm is motivated by the observation that pages from the same web site often use the same HTML design template, so that their HTML structures are quite similar or the same. Moreover, the same information concurrently appearing in those pages is often used for navigational purposes, advertisement or other nepotistic purposes; usually such information has little to do
with the main content of the pages. The basic idea here is to compare two pages from the same web site to remove their common sections, and identify the remaining ones as data-rich sections.

To the best of our knowledge, there is currently no good approach to identify the data-rich section of web pages. An approach is mentioned in [5] and [8], which identifies the section of each page that has the highest fan-out as the data-rich section. However, this heuristic does not hold under some cases where the number of data objects contained in the input pages is relatively small. For example, suppose a web page contains two <TABLE>s. One contains only 2 data objects and the other one contains 3 hyperlinks, “home”, “previous”, and “next”, used for navigational purposes. In this case, the second <TABLE>, since it has the highest fan-out of 3, will be wrongly identified as the data-rich section although the first <TABLE> is the correct one. We will show some experimental results on comparing this approach with our DSE algorithm in Section 5.

After downloading the two pages from the same web site, we need to parse them and employ a data structure to represent their layout in order to run our comparison algorithm. As HTML pages are composed of tags and text enclosed by tags, it is possible to represent an HTML page’s layout by a tree-like structure referred to as the Document Object Model (DOM) [19]. We work on a simplified DOM tree for HTML pages because not all HTML tags provide structural information for a page [20]. Given two DOM trees representing the two downloaded pages, we now face the problem of matching the “similar” structures of the trees. Our basic idea is to traverse these two trees using a depth-first order and compare them node-by-node from the root to the leaves. However, before explaining our DSE algorithm in more detail, we need to define several functions employed in the node comparison.

- **RightSibling(parentnode, nodei)** returns the right sibling of nodei that shares parentnode with nodei
- **TagName(nodei)** returns the tag name of nodei
- **Attr(nodei)** returns the attribute of the nodei, if the node is <A> or <IMG>, otherwise it returns null
- **Same(nodei, nodej)** = 1, if TagName(nodei) = TagName(nodej) and Attr(nodei) = Attr(nodej)
  = 0, otherwise

The algorithm works recursively. For two nodes, if they are internal nodes and are the Same (as defined above), then the algorithm will go down one level to match their children from the leftmost to the rightmost one. If they are leaf nodes and are the Same, then they will be removed from the trees. If the nodes are not the Same, the algorithm will return to their parent and continue to compare the other children, if any. If all of the children of two parent nodes have been compared, then they will be removed only if all of their children have been removed. Figure 5 shows the pseudo-code of our tree-matching algorithm.
Figure 6 presents a simple example to demonstrate how we deal with the tree-matching problem. Suppose in Figure 6, tree A and tree B are two tag trees representing two pages from which we want to extract the data-rich sections. From the roots, we match the structures of tree A and tree B. Because the root pair (table, table) are the Same, we go down one level to compare their children. The first child pair (tr, tr) are the Same (see arrow 1 in Figure 6) and, thus, we go down one more level to (td, td), which are also the Same. Next, (a1, a1) are compared. Since they are the Same, (a1, a1) are removed. Then (a2, a2) and (a3, a3) are compared and removed. Next, we go back to (td, td) and find that all of their children are removed; thus, we remove (td, td) also. Similarly, we remove (tr, tr). Next, we go back to (table, table) and compare their next children. Thus, (table, map) are compared and they are different (see arrow 2). Then we compare (table, tr) and they are different also (see arrow 3). Because “tr” has no more right siblings to compare with “table”, we finish processing “table” and go back again to the root. Note that the leftmost (tr, tr) have already been removed. Thus, we next compare the rightmost “tr” with “map” (see arrow 4), and “tr” with “tr” (see arrow 5). Accordingly, we go down the two trees and eventually compare (ul, ul). Here we have a more interesting case. To compare the children of (ul, ul), we first process (a5, a6), detect a difference and then go to (a5, a5) (see arrow 6 and arrow 7). We remove these nodes since they are the Same. After that, we go up to (ul, ul) and go down again. At this point, only (a6, a6) are left. Therefore, we remove them and their parents (ul, ul). Note that when we go back up to “td,” a4 is still there. Therefore, we will remove neither “td,” “tr” nor “table.” The pruned tree A is pictured in Figure 6.
From the above illustration, we can see that the algorithm only compares nodes from the same layer of two trees. If we let the height of the target tree be $H$, and the maximum number of nodes in a single layer be $M$, the time complexity of this algorithm will be $O(H^2M^2)$. However, this is the worst case. In practice, we can avoid a lot of comparisons since we do not need to compare those nodes whose parents have different HTML tags.

