Automatically Maintaining Wrappers for Semi-Structured Web Sources

Juan Raposo, Alberto Pan, Manuel Álvarez
Department of Information and Communication Technologies. University of A Coruña.
{jrs,apan,mad}@udc.es

Justo Hidalgo
Denodo Technologies Inc.
jhidalgo@denodo.com

ABSTRACT

In order to let software programs gain full benefit from semi-structured web sources, wrapper programs must be built to provide a “machine-readable” view over them. Wrappers are able to accept a query against the source and return a set of structured results, thus enabling applications to access web data in a similar manner to that of information from databases. A significant problem in this approach arises as Web sources may undergo changes that invalidate the current wrappers. In this paper, we present novel heuristics and algorithms to address this problem. In our approach the system collects some query results during normal wrapper operation and, when the source changes, it uses them as input to generate a set of labeled examples for the source which can then be used to induce a new wrapper.

1 INTRODUCTION

1.1 Semi-structured Web sources

A substantial subset of Web data follows some kind of underlying structure. For instance, the pages obtained as a response to a query executed through a Web query form, are usually generated by a program that accesses structured data in a local database and embeds them in a HTML template. These Web sources are usually called *semi-structured sources*.

In many cases, semi-structured Web sources contain highly valuable information that cannot be easily accessed by other means. For instance, data may be provided by an autonomous organization that only provides Web access for its information. On other occasions, directly accessing the back-end database may be difficult or inconvenient.

1.2 Web wrappers

A program that is able to provide software applications with a structured view of a semi-structured Web source is usually called a *wrapper*. Wrappers receive queries from calling applications and automatically execute them on the source. Answering a query usually involves automatically filling in some Web form and extracting the data from the returned web pages, thus delivering a set of structured results to the calling application. For instance, Fig. 1 outlines the process followed by a wrapper to execute a certain query on an Internet bookshop.

Since wrappers enable applications to access Web data in a similar manner to that of information from databases, they are a key part of most current Web data integration architectures such as mediator systems [22].
Automatic and semi-automatic generation of wrappers has been an active research field for some years. Most works have focused on step 3 of Fig. 1: the problem of extracting structured data records from HTML pages [2,4,5,6,7,8,9,10,12,13,15,19,20,21,23,24] (see [14] for a survey). Wrappers leverage on the observation that semi-structured websites encode data records according to a consistent HTML template. In this context, by HTML template, we mean the particular way used to encode in HTML the data records obtained from the underlying database for presentation in the web browser.

Therefore, it is feasible to leverage on the regularities driven by the common template to generate extraction rules (which exact nature is specific to each wrappers system), able to parse the desired data from the target pages.

An approach to wrapper generation of particular interest to our case is called wrapper induction [7,9,10,12,13,19]. In this approach, the wrapper creator manually labels some data records in a set of sample pages from the source. These data records are then used as labeled examples by the system to automatically learn how to extract the desired data from other pages following the same template.

Fig. 2 sketches the process needed to generate a wrapper able of extracting data from an Internet bookshop using the wrapper induction approach.

1.3 Automatically maintaining wrappers

The main problem with wrappers arises when the Web sources change the underlying HTML template of their pages. Since the extraction rules used by wrappers are typically dependent of the particular template used, changes on it will typically make the current wrapper stop working properly. This is called the maintenance problem. In this paper, we deal with this issue.
Our approach for automatic maintenance is based on collecting some of the query results obtained during wrapper operation. When the source changes\(^1\), these results are used by a novel algorithm based on a series of heuristics, to generate a new set of labeled examples that can then be used to generate a new wrapper using induction techniques (i.e. once they have been labeled, the collected results act as user-provided examples during the initial wrapper generation). Fig. 3 sketches the process stages.

Thus, the automatic maintenance process comprises the following four stages in our approach:

1) Collecting results. While the wrapper answers queries, it stores the results from some of them.

2) Wrapper verification. The system periodically checks whether the wrapper is still valid or not.

3) Generating a new training set. When it has been determined that the wrapper is no longer valid, the system must generate a new set of labeled examples to regenerate the wrapper. Having as input the query results stored in stage 1 and the web pages that constitute the current response in the source to the stored queries, this stage outputs a set of examples labeled in the response pages.

4) Regenerating wrappers. From the examples generated in the previous stage, the new wrapper can be re-generated using a wrapper induction method.

The main focus of this paper is the third stage, although we will also describe the methods we use for the simpler first one. Methods for stage 2 have been addressed in [11,16]. In stage 4, we

\(^1\) In the remaining of the paper, references to changes on the source refer to those in the HTML template, being the type of changes we deal with in this paper.
use our own induction algorithm (see section 7), although our implemented system could also be easily adapted to another one.

1.4 Organization of the paper

The rest of the paper is organized as follows. Section 2 describes how we model wrappers and what assumptions our approach makes about their nature. Section 3 describes the methods used to select the query results that must be stored during wrapper operation (stage 1 of our approach). Sections 4, 5 and 6 are the core of the paper and contain our main contributions: they present the novel heuristics and algorithms that constitute our approach for generating the new set of examples from the stored query results (stage 3). Section 4 formally states the problem. Section 5 describes our approach for the base case where the data records extracted by the wrapper may be modeled as a type exclusively composed of atomic fields. Section 6 generalizes those techniques to deal with types of arbitrary depth. Section 7 overviews our wrapper induction algorithm for regenerating the wrapper from the new set of examples.

We have experimented with our techniques in several important real-world Web data extraction problems and have found them to be highly accurate. These experiments are described in Section 7. Section 8 discusses related work, and section 9 summarizes our conclusions.

2 WRAPPER MODEL

2.1 Data model

In this section, we describe the data model we use to represent the results returned in response to a query on a semi-structured web source. We consider each query result as belonging to a structured type [1]. A type is defined recursively as follows:
• A type can be atomic. In that case, its instances are character strings.

• If $t_1, ..., t_n$ are types, then we can define a new type $T<t_1, t_2, ..., t_n>$. $t_1, t_2, ..., t_n$ are called fields (or attributes) of $T$. An instance of $T$ is an ordered list of the form $<v_1, ..., v_n>$, where $v_i$ is an instance of $t_i$.

• If $t$ is a type, then we can define a new type $\{t\}$. An instance of $\{t\}$ is a set of elements $r_1, ..., r_m$. Each $r_i \in \{1..m\}$ is of type $t$.

**Example 2.1**: Fig. 4 shows some data records obtained in response to a query in an Internet music shop. We can model their schema as a type $ALBUM<TITLE, ARTIST, DATE, \{EDITION<FORMAT, PRICE>\}>$, where $TITLE, ARTIST$ and $DATE$ are atomic types and $EDITION$ is a compound type defined upon the atomic types $FORMAT$ and $PRICE$.

A type can be seen as a tree whose leaves are atomic types and the non-leaf nodes represent non-atomic types. Fig. 4 shows the tree of the $ALBUM$ type along with some data instances of it. Fields which are not direct descendants of the root will be usually referred to by using its complete path in the tree. For instance, in our example we would refer to $FORMAT$ and $PRICE$ fields as $EDITION.FORMAT$ and $EDITION.PRICE$.

This type definition is suitable for our purposes of modeling the query results obtained from semi-structured web sources, since they are typically obtained from an underlying structured database. In addition, query results in websites often present a hierarchical schema including multivalued fields; the compound types in our definition allow representing them.

### 2.2 Wrapper Execution Model: Extraction rules

In this section, we describe our assumptions about the nature of wrappers and their execution.

We define a wrapper as follows:
**Definition (wrapper of a given type):** A wrapper of type $T$ for a semi-structured web source $s$, is a function which receives as input the set of pages $P$ obtained as response to any query $q$ executed on $s$, and returns $R = \{r_1, \ldots, r_m\}$, the set of the query results contained in $P$, where every $r_i$, $i \in \{1, \ldots, m\}$ is of type $T$. □

Our techniques do not assume any particular technology for implementing wrappers. Instead, we model the execution of a wrapper as the application on $P$ of a set of extraction rules, each one of them able to return a subset of the query results. More precisely, we define an extraction rule as follows.