It is worth mentioning that HTML pages vary so much that even two pages that look the same in a browser can have different tag structures. For example, `<TABLE><A>…</A></TABLE>` and `<TABLE><TABLE><A>…</A></TABLE></TABLE>` result in different subtrees, but they look the same in a browser. Unfortunately, such variations of tag combinations can be infinite. In our tree-matching algorithm, we do not consider such cases. However, since the basic assumption of the DSE algorithm is that the same template generates HTML pages from the same web site, therefore, the tree structures of those web pages should be the same.

4. Pattern Extractor and Wrapper Generator

To explain how the pattern extractor works, we first formally define several concepts, then introduce the token suffix-tree and its construction, and finally present details on how to discover continuous repeated patterns in the suffix-tree and possible nested structures.
**Definition 1:** Given an input string $S$, a *repeated substring (pattern)* of $S$ is one having at least two matching occurrences at distinct positions within $S$ [16].

**Definition 2:** Given an input string $S$, a *maximal repeated substring (pattern)* of $S$ is a repeated substring of $S$ having at least a pair of its occurrences that cannot be extended further in the left and right direction [16].

**Definition 3:** 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.

Note that a maximal repeated substring might have overlapped occurrences, while a *C-repeated* substring might not be maximal repeated. For example, given a sequence ABCABCAD, ABCA is a maximal repeated substring with overlapped occurrences (1,4) and (4, 7), and ABC is a C-repeated substring, but not maximal repeated, with adjacent occurrences (1, 3) and (4, 6). The above example shows some differences between a maximal repeated pattern and a C-repeated pattern. In fact, we believe that extracting C-repeated patterns is more suitable for the web information extraction problem than extracting maximal repeated patterns. Maximal repeated pattern extraction would fail for some cases, like the above example, and the number of maximal repeated patterns is usually larger than the number of C-repeated patterns in the token string of the real HTML pages.

**4.1 Token suffix-tree**

A suffix-tree is a classic data structure that exposes the internal structure of a string and is used in a wide range of string applications. The original definition of suffix-trees is for the character-level. In this paper, to cater to our objective of discovering tag-based patterns in the string sequence representing web pages, we adjust the definition of suffix-tree from the character-level to the token-level. Here we quote and adjust the definition of suffix-tree as given in chapter 12 of [10].

**Definition 4:** A *token suffix-tree* for an $n$-token sequence $S$ is a directed tree with exactly $n$ leaves numbered 1 to $n$, representing suffixes $S[1..n*k]$, $S[k+1..n*k]$, … and $S[(n-1)*k+1..n*k]$], where $k$ is the uniform length of tokens. Each internal node, other than the root, has at least two children and one edge labeled with a nonempty substring of $S$. No two edges out of a node can have edge-labels beginning with the same token. The key feature of a 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 a unique prefix of the suffix of $S$ that starts at token $i$, i.e., $S[i*k+1..n]$.

---

3 Section 5 shows the experimental results on the comparison.
To make sure there is no suffix of S that can be a prefix of any other suffix, we add a special terminating token ‘$’, which appears nowhere else in S, to the end of the token sequence. An example token sequence and its token suffix-tree are shown in Figure 7. 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 instance, the edge label between node $b$ and node $c$ is ‘<IMG>text’, i.e., the substring starting from the second token up to, but not including the fourth token. The edge label between node $d$ and node $f$, namely the suffix starting from 3, is ‘<P>’, i.e., the token at position 6 in “text<IMG>text<IMG>text<P>text<P>”. The concatenation of edge labels from the root to node $g$ is “text<IMG>text<P>”, which is the unique prefix indicating the fifth suffix string “text<IMG>text<P>text<P>”.

In fact, our token suffix-tree is a special suffix-tree built on the alphabet composed of HTML tags and a token of “text”, where all tokens have the same length. Therefore, the construction time cost of the token suffix-tree is the same as that for the generalized suffix-tree. Suppose the input sequence has a size of $n$, then the construction of the suffix-tree can be performed optimally in $O(n)$ time [17]. If the labels of all edges are represented by pointers to original string, then the space complexity of a suffix-tree is $O(n)$. 

**Figure 7.** An example token string and its token suffix-tree.
4.2 Discovering C-repeated patterns

Given a suffix-tree for a given string sequence S, let the path label of a node be the concatenation of all edge-labels from the root to that node. Each path label of an internal node represents a repeated substring of S, because all leaf children of that node share the substring as a common prefix. Since every internal node has at least two children and they must have different beginning tokens in their edge-labels, the path-label of the internal node is a right maximal repeated substring of S. The following theorem states the relationship between repeated patterns and path-labels, given a sequence S with a size of \( n \) and its suffix-tree T.

**Theorem:** For a path-label of an internal node in T, every prefix of this label (including itself) is a repeated pattern of S; for a repeated pattern of S, at least one internal node in T can be found with the pattern being a prefix of its path-label.

**Proof:** The first part of this theorem is obvious, since the path-label of an internal node implies that at least there exist two leaf nodes that share this path-label as their common prefix. So, the two leaf nodes also share all prefixes of the path-label. For a repeated pattern in S with two distinct occurrences \( S[i..i+k] \) and \( S[j..j+k] \), there must be two leaf nodes in T representing suffixes \( i \) and \( j \) of S. The lowest common ancestor of leaves \( i \) and \( j \), which is the deepest node in T that is an ancestor of both node \( i \) and node \( j \), has a path-label identifying the longest common prefix of suffix \( i \) and \( j \). Because of its maximality in length, the repeated pattern must be a prefix of the path-label.

From the theorem, we can see that scanning the suffix-tree once is sufficient to retrieve all repeated patterns in the string. Moreover, path-labels of all internal nodes and their prefixes in the suffix-tree are our candidates to discover continuous repeated substrings, i.e., 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. The occurrence retrieval of each repeated pattern is quite simple in a suffix-tree. For a repeated pattern P, we can find the highest internal node in the tree, among all nodes with their path-labels containing the repeated pattern as a prefix. Leaf nodes of that internal node record how many times the pattern occurs and where it occurs.

For example, “text<IMG>” is a repeated pattern in the sequence in Figure 7. Node \( c \) and node \( d \) both are internal nodes with their path-label containing the pattern as a prefix, while node \( c \) is the higher one and its leaf nodes \( e, f \) and \( g \) indicate that the pattern appears three times in the sequence with starting positions 1, 3 and 5. Specifically, this pattern is a C-repeated pattern because two of its occurrences are adjacent (3-1=2 and 5-3=2). Similarly, C-repeated pattern “text<P>” can be discovered from node \( h \) and C-repeated pattern “<IMG>text” can be discovered from node \( k \).

In practice, we can find C-repeated patterns in a reverse way. Rather than obtaining every repeated pattern as the candidate and retrieving its occurrences for adjacency, we can start from the occurrences to
retrieve C-repeated patterns. We traverse the tree from bottom to top. For each node we create a simple sorted list to store its occurrences. We process each node after all its children have been processed by merging the sorted occurrence lists from its children into a sorted occurrence list of the current node. During the merging process for the current node, we examine the distance between all adjacent pairs of its occurrences. If any distance, suppose it is $m$, is equal to or less than the length of its path-label, this implies that the prefix with length $m$ of the current path-label is a C-repeated pattern. For instance, in Figure 7, when processing node $c$, we merge the occurrence lists $<1, 3>$ and $<5>$ from its children nodes $d$ and $g$. The occurrence list for node $c$ becomes $<1, 3, 5>$. During the merging, we find that node $c$ has two pairs of adjacent occurrences $(1, 3)$ and $(3, 5)$ both having a distance of 2. Since this distance is less than the length of the path-label “text<IMG>text” of node $c$, this implies that the prefix “text<IMG>” is a C-repeated pattern.