**Definition (Extraction rule of a given type):** An extraction rule of type $T$ for a semi-structured web source $s$, is a function which receives as input the set of pages $P$ obtained as response to any query $q$ executed on $s$, and returns $R = \{r_1, \ldots, r_k\}$, a subset of all the query results contained in $P$, where $r_i$, $i \in \{1, \ldots, k\}$ is of type $T$ contained in $P$. □

This definition captures the idea that an extraction rule is some software artifact able to extract some query results (not necessarily all) from the set of pages $P$ obtained as response to a query executed on $s$.

Typically, a wrapper will have a set of different extraction rules, each one able to extract query results formatted in a particular way. For instance, if some results have values for a field and others do not (that is, the value for the field is optional), a wrapper will typically need a different extraction rule for each kind of result.

It is important to notice that this definition does not assume any particular way of implementing extraction rules, which makes our techniques independent of the specific technology used for web data extraction. For instance, in some systems an extraction rule could be mapped to a set of Xpath expressions, each one extracting the data of a particular field of the query results, while
other systems may map an extraction rule to an expression or a set of expressions written in a language specifically designed for web data extraction tasks. Our system implements extraction rules as DEXTL patterns. DEXTL [20] is a language we have specifically designed to address web data extraction tasks. The central construction in DEXTL is the DEXTL pattern, which is conceptually similar to a regular expression, but adapted to the particular nature of HTML pages. See section 7 for a more detailed description of DEXTL.

3 STAGE 1: COLLECTING RESULTS FROM QUERIES

The objective of this stage is to collect the results from a subset of the queries executed by the wrapper during its normal operation so they can be used to generate a new training set of labeled examples when the source changes. The system works as follows:

1) When the wrapper answers a query, it stores up to a certain (customizable) number of results from the query in a local database. For each result collected, the system stores the following data: the query that generated it, the extraction rule used by the current wrapper to parse it and an expiration date computed from the current time.

2) The current content in the database is explored periodically to check whether the non-expired results are sufficient to satisfy certain requirements we will describe below or not. If so, all the expired results are removed. If not, then the system computes a minimum set of expired queries that must remain stored.

The requirements we want the stored results to satisfy are:

1) Storing at least a certain (customizable) number of query results. In general, the regeneration process will provide better output results when using a sufficient number of results. Our current implementation tries to store at least 100 results.
2) Storing results from at least a certain (customizable) number of different queries. A query may implicitly determine some properties of the response pages. For instance, an electronic shop may assign a given discount (e.g. 15%) to all the books by a given author. If we stored results only from the query ‘author=a’, the wrapper induction process could wrongly consider the string “Special discount: 15%” to be present in all the data records from the source, and this could lead to the generation of incorrect extraction rules for the new wrapper. Our current implementation stores results from at least 10 different queries.

3) Storing a minimum (customizable) number of results for each current visual representation of a result in the source. Most web sources use slightly different visual representations for the results of a query. For instance, a book may include some discount information that another book does not. The wrapper will usually need a different extraction rule to extract data records using each visual representation. Therefore, by storing the extraction rule used by the current wrapper to obtain a given result, we can tell if two query results have the same visual representation in the source or not. When the source changes, the wrapper regeneration process will usually benefit from having examples of each visual representation, in order to generate the extraction rules needed to parse all the data records. We have found it is very useful to store at least a minimum number of results using each former visual representation (before the change on the source) in order to increase the probability that the stored results also cover all the new visual representations (after the change). In our implementation, we have set this value to 5.
4 LABELING QUERY RESULTS: PROBLEM DEFINITION

Our approach for wrapper maintenance is based on collecting some of the query results obtained during wrapper operation. When the source changes, these results are used to generate a new set of labeled examples that can then be used to generate a new wrapper using induction techniques.

In this section, we formally define the concept of labeled example and state the problem of generating them from the set of stored query results.

For the remaining of this section, let \( Q = \{ q_1, \ldots, q_n \} \) be the set of queries whose results were collected during wrapper operation. We denote the stored results for a query \( q_i \) as \( R_i = \{ r_{i1}, \ldots, r_{im} \} \) and the set of pages that form the current response to \( q_i \) in the target source as \( P_i \).

4.1 Candidate Occurrences and Labeled Examples

In this section, we introduce two key concepts of our approach: candidate occurrences of a query result and labeled examples.

**Definition (value list):** The value list of a stored result \( r_{ij} \in R_i \) is the list of atomic values of \( r_{ij} \), and it is notated as \( V_{ij} = \{ v_{ij(1)}, \ldots, v_{ij(f)} \} \).

**Example 4.1:** Let us consider the result \( r \) shown in Fig. 4. Its value list is \( V = \{ "In Through the out door", "Led Zeppelin", "8/1994", "CD", "$11.05", "$29.28", "$11.05" \} \).

**Definition (candidate occurrence):** A candidate occurrence in a page \( p \in P_i \) for a query result \( r_{ij} \in R_i \) is of the form \( c = \{ ( v_{ij(l)}, pos_l), \ldots, ( v_{ij(f)}, pos_f) \} \), where \( (v_{ij(k)}, pos_k)_{k \in \{1..f\}} \) denotes the \( pos_k \)-th occurrence in \( P \) of the atomic string \( v_{ij(k)} \in V_{ij} \). Each \( (v_{ij(k)}, pos_k)_{k \in \{1..f\}} \) is called a candidate field occurrence.

**Example 4.2:** Let us consider a stored result \( r_{ij} = \langle TITLE = "Beginning XML, Second Edition", AUTHOR = "David Hunter et al", FORMAT = "Paperback", PRICE = "27.19" \rangle \) of type \( T = \)
<TITLE, AUTHOR, FORMAT, PRICE>, where TITLE, AUTHOR, FORMAT and PRICE are atomic. Thus, the value list of \( r_{ij} \) is \( V_{ij} = \{ "Beginning XML, Second Edition", "David Hunter et al", "Paperback", 27.19 \} \). Fig. 5a shows an HTML page \( p \) and a simplified version of its HTML code. \( p \) contains several candidate occurrences of \( r_{ij} \). We annotate the occurrences on \( p \) of each atomic value in \( V_{ij} \) subscripted with its relative order in \( p \). For instance, the value “Beginning XML, Second Edition” appears twice on the page, and so its appearances are marked as “Beginning XML, Second Edition_1” and “Beginning XML, Second Edition_2”.

Fig. 5b shows some of the candidate occurrences of \( r_{ij} \) on the page.

**Definition (correct candidate occurrence / labeled example):** A candidate occurrence \( c = \{(v_{ij}(1), pos_1), \ldots, (v_{ij}(f), pos_f)\} \) in a page \( p \) for a query result \( r_{ij} \) is correct iff \( v_{ij}(k), k=1..f \) are part of the same data record in \( p \). In that case, \( c \) is also called a labeled example.

In this definition, by data record we mean a fragment of \( p \) representing a query result from the underlying database. For instance, the page shown in Fig. 5a has four data records, each one corresponding with the information from a book (one on the ‘Most popular results’ list and three on the ‘All results’ list). Although this data record definition is not formal but semantic, humans are usually able to identify data records in a web page unambiguously, so it is enough for our purposes.

**Example 4.3:** Let us consider again \( r_{ij} = <TITLE = "Beginning XML, Second Edition", AUTHOR, "David Hunter et al", FORMAT = "Paperback", PRICE="27.19"> \) from Example 4.2. Fig. 5b shows some of the candidate occurrences of \( r_{ij} \) on the page. Candidate occurrences \( c_1 \) and \( c_3 \) in our example are not correct, since they are formed of values from different books. For instance, candidate occurrence \( c_3 \) takes the price value from the book titled ‘Learning XML, Second Edition’, whereas the values for the remaining fields are taken from the book ‘Beginning
In turn, occurrence $c_2$ is correct and, therefore, it is a valid labeled example for the re-induction stage. □

It is important to notice that our definition of correct candidate occurrence does not exclude the possibility of having more than one correct candidate occurrence for the same query result. This happens when a query result is shown in two different result listings in the same page. In those cases, we aim to privilege the list that contains the higher number of the stored query results, since it increases the probability of choosing the same list that was being used by the previous wrapper.

**Example 4.4**: Candidate occurrence $c_4$ in Fig. 5b is a correct candidate occurrence for $r_{ij}$ which corresponds with its appearance on the list of ‘most popular results’, while $c_2$ is a correct candidate occurrence which corresponds with its appearance on the complete list of results. Let us suppose the complete list of results contains more data records corresponding with stored results of $q_i$ than the ‘most popular’ list, then we would choose $c_2$.

**4.2 Obsolete Query Results**

In this section, we introduce the concept of *obsolete query result*.

**Definition (obsolete query result)**: A stored query result $r_{ij} \in R_i$ is obsolete in $P_i$ iff no $p \in P_i$ contains any correct candidate occurrence of $r_{ij}$.

**Example 4.5**: Fig. 6 shows a sample page where there exist several candidate occurrences of the query result $r_{ij}$ from Example 4.2. Nevertheless, none of these candidate occurrences is correct. Let us assume that the page shown is the only one in $P_i$ containing candidate occurrences of $r_{ij}$. Then, we can conclude that $r_{ij}$ is an obsolete example in $P_i$. □

We have to deal with obsolete query results because there will be some time interval between the moment the query results are stored and the moment we try to regenerate the wrapper. Although
this time interval will not usually be longer than a few hours (automated verification methods can be used to detect the change), the data on the source may have already changed. For instance, in our example bookshop of Fig. 6, the price of the book changed, turning obsolete the query result corresponding to it.

Note that the main assumption of our approach is that we can usually be confident that a sufficient number of the stored query results contained in $R_i$ is still present in $P_i$ after the change in the source, and, therefore, they can be used to generate a sufficient number of labeled examples for the wrapper re-induction stage. It should be noted that in those cases where the values of the query results to be extracted from the source vary very rapidly (e.g. stock quotes information) this assumption could be wrong. In section 5.6, we will describe how our approach can be extended to deal with those sources.

### 4.3 Problem Statement

**Labeling Problem**: Let $Q=\{q_1, \ldots, q_n\}$ be the set of queries on a source $s$ whose results were collected during wrapper operation. Let $R_i=\{r_{ij}, \ldots r_{ijn}\}$ be the set of stored results of the query $q_i$, and $P_i$ be the set of pages that constitute the current response to $q_i$ on $s$. The labeling problem is stated in the following way: for each $r_{ij}, i=1..n,j=1..m \in R_i$, find (when exists) $c_{ij}$, a correct candidate occurrence for $r_{ij}$ in $P_i$. If several correct candidate occurrences exist for the same $r_{ij}$, then privilege the one which is part of the list having the higher number of data records which corresponds with correct candidate occurrences of other query results $r_{ik}, k \neq j$.

### 5 Labeling Query Results of Flat Types

This section describes the basic heuristics and algorithms of our approach for the base case where the query results are modeled as a type exclusively composed of atomic fields.
The pseudo-code of the algorithm to label a query result $r_j \in R_i$ in $P_i$ is shown in Fig. 7.

Overall, the algorithm computes all the candidate occurrences of $r_j$ in $P_i$ (if their number is higher than a certain limit, then the most promising ones are chosen), and ranks them according to three heuristics: the “sibling nodes” heuristic, the “pattern” heuristic and the “proximity” heuristic. The candidate occurrence that gets the best ranking is considered as correct and, therefore, will be outputted as a labeled example for the re-induction stage.

The algorithm uses the “sibling nodes” heuristic as the main criterion for choosing the best candidate occurrence. Nevertheless, there are some cases where it is inapplicable. In that case, the “pattern” heuristic is used as the main criterion. The “proximity” heuristic is used only when the previous heuristics return the same value for several candidates.

The algorithm also needs to detect obsolete query results: since these results do not have any correct candidate occurrence, we should detect and discard them.

Sections 5.1, 5.2 and 5.3 respectively describe the heuristics. Section 5.4 addresses the problem of pre-selecting candidate occurrences when their number is too high to compute the heuristic values for all of them. Section 5.5 deals with the detection of obsolete query results. Finally, section 5.6 describes the extensions needed for our approach to deal with the case of sources having data fields that vary very rapidly in time.

5.1 The “Sibling nodes” Heuristic

In this section, we describe the “sibling nodes” heuristic, which is one of the main criteria we use to find the correct candidate occurrences for the stored query results. We begin by stating the observations it is based on.

For the remaining of this section, the discussion is in the context of $R$, the set of stored results to a query $q$, and $P$, the set of pages that constitute the current response to $q$ in the target source.
**Observation 5.1:** The results obtained as response to a query in a semi-structured source are usually organized as a list. In addition, they are formatted according to the same underlying HTML template. This observation also applies when the query results are distributed on several pages. For instance, if each page contains only one query result (e.g. the detail page for each book in an electronic bookshop), the observation still applies by simply considering a single “aggregated page” containing all the original pages.

**Observation 5.2:** Given the DOM tree of an HTML page containing a list of data records, each data record is usually represented using one sub-tree that is sibling of the sub-trees corresponding with the other data records on the list. We say two sub-trees are *siblings* if their root nodes have the same Xpath route from the root.

**Example 5.1:** Fig. 8 shows an excerpt of the DOM tree of the HTML page of Fig. 5a. As can be seen, each book on the list is represented in one sibling sub-tree.

### 5.1.1 Symmetrical candidate occurrences

In this section, we introduce the concept of *symmetrical* candidate occurrences. This concept will be used in the sibling nodes heuristic to help identifying correct candidate occurrences.

**Definition (symmetrical candidate occurrences):** Given a candidate occurrence $c_1=$\{(v$_{i1}$, pos$_{i1}$), ..., (v$_{if}$, pos$_{if}$)\} for a query result $r_i \in R$ and a candidate occurrence $c_2=$\{(v$_{j1}$, pos$_{j1}$), ..., (v$_{jf}$, pos$_{jf}$)\} for a result $r_j \in R$, we say $c_1$ and $c_2$ are *symmetrical* when:

1. For every $k \in \{1..f\}$, the Xpath route in the DOM tree of $(v_{ik}, pos_{ik})$ is the same as the Xpath route of $(v_{jk}, pos_{jk})$.
2. For every $k \in \{1..f\}$, $(v_{ik}, pos_{ik}) \neq (v_{jk}, pos_{jk})$. This ensures $c_1$ and $c_2$ do not share any field occurrence.
3. Given $k \in \{1..f\}$ and $m \in \{1..f\}$, if $v_{ik}$ appears in $c_1$ before $v_{im}$ (considering depth-first traversal order), then $v_{jk}$ appears in $c_2$ before $v_{jm}$. □

This definition captures the intuition of two candidate occurrences formatted according to a consistent template. In particular, it ensures that values for the same field across different query results are formatted in the same way. It also ensures the ordering of the field values is consistent (e.g. if the TITLE appears before the AUTHOR in $c_1$, the same should happen with $c_2$). □

**Observation 5.3:** Given two query results $r_i \in R$ and $r_j \in R$, the correct candidate occurrence of $r_i$ and the correct candidate occurrence of $r_j$ are symmetrical.

### 5.1.2 Minimum Sub-trees and Incorrect Candidate Occurrences

In this section, we define the concept of *minimum sub-tree* of a candidate occurrence and introduce one observation that will be used by the sibling nodes heuristic to detect incorrect candidate occurrences.

**Definition (minimum sub-tree of a candidate occurrence):** Given a page $p \in P$, and a candidate occurrence $c = \{(v_{i1}, pos_1), \ldots, (v_{if}, pos_f)\}$ for a query result $r_i$, the minimum sub-tree of $c_1$ is the deepest sub-tree in the DOM tree of $P$ which contains every $v_{ij}, j=1..f$.