Since the suffix-tree $T$ has $n$ leaves, and each internal node must have at least two children, $T$ can have at most $n$ internal nodes. The merging process is equivalent to a multi-parallel merge sort, which has time complexity $O(n \log n)$. If we store the occurrence list for every node, the space complexity is $O(n \log n)$, since the height of the tree is not higher than $\log n$.

### 4.3 Building pattern trees

We have shown how to discover C-repeated patterns from a suffix tree built for a given string sequence. However, this is not our final goal. Our final objective is to discover nested structures from the string sequence representing an HTML page. In this section, we propose a hierarchical pattern-tree to handle this task. Our basic idea is based on the iteration of building suffix-trees and discovering C-repeated patterns. For example, in Figure 8 (Iteration I), we first build a suffix-tree and find “<A>text</A>” as a C-repeated pattern. Next, we mask the occurrence from $S_2$ to $S_4$ and form a new sequence in Figure 8 (Iteration II). Then, we build a suffix-tree for the new sequence and find a new C-repeated pattern “<P><A>text</A>text</P>”. Finally, we can obtain a wrapper from these two iterations, “<P>(<A>text</A>)*text</P>” that correctly represents the structure for the two data objects in the given string, where “*” here means appears zero or more times. On the contrary, if we only discover maximal repeated patterns in the given string sequence, as is done in the IEPAD algorithm [6], we can only get patterns “text”, “<A>text</A>”, “<P><A>text</A>text</P>” and “<A>text</A>text</A>text</P>”. These patterns can only cover part of the data objects.

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 8, “<P><A>text</A>text</P>” cannot be extracted unless “<A>text</A>” 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 “text<IMG>text<IMG>text<IMG>text<P>text<P>” in Figure 7, the discovery of C-repeated pattern “text<IMG>” is not dependent on the discovery of “text<P>” and vice versa. Therefore,
we employ a tree structure, called a pattern tree, to represent both the dependence and independence between discovered C-repeated patterns and to record the whole iterative discovery process.

We give the formal definition of a pattern-tree below:

**Definition 5:** 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.

**Definition 6:** In a pattern-tree, 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.

<table>
<thead>
<tr>
<th>Iteration I</th>
<th>Iteration II</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Iteration I" /></td>
<td><img src="image" alt="Iteration II" /></td>
</tr>
</tbody>
</table>

**Figure 8.** Iteratively discovering C-repeated patterns.

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 9 is an example of a complete pattern-tree with each character representing a token in the HTML sequence. In Figure 9, the string is “ABACCCABBAABBACCC”, 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).

To lower the complexity, we employ a heuristic to filter out patterns that cross pairs of HTML tags, e.g., “</TR><TR>text”. We also propose two 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.

These 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 9, after masking newly discovered pattern \(P_6\), we get a sequence “ABACACBACC”. If we extract outer-level patterns first, we will obtain \(P_8\) “(AB\(\star\)A)\(\star\)CC”, 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\(\star\)A)\(\star\)C” representing the nested structure in S.

\[ S = A B A C C A B B A A B A C C C \]

Applying the rules to Figure 9, 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 \(P_3P_5P_{10}P_{14}\) 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.

![Figure 9. A complete pattern-tree.](image-url)
Consequently, we introduce another 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.

**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 rule 1 and rule 2, we only retain the one with the largest coverage and discard the others.

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, if we process larger-coverage patterns first, we 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. For each level of the pattern tree, \(O(n)\) is required to build a suffix-tree for the current sequence and \(O(n \log n)\) is required to discover C-repeated patterns in the sequence. We believe that the height of the pattern tree, i.e., the nested-level of the extracted data object structure, is quite small (less than 6 in our experiments) compared to the number of tokens in the string sequences representing HTML pages (hundreds of tokens on average). Therefore, 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 \(P_1 \text{“B”}\) and \(P_5 \text{“C”}\) in Figure 9. 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.