**Example 5.2:** The sub-tree $s_1$ in Fig.9 is a minimum sub-tree of the candidate occurrence $c_1$. □

**Observation 5.4:** Let $s_1$ be the minimum sub-tree containing a candidate occurrence $c_1$ for a query result $r_i$, and let $S = \{s_1, \ldots, s_t\}$ be the set of sibling sub-trees of $s_1$ (including $s_1$ itself). When observation 5.2 is true, if any $s_j \in S$ contains candidate occurrences of two or more different query results, then $c_1$ is usually not a correct candidate occurrence.

**Example 5.2:** Fig. 10a shows a candidate occurrence $c_1$ for a query result $r_i$. $c_1$ is incorrect because it spans two data records. The minimum sub-tree containing $c_1$ also contains candidate occurrences for query results $r_3$ and $r_4$. □
Fig. 10b also shows an incorrect candidate occurrence \( c_1 \) for a query result \( r_1 \). \( s_1 \) is the minimum sub-tree of \( c_1 \). Its sibling sub-tree \( s_2 \) contains candidate occurrences of \( r_2 \) and \( r_4 \).

The justification for the above observation is the following. Given a candidate occurrence \( c \) for a query result \( r_i \), and \( s \), the minimum sub-tree containing \( c \):

- If \( c \) is correct then, when the observation 5.2 applies, by definition \( s \) and its sibling sub-trees will only contain one data record. Therefore, there is a very small chance of finding candidate occurrences of several different query results in \( s \) or its siblings.
- In addition, if \( c \) is incorrect because it spans several data records, \( s \) will contain several data records of the list (at least, all the data records \( c \) spans). Therefore, there is a good chance of finding in \( s \) candidate occurrences of a query result \( r_j \), \( j \neq i \).

5.1.3 Formulation of the Sibling Nodes Heuristic

The “sibling nodes” heuristic performs the following steps to rank a candidate occurrence \( c \) of an example \( r_i \in R \):

1. Find \( s \), the minimum sub-tree containing \( c \).
2. Let \( \{s_1, \ldots, s_t\} \) be the sibling sub-trees of \( s \). In each \( s_{p\in\{1..t\}} \), search for candidate occurrences of other query results \( r_j \in R \), \( j \neq i \), which are symmetrical to \( c \). The obtained number will be the “ranking” for \( c \).
3. In addition, according to observation 5.4, if candidate occurrences for two or more different query results are found in \( s \) or in any \( s_{p\in\{1..t\}} \), then \( c \) is ranked 0.

Example 5.3: Let us consider the example shown in Fig. 9. The figure shows two candidate occurrences \( c_1 \) and \( c_2 \) for the result \( r_1 \) along with the set of stored results \( R=\{r_1,r_2,r_3\} \) and the DOM tree of a page containing some results from \( R \). \( c_1 \) is a correct occurrence of \( r_1 \) while \( c_2 \) is not (since it comprises values from two different data records).
The sub-tree $s_1$ is the minimum sub-tree containing $c_1$. As can be seen, its sibling sub-trees $s_2$ and $s_3$ contain candidate occurrences of, respectively, $r_2$ and $r_3$, and they are symmetrical to $c_1$; consequently, the ranking for $c_1$ according to the sibling nodes heuristic is 2.

In turn, the minimum sub-tree containing $c_2$ is $s_4$, which has as root a node which is “too high” in the DOM tree and, therefore, its siblings ($s_5$,...) do not contain candidate occurrences of any other examples; thus, the ranking for $c_2$ will be 0.

Another reason for ranking $c_2$ with the value 0 comes from the fact that $s_4$ contains candidate occurrences for $r_1$, $r_2$ and $r_3$. Therefore, according to observation 5.4, $c_2$ should also be ranked 0.

A final remark regards to the cases when there are more than one correct candidate occurrence for the same query result (recall Example 4.4). The ranking the sibling nodes heuristic assigns to a candidate occurrence rises as we find symmetrical candidate occurrences of other query results in sibling sub-trees. Therefore, it tends to privilege those candidate occurrences that are part of the list having the higher number of data records corresponding with other query results, thus fulfilling our objective in dealing with these cases.

5.1.4 Applicability of the sibling nodes heuristic

According to our experiments the sibling nodes heuristic is very effective. Unfortunately, it is not always applicable. In some relatively rare cases, the DOM tree of the page is built in such a way that there is not a unique sub-tree containing each data record of the list, but instead all data records share the same minimum sub-tree. Fig. 11 shows an example of this situation.

In these cases, even the correct candidate occurrences will rank zero in the heuristic since we will always find candidate occurrences of more than one result in the minimum sub-tree of the occurrence.
Therefore, when the sibling nodes heuristic returns zero for every occurrence of every result, we consider the sibling nodes heuristic is inapplicable and the pattern heuristic will be used.

5.2 Pattern Heuristic

The pattern heuristic also exploits the observation 5.1.

In this case, to rank a candidate occurrence $c$ for a query result $r_i \in R$, we compute from $c$ a regular expression-like “pattern” trying to capture its template.

The pattern will contain placeholders in the positions where the values of $c$ appear. Then we will apply the pattern to $P$, obtaining a set of matches, where each match is composed of one string value for every placeholder in the pattern. The basic intuition is that if the candidate occurrence is correct, then the values extracted for some of the matches will coincide with the values from other query results from $R$. The number of query results found will be the ranking assigned to $c$ by the heuristic.

To compute the pattern for $c$, we traverse the tree in deep-first order from the first value of the occurrence to the last, adding each found tag and string to the pattern. We substitute the atomic values of the query result by the placeholders $\texttt{FIELD\_NAME}$ and the rest of the strings by the special token $\texttt{ANY}$. To generate the pattern we ignore some tags that we have found of little relevance to capture the basic template of the occurrence such as font, span, bold or italic tags. The patterns are applied much in the same way as regular expressions. The placeholders in the pattern match with any string on the page, while a tag in the pattern matches with any tag in the page of the same kind.

Example 5.4: Fig. 12 shows two candidate occurrences $c_1$ and $c_2$ for the result $r_1$. $c_1$ (the correct occurrence) has been used to generate the pattern $P_1$ and $c_2$ has been used to generate the pattern $P_2$. 
Applying $P1$ on the page, we obtain the matches shown in the lower part of the figure. The values of the first and second match coincide with the results $r_2$ and $r_3$, so the ranking for $c_1$ will be 2.

In turn, the match obtained by the pattern $P2$ generated from $c_2$ does not coincide with any other stored result, so the ranking for $c_2$ will be 0.

The pattern heuristic is similar in spirit to the sibling nodes heuristic, but it is slightly less effective, because the generated pattern may miss stored query results matches that actually appear on the page but with a slightly different template. For instance, in Fig. 12 the pattern generated from $c_1$ does not match with $r_4$, and $c_1$ gets a ranking of 2, whereas ideally it should be 3. Nevertheless, this is not a major drawback in practice because we only need enough matches to discriminate between correct and wrong occurrences (one only match is enough in most cases).

### 5.3 Proximity Heuristic

The proximity heuristic is based on the following observation:

**Observation 5.5:** All the data values belonging to a certain data record tend to be grouped near each other on the page.

In conformance with this observation, the candidate occurrences whose values are sparse on the page should be considered less promising.

We can measure the “sparseness” of a candidate occurrence $c=\{(v_{i1}, pos_1), \ldots, (v_{if}, pos_f)\}$ for a query result $r_i$, by computing the sum of the distances between every two consecutive values of the occurrence. The distance between two values is computed as the number of nodes between the last node of the first value and the first node of the next value (depth-first traversal order).
Another possibility for implementing this heuristic is actually computing the average “visual distance” on the page rendered by the browser between the fields of the candidate occurrence. The visual distance between two DOM nodes may be computed by using the functionalities provided by current browser APIs. In our tests, the DOM distance measure performs almost as well as the visual distance “measure”. Thus, for convenience and efficiency reasons, our current implementation uses the DOM distance measure.

5.4 Detecting obsolete query results

Obsolete query results were defined in section 4.2. We need to detect them to avoid generating incorrect labeled examples to the re-induction stage. We apply the following criteria to detect them:

1. Trivially, query results without any candidate occurrence are obsolete.
2. When the sibling nodes heuristic is applicable, the results that have a zero value in that heuristic for every candidate occurrence are rejected.
3. When the sibling nodes heuristic is inapplicable, we analyze overlaps on the page between the best-ranked candidate occurrences of several results. If the best candidate occurrences of two results overlap, then one of them is very probably wrong, since sources do not show a data record mixed with another. The criterion we use is rejecting the result which best candidate occurrence has a lower rank according to the proximity heuristic.

**Example 5.5**: Fig. 13 shows two results $r_1$ and $r_2$ and their best candidate occurrences $c_1$ and $c_2$ on a sample page. Since $c_2$ and $c_1$ overlap, the system detects that one of them must be wrong. We would reject $c_1$ because it ranks worse according to the proximity heuristic. □
5.5 Dealing with too many candidate occurrences

Sometimes the number of candidate occurrences of a result is too high and it is not possible to compute the heuristic values for all of them. In these cases, the algorithm needs to prune the number of generated candidate occurrences (step 3.a in Fig. 7).

The pre-selection process is based on the following steps: given a certain number $N$, which is an upper limit to the number of candidate occurrences we wish to generate, and a query result $r_i=<\text{field}_1=v_1,\ldots,\text{field}_f=v_f>$ of type $T<\text{field}_1,\ldots,\text{field}_f>$:

1. Find $v_i$, the value of the field of $r_i$ that has fewer occurrences on the page. The number of occurrences of $v_i$ will be denoted $|v_i|$.

2. Choose $k$, the maximum number such that $k^{r-1}|v_i| \leq N$. Since the total number of generated candidate occurrences for $r_i$ will be equal to the product of the number of candidate occurrences of each field, this will allow us to choose up to $k$ occurrences of every other value $v_j, j \neq i$ for each occurrence of $v_i$, without surpassing the upper limit $N$.

3. For each occurrence $o$ of $v_i$ on the page:

   a. For every $v_j, j \neq i$ of $r_i$, choose at most $k$ occurrences of it on the page. If there are more than $k$ occurrences of $v_j$, the metrics used to select the $k$ best ones are:

      i. The proximity with respect to $o$ in the DOM tree, using depth-first order.
      ii. The length of the common path in the DOM tree between the occurrence of $v_j$ and $o$. The longer the common path, the higher the occurrence of $v_j$ is rated.
5.6 Dealing with Highly Time-Variable Data

When data that are to be extracted from a source vary at very short time intervals (e.g. stock quotes), we cannot assume any query result to remain completely valid in that source. In these situations, the system computes the candidate occurrences in a different way. We distinguish between the fields that are highly variable (e.g. the current change of a stock quote) and those we can assume will remain more stable (e.g. the company acronym):

- For “stable” fields, we will compute the candidate field occurrences by using stored query results in the same way as in conventional sources.
- For “highly variable” fields, we will instead compute the candidate field occurrences by using a regular expression describing the format of the expected values (e.g. \([0-9]+ \cdot [0-9]+\) for the current change of a stock quote). Regular expressions are provided by the user during wrapper generation and they are stored by the maintenance system to be used whenever the source changes. We also allow expected prefixes and suffixes to be included in the expression for the data values.

**Example 5.6**: Fig.14 shows a sample website with one stable field (symbol,…) and a two highly variable fields (trade, %chg). It also shows the regular expression defined for finding candidate field occurrences of trade and %chg and two candidate occurrences \(c_1\) (the correct one) and \(c_2\) for a result \(r_1\).

As we will remark in the experience section of the paper, these modifications make the techniques described in the previous sections maintain their effectiveness in this kind of sources. For instance, the techniques would correctly choose \(c_1\) as the correct candidate occurrence in Example 5.6.
6 Generalization to Types of Arbitrary Depth

In this section, we show how to generalize the *obtainBestAtomicCandidateOccurrences* algorithm (Fig. 7) to results of types of arbitrary depth. The main idea is based on applying the algorithm recursively to the tree of the result.

Let us consider the tree for a result with depth $k$. At the $(k-1)$th level, all the non-atomic fields of the result are composed exclusively of atomic fields. Thus, they can be considered as if each of them were a result belonging to a type of depth 1.

For instance, in Fig. 15a two results of type $T<TITLE, ARTIST, DATE, \{EDITION<FORMAT, PRICE>\}>$ are shown. At the $(k-1)$th level such data elements as $R1.EDITION2=<format="LP", price="18.61">$ from Result 1, and $R2.EDITION2=<format="LP", price="17.34">$ from Result 2 may be considered as two results of type $EDITION<FORMAT, PRICE>$. We will call these data elements “sub-results of $(k-1)$th level”.

Thus, as a first step we can consider every input result, obtain their sub-results of $(k-1)$th level and apply the *obtainBestAtomicCandidateOccurrences* algorithm to them to obtain their best candidate occurrences.

It is important to note that the *obtainBestAtomicCandidateOccurrences* algorithm may return more than one “best” candidate occurrence for each sub-result. For instance, in Fig. 15a the candidate occurrences marked as $c_1$ and $c_2$ will be both returned as “best” candidate occurrences for both sub-results $R1.EDITION1$ and $R2.EDITION1$.

The following step is to substitute in $P$ the minimum HTML code containing each “best candidate occurrence” of the sub-results of $(k-1)th$ level by a special unique markup on the page.

For instance, in Fig. 15b the best occurrences of the sub-results of type $EDITION$ have been substituted on the page for special markups of the form $SEDITION-I$, where $I$ is a number. If two
sub-results are formed by exactly the same data values and both are present on the page, then they will share at least two best candidate occurrences, which will be substituted on the page by the same special markup. For example, in Fig. 15b the occurrences c₁ and c₂ are both substituted for the special markup $EDITION1$.

By performing this process for the (k-1)th-level sub-results of all the input results, the best candidate occurrences for non-atomic sub-elements of level k-2 can then be computed by simply invoking obtainBestAtomicCandidateOccurrences with this representation of P, and using the special unique markups as values for the new atomic fields of (k-2)th level (which have substituted the former non-atomic fields of (k-1)th level).

For instance, in Fig. 15b the sub-results of (k-2)th level for, respectively, R₁ and R₂ are:


Note how atomic fields of the form EDITION[i] have replaced the non-atomic field EDITION in the results.

It is important to notice how the ambiguity we had at (k-1)th level, where the candidate occurrences c₁ and c₂ were both considered as “best candidate occurrences” for both R₁.EDITION₁ and R₂.EDITION₁, is solved at this level. The context provided by the other fields of the results will allow the sibling nodes, pattern and proximity heuristics to determine c₁ as the only “best candidate occurrence” for R₁.EDITION₁ and c₂ as the only “best candidate occurrence” for R₂.EDITION₁.
By repeating recursively this process until the $0th$ level, we will obtain the best candidate occurrences for the complete results. The algorithm is sketched in Fig. 16.

## 7 WRAPPER INDUCTION

In this section we overview our algorithm to reinduce the wrapper. In the section 7.1, we provide a quick introduction to DEXTL (which stands for Data Extraction Language), the wrapping language we use (see [20] for detail). In the section 7.2 we describe the induction process which takes care of how the DEXTL programs are automatically generated from a set of labeled examples.

### 7.1 DEXTL Overview

A DEXTL program is composed of hierarchically structured *DEXTL elements*. Typically, a program for extracting data records of a given type $T$ will have a *DEXTL element* for each field from $T$. The DEXTL elements will also be hierarchically arranged in the same way that the fields they represent are related in $T$.

Thus, each DEXTL element can be either atomic (those without sub-elements) or non-atomic (those with sub-elements). Each non-atomic element has (among others not relevant here) the following parts: *EXTRACTION* clause (mandatory) and *UNTIL* clause (optional).

The *EXTRACTION* clause relates an element with its sub-elements and specifies how they are laid in the target source pages. This clause will be the basis to identify and extract the element data instances in the document. The *UNTIL* clause delimitates the end of the region of the document where the occurrences of the element must be searched for.
The basic structure used to specify the *EXTRACTION* and *UNTIL* clauses is called a *DEXTL pattern*. As we will see, a DEXTL pattern defines a sequence of data elements and separators among them.