**4.4 Generalizing the wrapper**

Up to this point, we have assumed that the patterns in a nested structure appear contiguously and thus we can infer regular expressions 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 “ABC\(\underline{A}\) ABCCA ACC\(\underline{A}\) ACA” with “\(\underline{B}\)” as the optional attribute, we will discover two patterns with the same nested-level, “ABC\(\underline{A}\)” and “AC\(\underline{A}\)”\(\underline{A}\)”, each of which has only two occurrences in the string. However, if we generalize these two patterns, we can obtain “AB\(\underline{C}\)\(\underline{A}\)”\(\underline{A}\)”, where “?” means appears once or zero times, which matches all four data objects in the string.
To solve this problem, we download $K$ training pages ($K \geq 2$) for each web site and extract repeated patterns with the highest nested-level from them as our wrapper candidates. Note that there may be more than one pattern with the highest nested-level for each page. Therefore, we may get more than $K$ wrapper candidates. It has been observed that data objects embedded in web pages from the same web site usually share a common data structure. Based on this observation, 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 or fields missing. Therefore, we 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”, $P_2$=“BCEXF” will be:

$$
P_1: \quad \begin{array}{cccccc} A & B & C & D & X & F \\
\end{array}
$$

$$
P_2: \quad - & B & C & E & X & F
$$

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. For instance the generalized wrapper for $P_1$ and $P_2$ will be “(A)?BC(D|E)XF”.

The final issue to consider in generalizing a wrapper is that 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 2 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, when constructing the generalized wrapper 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 then do other patterns, we believe that it correctly represents the boundary of the data object. 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

---

1 Section 5 shows the experimental results on choosing different $K$.
2 Section 5 shows the experimental results for the distribution of the page-occurrence of discovered patterns.
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 length).

5. Experimental Results

Our system is written in Java and our experiments were performed on a PC equipped with an Intel Pentium III 450MHz processor with 256Mbytes of RAM, running Windows 98 and Sun JDK 1.3. HTML pages from several data-intensive web sites were downloaded to run our experiments. To avoid errors and missing tags in the HTML pages, we employ JTidy to clean our data sources.

To test the ability of our system to extract nested-structured data objects, we chose 6 web sites (the first 6 rows in Figure 10(a)) from which to download pages as training pages all of which contained nested data with a nested-level of at least 2. The web sites include one computer bibliography search and five online shopping sites, selling books, multimedia and cosmetics. We also chose 8 popular search engines (the last 8 rows in Figure 10(a)) from which to download their query result pages, with a nested-level of 1, to verify the performance of our algorithm on pages containing repeated data objects with plain structures.

We first show the performance of the DSE algorithm, which is utilized to extract data-rich sections of HTML pages by comparing two pages from the same web site. The table in Figure 10(a) compares the average recall of the DSE algorithm (the column labeled “DSE”) and the Highest Fan-out Rule algorithm proposed in [5] and [8] (the column labeled “HFR”) in locating the data-rich sections of 10 HTML pages for each web site. The recall for a page is equal to 1 if an algorithm locates a section in the page containing all the data objects and it is equal to 0 if an algorithm misses some data objects. (In the table, the recall ranges from 0 to 1 since 10 pages were processed for each web site.) The result is encouraging since the DSE algorithm does not miss any data objects for each site while the Highest Fan-out Rule algorithm fails for almost 6 of the 14 web sites.

Besides measuring whether the DSE algorithm misses any data objects (recall), we also need to measure whether it includes any irrelevant information (precision). Therefore, we measure the average number of HTML elements (tags and text) in the output of the DSE algorithm and the minimal number of elements that includes all data objects. The results shown in Figure 10(b) indicate that the DSE algorithm is quite close to the optimal answer that is obtained by manual identification and it can prune many irrelevant elements (43.3% on average) from the HTML pages.

---

* Section 5 shows the experimental results on the number of discarded patterns.
Recall that after identifying and extracting the data-rich sections of a web page, the next step is to identify and extract patterns within the extracted data-rich sections. From each web site we download \( K \) pages to extract one or more than one nested-structured data object from each page. We illustrate in Figure 11 the number of unique patterns discovered by the pattern extractor from these web sites according to \( K \). The six web sites with nested-structured data objects, where the nesting level is greater than 2, are shown in Figure 11(a). The eight web sites (search engines) with plain-structured data objects are shown in Figure 11(b). These two figures show that the number of discovered patterns remains stable after a certain number of pages are processed although this number may vary from site to site. We can also see that, for web sites with nested-structured data objects, the number of pages that need to be processed (around 20) before the number of discovered patterns stabilizes is larger than that (around 5) for web sites with plain structures. The reason for this difference is because the web sites with nested-structured data objects have
more optional attributes in their data objects than the web sites with plain-structured data objects. Thus, more pages need to be processed to discover the patterns that include the optional attributes.

![Figure 12. Page-occurrence of the discovered patterns.](image)

After obtaining patterns from the downloaded pages as wrapper candidates, the wrapper generator needs to choose one specific pattern as the correct identification of the data object boundary and align other patterns to it in order to generalize one wrapper. As stated in section 4.4, we choose the pattern with the largest page-occurrence, i.e. the largest number of web pages in which the pattern is discovered. Consequently, we test the correctness of this heuristic. In Figure 12, we show the page-occurrence of each unique discovered pattern for each web site, where the y-axis is the number of patterns discovered. The height of each bar shows the total number of patterns discovered for each site. The different fill effects of each bar show the number of unique patterns discovered with the height of each fill effect indicating how many times this pattern was discovered (e.g., for site bguide, a total of 30 patterns were discovered, but among these there was only one unique pattern and this pattern occurred 30 times). The web sites with nested-structured data objects are shown in Figure 12(a) and the web sites with plain-structured data objects are shown in Figure 12(b). From these two figures, we also can see that most sites do have a single pattern that has a larger page-occurrence than other patterns, especially those web sites with plain-structured data objects. Moreover, we manually checked the chosen pattern with the largest page-occurrence for each web site to verify whether it correctly identifies the data object boundary or not. The results are encouraging as all of the chosen patterns do correctly identify the data object boundary.

Furthermore, we compared the performance of the IEPAD algorithm in [6] with our approach (labeled “C-repeat” in Figure 13) by employing the wrappers discovered by the two algorithms to extract data objects from 10 newly downloaded web pages from the web sites. Moreover, we also apply the DSE algorithm as a preprocessing step for the IEPAD algorithm. Figure 13 shows the results of the comparison, both in tabular form Figure 13(a) and graphically Figure 13(b), in terms of the number of discovered
patterns from which the wrapper is generalized. The recall is defined as the ratio of the number of correctly extracted data by the wrapper to the number of data objects the page covers.

In Figure 13, we can see that the number of patterns discovered by our approach is much less than the number of patterns discovered by the IEPAD algorithm, whether the IEPAD algorithm runs with or without the DSE algorithm. The reason is that the IEPAD algorithm is proposed for plain-structured data objects and it considers all maximal repeated patterns as its wrapper candidates no matter whether the pattern is “relevant” for the data object structure or not. By contrast, our approach is designed to extract nested-structured data objects by iteratively discovering C-repeated patterns. Moreover, we only obtain the patterns with the highest-nested level for each web page as wrapper candidates and discard the patterns with low-nested levels. Thus, we lessen the possibility to obtain “irrelevant” patterns. It should be noted that the IEPAD algorithm generalizes a wrapper by aligning its discovered patterns pair by pair, which has time complexity of $O(nM)$ with $M$ being the number of patterns and $n$ being the pattern length. By contrast, our approach only needs $O(Mn^2)$ time since pattern candidates are only aligned to the specific “center” pattern that correctly identifies the data object boundary.

The fourth column, labeled “Discarded” in Figure 13(a), shows the number of discovered patterns that are discarded when aligned to the “center” pattern due to too many token insertions. We can see that, after discarding some patterns, we further reduce the number of patterns that need alignment. From the recall in Figure 13(a), we can see that our approach outperforms the IEPAD algorithm, especially for the first six web sites whose data structures are nested. Additionally, our approach and the IEPAD algorithm can both achieve a precision of 1 for each web site. However, when using the IEPAD algorithm, the user needs to choose one of the generalized patterns as his/her extraction rule to extract data objects. Thus, the high precision of the IEPAD algorithm in this experiment is because we assumed that the user has chosen the right wrapper. On the contrary, our approach does not need this assumption since it only generates one wrapper for each web site and therefore is fully automatic and “truly” accurate.