The *EXTRACTION* clause will typically contain one DEXTL pattern for each possible visual layout of the element inside the target pages. The *UNTIL* clause uses DEXTL patterns to locate the end of the search region for the element. When the system is extracting instances of an element and an occurrence is found for a DEXTL pattern from its *UNTIL* clause, the system stops looking for occurrences of that element and continues searching for occurrences of its parent element (or finishes if there is not a parent element).

### 7.1.1 DEXTL Patterns

A DEXTL pattern is comprised of a list of *text tokens* which are laid consecutively in the document and which are delimited by *tag-separators*.

*Text tokens* represent text in the browser-displayed page. They are enclosed between '[' and ']' and they can be divided into portions by applying Perl5-like regular expressions. A placeholder (prefixed by the 'S' character) can be assigned to the parts of the regular expression enclosed between '(' and ')'. The name may correspond either with an atomic field of the elements we wish to extract or with a special value called *IRRELEVANT*, which is used to represent non-constant strings appearing in the pattern, but which we do not wish to extract.

*Tag-separators* represent a regular expression concerning HTML document tags. For instance, we could define a tag-separator called *EOL* (EndOfLine) as follows:

```
EOL = ("<br>"| "</p>"| "</tr>" | "</td>" (\[ \n\r\t\])* </tr>)
```

Though they can be defined to suit, DEXTL includes built-in tag-separators which are enough for the vast majority of situations concerning wrapper generation. Nevertheless, as we will see
later, in the case of wrapper maintenance, tag-separators sometimes need to be dynamically built according to the characteristics of the pages we wish to deal with.

Each DEXTL pattern has a set of associated tag-separators, which are indicated through the SEPARATORS construction. All HTML tags that do not conform to a tag-separator are just ignored by the DEXTL extraction programs.

Example: Fig. 17 shows two search results from an electronic music shop. We wish to extract items having type $ALBUM = \{TITLE, ARTIST, DATE, EDITION\{FORMAT, PRICE\}\}$. Fig. 18 shows the respective fragment of HTML code, where tag attributes have been omitted. Fig. 19 shows a DEXTL program to extract the occurrences of the element $ALBUM$.

In the $EDITION$ extraction pattern the only tag-separator used is $EOL$, while in the $ALBUM$ extraction pattern the $TAB$ ("\<td\>") and $ANCHOR$ ("\<a\>") tag-separators are used.

### 7.2 Inducing DEXTL Patterns

Our induction algorithm will generate a set of DEXTL patterns for each non-atomic field of the type of the data records we wish to extract. Then, it will compose a DEXTL program nesting the generated patterns according to the type structure. It will also find any $UNTIL$ pattern needed.

Our basic algorithm for induction is $obtainAtomicPattern$, which is shown in Fig 20. This algorithm receives as inputs a type $T$ which is assumed to be exclusively composed of atomic fields, a set of labeled examples for data records of type $T$ and a page $P$ containing those examples (or, as usual, a set of pages considered as a unique page where their DOM trees are siblings). The algorithm outputs a set of DEXTL patterns for extracting occurrences of the desired elements of type $T$ from the target pages.

The $obtainAtomicPattern$ algorithm iterates over the examples to generate a candidate pattern based on each example. The process stops when the set of current patterns recognizes all the
examples. Then, the set of results matched by each pattern is explored to remove redundant patterns (those which only match examples also extracted by other patterns). The final step of the algorithm consists in lengthening the pattern to make it as less ambiguous as possible while still recognizing the same examples. This is made by adding tokens from \( P \) at the end and the beginning of the candidate pattern.

The following sub-sections describe in detail certain aspects of the algorithm: how candidate patterns are generated (7.2.1), how new tag separators are induced when needed (7.2.2) and how the algorithm is generalized to accept types of arbitrary depth.

### 7.2.1 Generating candidate patterns

The process for generating the candidate pattern for a labeled example \( e_1 \) is illustrated with an example in Fig. 21. The basic idea is as follows: first, we obtain an HTML portion containing the example and generate a preliminary DEXTL candidate pattern using the current set of tag-separators (we begin by using the former set of tag-separators used by the wrapper), and using the special placeholder \textit{IRRELEVANT} to represent all the visible texts present in the HTML portion (Fig. 21 step 1).

Then, we execute the pattern on \( P \). The set of matches is probable to contain occurrences of other labeled examples (the examples having the same “visual layout” than \( e_1 \)). For instance in the Fig. 21, the preliminary pattern generated for the first example also matches with the third one.

Therefore, we identify these matches and use them to refine the pattern in the following way:

- We consider the text tokens that correspond with the field occurrences of the other input examples enclosed. We substitute the field occurrence by a placeholder with its name, and we find the longest common prefixes and suffixes of the strings surrounding the field
occurrence. The common prefixes and suffixes will be considered as “fixed” in the refined regular expression for defining the text token (see Fig. 21 step 2 for an example).

- The process of looking for common fixed prefixes and suffixes is also applied to the text portions which do not contain any field occurrence (using \textit{IRRELEVANT} as placeholder for the variable parts), but only if the number of other examples matched by the pattern exceeds a certain parameter $k$ (currently set to 5). This is a compromise to avoid generating “fixed” text sections that are too specific.

7.3 Inducing New Tag-Separators

The process gets more complicated if the current set of tag-separators is no longer valid when the source changes. This situation is detected in the step 3 of the algorithm for generating the candidate patterns. The system can correct this situation by applying the following rules:

1) The DEXTL interpreter assumes that the tag-separators cannot appear inside of the strings which match with the text tokens of the pattern (in other case, they would not actually be \textit{separators}). Therefore, if there are any occurrences of a tag-separator \textit{inside} an occurrence of a field of an example, then that separator must be removed from the list of tag-separators. For instance, in the Fig. 22 (step 1) the tag separator \textit{EOL} is not longer valid because it splits in two parts the commentary of the example news stories.

2) Any pair of text tokens (field names or \textit{IRRELEVANT}) in a DEXTL pattern must be separated either by a text (specified as a prefix or suffix in the regular expression associated to the text token for the field) or by a tag-separator. If this does not happen for a preliminary DEXTL pattern, the system will add a new tag-separator by analyzing the HTML tags between both fields in all examples matching with the pattern, and choosing a common string of tags from either the beginning or the end (ignoring attributes), and
ensuring that the chosen separator does not violate the former rule (it cannot appear inside of a field occurrence). For instance in Fig. 22 (step 2), after removing EOL as a valid tag-separator, the title and commentary of the news stories are not separated by neither a text nor a current tag-separator. Therefore, we need to generate a new tag-separator. “</A>” is a suitable election because it is a string of tags (of length 1) which appears between the title and commentary of the news stories but does not violate rule 1.

3) In the same way, the complete examples also need to be separated either by a text or by a tag-separator. If this does not happen for a preliminary DEXTL pattern, the system will add a new tag-separator in a similar way as in the previous case.

7.3.1 Generalization to types of arbitrary depth

The basic idea for the generalization of the obtainAtomicPattern algorithm to types of arbitrary depth is reminiscent of the one used to generalize the algorithms for labeling examples in section 6. We recursively build candidate patterns for all the non-atomic sub-elements of the type and assemble them hierarchically to make up a DEXTL program. An additional difficulty for the generalized algorithm is computing the UNTIL clauses that the DEXTL interpreter needs in order to detect the end of the region where the occurrences for a certain sub-element are found. This is overcome by searching a DEXTL pattern with zero matches inside the examples of the sub-element and which appears always after the last example of the sub-element.

8 EXPERIMENTS

To evaluate the effectiveness of our approach we monitored a set of Web sites during six months. We selected sources from different domains presenting different characteristics and data types. For instance, we included certain sources with some highly variable data fields (e.g.
Nasdaq) to test our techniques in that scenario. We also included some sources where the contents change very frequently (e.g. New York Times).