Finally, we verify the effects of multiple string alignment on the wrapper induction and illustrate the results in Figure 14. The highest line and the lowest line represent the retrieval rate of the wrapper induced by our approach and the wrapper induced by the IEPAD algorithm, respectively, when both use multiple string alignment. Recall that in our approach, we choose the discovered pattern with the largest page-occurrence as the correct identification of the data object boundary and align other discovered patterns to it to generalize a final wrapper. Experimentally, when we choose the pattern with the largest page-occurrence as our wrapper to extract data objects from web pages without alignment, we attain the retrieval rate shown as the middle line in Figure 14. We can see that the effect of multiple string alignment varies from site to site. The retrieval rate drops a lot without multiple string alignment for CS Bibliography, while it remains the same for some other sites like BookGuide, Infoseek, Lycos and MSN. By carefully observing Figure 14 together with Figure 13, we can see that if a web site, such as CS Bibliography and Sasa.com, has data objects with optional attributes, the number of patterns discovered by
our approach will be large and the retrieval rate will not be high without multiple string alignment. Therefore, we suggest that, in general, multiple string alignment is necessary to achieve a high retrieval rate.

Figure 13. Comparison of our approach with IEPAD.
6. Related Work

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. As a consequence, some semi-automatic wrapper generation toolkits appeared mainly based on learning user labeled examples [1], [2], [4], [11], [12], [13], [14], [15]. Provided with manually labeled training examples, these systems could learn to generalize extraction rules and extract data from web pages. However, human-participation still meant that these systems had high cost in terms of time and efficiency.

Subsequently, researchers explored new approaches to fully automate wrapper generation without users’ training examples. [5] and [8] illustrate 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 starting tag and the corresponding ending 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. In addition, these two approaches identify the subtree with the highest fan-out as the data-rich section of a web page, i.e., the section or frame in the page that contains the main content of that page. However, as pointed out in Section 3 and illustrated in Section 5, this heuristic does not hold under some cases where the number of data objects contained in the web page is relatively small.

More recently, two different unsupervised approaches [6], [7] have been simultaneously proposed. Crescenzi et al. develop in [7] 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, when generalizing a wrapper each mismatch can be treated in two alternative ways, which results in the algorithm having an exponential time complexity.

Chang et al. propose a system called IEPAD in [6] 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.

Our approach is inspired by IEPAD, to some degree, since our work also treats documents as string sequences and extracts repeated substrings from them. However, we observe that the coding phase of IEPAD, which codes web pages as binary sequences, is not necessary. Rather than building a suffix-tree on characters in the alphabet, we build a specific suffix-tree on tokens, i.e., the HTML tags. Moreover, our approach tries to discover nested structures from the string sequence by introducing a new notion of C-repeated (continuous repeated) pattern. The number of patterns discovered by our approach is much fewer than that discovered by the IEPAD algorithm; thus our approach has fewer magic tuning constants than does the IEPAD algorithm.

7. Conclusion

The World Wide Web is a huge data pool with HTML pages containing a large amount of information. To discover extraction rules or induce wrappers for these semi-structured pages is one of the most significant issues in information extraction from the Web. Earlier works were mainly based on machine learning techniques, and therefore, are time-consuming and need human participation. We propose an innovative approach to generate wrappers and retrieve data objects from web pages. In our approach, data-rich sections of web pages are first identified, extracted, and considered to be token sequences composed of HTML tags and text strings. Then, a classical data structure used in string matching problems, the suffix-tree, is utilized to discover repeated patterns from the token sequences. Additionally, a new concept, C-repeated pattern, is defined and utilized to discover nested data object structures. As shown by our experiments, the whole process is fully automatic, fast (less than 10 seconds per page) and accurate.

For future work, we would like to investigate two interesting problems. First, how do we automatically locate the web sites containing structured data of interest to the user? Second, how do we automatically
annotate or label the fields of the extracted data? The second question is especially crucial since we cannot integrate or even query the extracted data without knowing what they represent.

References