For every Web site, we first generated a wrapper and used it on a daily basis to execute different queries. The system automatically stored some of the queries and their results according to the techniques explained in section 3. When a change in the format of the pages of a Web site was detected, the system generated a new set of labeled examples and used them as input to induce a new wrapper for the site (in these experiments we used our own wrapper induction algorithm for this stage). Once the wrapper had been regenerated, we tested it with a new set of pages from the source obtained through different queries.

We will quantify the results at two different points of the regeneration process: after generating the new set of labeled examples and after the reinduction stage. At the first point, we can measure the effectiveness of our techniques for generating new correct labeled examples. At the second point, the wrapper is completely regenerated and we can measure the fitness of the generated examples to reinduce the wrapper using our induction algorithm.

To quantify the results of the stage of generating a new set of labeled examples we define the following metrics:

- LN: number of input results (from the stored queries).
- LPE: number of input results that are still present in the current HTML pages of the source after the change.
- LTE: number of total examples labeled by the system.
- LCE: number of examples correctly labeled by the system.
- Labeling Recall (LR) = LCE/LPE. It represents the ratio between the correctly labeled examples and all the examples that should be labeled on the pages.
- Labeling Precision (LP) = LCE/LTE. It represents the ratio between the correctly labeled examples and the total number of labeled examples.

To quantify the performance once the wrappers have been regenerated, we define the following metrics:
- N: number of data records that should be extracted from the test pages.
- TE: number of total data records extracted by the regenerated wrapper from the test pages.
- CE: number of correct data records extracted by the regenerated wrapper from the test pages.
- Recall (R) = CE / N. It represents the ratio between the correctly extracted data records and all the data records that should be extracted from the test pages.
- Precision (P) = CE / TE. It represents the ratio between the correctly extracted data records and all the data records that have been actually extracted from the test pages.

8.1 Analysis of Results

Table 1 lists the subset of the monitored sources where any changes occurred during the period of study. Each of the sources underwent just one format change except for Barnes & Noble, Amazon Magazine, Espacenet and Yahoo People, where two changes occurred (in these cases we computed the averages of the obtained results). The last column indicates if the sibling nodes heuristic was applicable in the source after the change.

Some of the changes were relatively small. For instance, in Yahoo People a new column was added to the table that renders the people’s information. On the other hand, some sources changed their underlying templates completely (for instance, Barnes & Noble).

Table 2 shows the values of the metrics computed for the labeling process. A first conclusion is that the ratio of preserved query results (LPE%) is high for almost all the monitored sources.
This supports the idea of using results from previous queries as a good basis on which to generate a new set of examples for re-induction.

The only source where the ratio of preserved examples is very low is *New York Times*, in which case only 19% of the results were preserved. Nevertheless, the system showed a high effectiveness even in this case.

In the sources with fields considered as highly variable (*NASDAQ*, *Yahoo Quotes*, *AllBooks4Less*), we computed the percentage of preserved results without considering those fields. As can be seen, the ratio is very high, which indicates that the values of the remaining fields in these sources are very stable.

The LR and LP metrics give a 100% value for all the sources except for *AllBooks4Less*, where the system did not discard two obsolete query results. The errors were caused because the sibling nodes heuristic was inapplicable and there were some rare cases where the proximity heuristic failed.

Table 3 shows the results calculated after complete wrapper regeneration. The second and third columns show the values computed for the Recall and Precision metrics. The fourth column indicates the number of data records that should be extracted from the pages used to test the wrapper. The fifth column shows the number of input examples to the re-induction algorithm (i.e. the examples labeled in the previous stage). Remember that all examples were correctly labeled in all sources except in *AllBooks4Less* where 2 of the 64 examples were erroneous. The sixth column shows the number of extraction rules created by our induction algorithm for each wrapper. An extraction rule can be roughly matched to a different “visual representation” of the data records in the source.
The basic factor influencing the results at this stage is that the induction process benefits from having at least a certain number of examples for each possible visual representation, so the appropriate extraction rules can be induced.

If the examples do not cover the entire range of visual representations, then, even when 100% of the examples are correctly labeled at the previous stage, the system will still not reach 100% for the recall and precision metrics. Consequently, the worst results were obtained in Amazon Book, which is, by far, the source with the largest variety of visual representations, and in New York Times, which is the source with more data variability (only 19% of the stored results are preserved).

9 RELATED WORK

Semi-automatic wrapper generation has been an active research field for years [4,5,7,9,10,12,13,19,20] (see [14] for a survey). These works are only concerned with wrapper generation and do not support automatic wrapper maintenance.

[11,16] address the wrapper verification problem and, thus, they are complementary to our work. [16,17,18] have addressed the problem of wrapper maintenance. [16] uses information collected during previous wrapper operation to perform an automatic process for labeling a set of new examples, thus producing the input for their induction algorithm. The process for identifying the correct candidate field occurrences does not take into account neither the context provided by the candidate occurrences of the other fields forming the element nor by the other results of the same type. In addition, they do not deal with the maintenance of wrappers for pages containing “lists” of data elements (they only consider sources where each page contains information of only one data item). In our experience, the need for extracting information from list pages is very common.
[17] identifies candidate field occurrences by assuming some structural features to remain invariant when sources change. These features are annotations (text strings that identify a certain data field such as “Title” or “Price”), hyperlink features and syntactic features (it assumes that the values will conform to a given regular expression). Then they use the identified candidate field occurrences and the schema of the target data elements to identify a set of blocks in the page that conforms to the structure and content suggested by the data schema. In sources where the assumption of invariance of annotations and hyperlinks does not hold, the system will fail to identify correct examples.

[18] proposes a wrapper reinduction approach, which iteratively uses the extracted data at a given time $t$ to regenerate the wrapper at a later moment $t+s$, where $s$ is small. Their approach can only induce LR wrappers [12]. LR (Left-Right) wrappers receive as input a left delimiter and a right delimiter for each field of the data records to extract. Then, the wrapper works by scanning the target page searching sequentially for the left delimiter of the field, and extracting the data up to the right delimiter. LR-wrappers are not expressive enough to deal with most modern Web sites. In fact, [12] concludes that only 53% of semi-structured web sources can be dealt with by using LR-wrappers and, this percentage is probably even inferior today since web sources are increasingly complex. In addition, [18] do not deal with the problem of correctly labeling the examples in the target pages after a source change.

Another difference with respect to [16,17,18] is that our system takes into account the context provided by other results in order to choose the best candidate occurrences for a stored result. This improves the accuracy of the labeling process and makes the approach less vulnerable, when the target pages have several portions that conform to the underlying schema.
[24] proposes an ontology-based schema for wrapper generation. In this schema, an expert creates an ontology describing a particular domain of interest, and the wrappers for sources compliant with the domain description can be automatically generated by the system. In this approach, the wrappers are resilient to changes, as long as the ontology continues describing the source after the change. Defining such ontologies for non-trivial domains is a very challenging task, since the differences between sources in the same domain can be vast.

The works presented in [2,6,8,15,21,23] deal with automatic wrapper generation and are also related to our system to some extent. [2,6,8] take as input a set of example pages of the same class and automatically induce the underlying template they conform to. [23] uses an approach based on tree-edit distance techniques to achieve a similar objective, but they do not need several pages; they only need a single page containing several structured records following a similar layout. [15] benefits from the usual way of structuring query answers into “result listing” pages and “data record detail” pages: they analyze the common content between both kind of pages to automatically find data records.

Automatic wrapper generation approaches have the advantage of not requiring input examples for wrapper generation (at the cost of having less information on which to base the building of accurate wrappers), but they do not annotate the extracted results. While human post-annotation of the data is acceptable in wrapper generation, it is not so for automatic wrapper maintenance. In [3] an approach is presented for automatic annotation of the extracted data using the texts surrounding the extracted items on the page. Nevertheless, it cannot deal with the common situations, where some data fields have not associated labels on the page.
10 CONCLUSIONS

In this paper, we have presented techniques for the generation and automatic maintenance of wrappers for semi-structured Web sources. Our approach leverages on using inductive learning algorithms for generating wrappers. For automatic maintenance, our system collects some query results during wrapper operation and, when the source changes, they are used to generate a new set of labeled examples which constitute the input to induce the new wrapper.

The main step involved in this process is correctly labeling on the new pages the results collected during previous wrapper operation. Our approach is based on three heuristics (“sibling nodes”, “pattern” and “proximity”), which arise from observations of the typical ways to arrange semi-structured information on a page or set of pages. We have experimentally tested our techniques for a variety of real-world Web sources, obtaining a high degree of effectiveness.

ACKNOWLEDGMENTS

This research was partially supported by the Spanish Ministry of Education and Science under project TSI2005-07730. Alberto Pan’s work was partially supported by the "Ramón y Cajal" programme of the Spanish Ministry of Education and Science.
REFERENCES


<table>
<thead>
<tr>
<th>Source</th>
<th>Data fields</th>
<th>Sibling nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllBooks4Less (Book)</td>
<td>title, author, format, price, quantity</td>
<td>No</td>
</tr>
<tr>
<td>Amazon (Book)</td>
<td>title, author, format, price, availability</td>
<td>Yes</td>
</tr>
<tr>
<td>Amazon (Magazine)</td>
<td>title, price, unit price, issues, months</td>
<td>Yes</td>
</tr>
<tr>
<td>Barnes&amp;Noble (Book)</td>
<td>title, author, format, price, availability</td>
<td>Yes</td>
</tr>
<tr>
<td>BigBook (Person)</td>
<td>name, street address, city, state, phone</td>
<td>Yes</td>
</tr>
<tr>
<td>Cordis (Project)</td>
<td>title, date, relevance, status, program acronym, project acronym, reference, start date, end date, abstract</td>
<td>Yes</td>
</tr>
<tr>
<td>Delphion (Patent)</td>
<td>title, publication, date, store</td>
<td>Yes</td>
</tr>
<tr>
<td>Discoweb (Album)</td>
<td>title, artist, date, edition: {format, price}</td>
<td>Yes</td>
</tr>
<tr>
<td>Espacenet (Patent)</td>
<td>title, publication, date, inventor, applicant, IPC</td>
<td>No</td>
</tr>
<tr>
<td>Nasdaq (Flash Quote)</td>
<td>symbol, last sale, share volume, market</td>
<td>Yes</td>
</tr>
<tr>
<td>NewYorkTimes (News Story)</td>
<td>title, author, summary</td>
<td>Yes</td>
</tr>
<tr>
<td>YahooPeople (Person)</td>
<td>name, street address, city, state, phone</td>
<td>Yes</td>
</tr>
<tr>
<td>YahooQuotes (Quote)</td>
<td>symbol, time, trade, change, % change, volume</td>
<td>Yes</td>
</tr>
<tr>
<td>Source</td>
<td>LN</td>
<td>LPE (%)</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----</td>
<td>---------</td>
</tr>
<tr>
<td>AllBooks4Less (Book)</td>
<td>100</td>
<td>62 (1)</td>
</tr>
<tr>
<td>Amazon (Book)</td>
<td>200</td>
<td>73.5</td>
</tr>
<tr>
<td>Amazon (Magazine)</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>Barnes&amp;Noble (Book)</td>
<td>100</td>
<td>65</td>
</tr>
<tr>
<td>BigBook (Person)</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Cordis (Project)</td>
<td>100</td>
<td>99</td>
</tr>
<tr>
<td>Delphion (Patent)</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Discoweb (Album)</td>
<td>100</td>
<td>87</td>
</tr>
<tr>
<td>Espacenet (Patent)</td>
<td>200</td>
<td>98</td>
</tr>
<tr>
<td>Nasdaq (Flash Quote)</td>
<td>100</td>
<td>100 (1)</td>
</tr>
<tr>
<td>NewYorkTimes (News Story)</td>
<td>110</td>
<td>19</td>
</tr>
<tr>
<td>YahooPeople (Person)</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>YahooQuotes (Quote)</td>
<td>100</td>
<td>100 (1)</td>
</tr>
</tbody>
</table>

(1) Without considering highly variable data fields
TABLE 3
WRAPPER REGENERATION METRICS

<table>
<thead>
<tr>
<th>Source</th>
<th>R (%)</th>
<th>P (%)</th>
<th>N</th>
<th>LTE</th>
<th>Extraction rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllBooks4Less (Book)</td>
<td>100</td>
<td>97.9</td>
<td>1000</td>
<td>64</td>
<td>2</td>
</tr>
<tr>
<td>Amazon (Book)</td>
<td>86.8</td>
<td>93.4</td>
<td>1000</td>
<td>147</td>
<td>27</td>
</tr>
<tr>
<td>Amazon (Magazine)</td>
<td>99.8</td>
<td>100</td>
<td>1000</td>
<td>50</td>
<td>7</td>
</tr>
<tr>
<td>Barnes&amp;Noble (Book)</td>
<td>98.2</td>
<td>100</td>
<td>1000</td>
<td>65</td>
<td>8</td>
</tr>
<tr>
<td>BigBook (Person)</td>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>Cordis (Project)</td>
<td>97.8</td>
<td>100</td>
<td>1000</td>
<td>99</td>
<td>5</td>
</tr>
<tr>
<td>Delphion (Patent)</td>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>Discoweb (Album)</td>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>87</td>
<td>2 / 1 (1)</td>
</tr>
<tr>
<td>Espacenet (Patent)</td>
<td>99.5</td>
<td>100</td>
<td>1000</td>
<td>196</td>
<td>12</td>
</tr>
<tr>
<td>Nasdaq (Flash Quotes)</td>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>NewYorkTimes (News Story)</td>
<td>94.5</td>
<td>100</td>
<td>200</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>YahooPeople (Person)</td>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>YahooQuotes (Quote)</td>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>100</td>
<td>2</td>
</tr>
</tbody>
</table>

(1) Album type / Edition subtype
Application

Wrapper automatically fills a form and submits the query

The application executes a query
\[ q = \text{TITLE contains XML} \]

Wrapper extracts structured query results and returns them to application

Extracted Query Results

Title: XML in a Nutshell, Third Edition
Author: Elliotte Rusty Harold, W. Scott Means
Format: Paperback
Date: September 2004
Price: $26.37

Title: Beginning XML, Second Edition
Author: David Hunter, et al
Format: Paperback
Date: March 2003
Price: $27.19

Author: [Blank]
Title: XML

Wrapper automatically fills a form and submits the query

Fig. 1. Wrapper execution
User labels data records in example pages

A set of extraction rules are induced from the examples

The induced wrapper can be used to extract data from pages following the same template

Fig. 2. Wrapper induction
1) During normal operation wrapper stores some results in the local database

2) When the source changes, system labels stored results in new pages

3) A new wrapper is induced

Fig. 3. Wrapper automatic maintenance
Fig. 4. 'ALBUM' type
## Fig. 5a. Candidate occurrences

### Most popular results for XML:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Title</th>
<th>Author(s)</th>
<th>Publisher</th>
<th>Edition</th>
<th>Date</th>
<th>List Price</th>
<th>Used Price</th>
<th>Customer Rating</th>
<th>URL</th>
</tr>
</thead>
</table>

### All results for XML:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Title</th>
<th>Author(s)</th>
<th>Publisher</th>
<th>Edition</th>
<th>Date</th>
<th>List Price</th>
<th>Used Price</th>
<th>Customer Rating</th>
<th>URL</th>
</tr>
</thead>
</table>
Fig. 5b. Candidate occurrences
### Table 1: XML Books

<table>
<thead>
<tr>
<th>Title</th>
<th>Author(s)</th>
<th>Format</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML in a Nutshell, Third Edition</td>
<td>Elliotte Rusty Harold, W. Scott Means</td>
<td>Paperback</td>
<td>$30.05</td>
</tr>
<tr>
<td>Beginning XML, Second Edition</td>
<td>David Hunter et al</td>
<td>Paperback</td>
<td>$30.00</td>
</tr>
</tbody>
</table>

### Figure 6: Incorrect Candidate Occurrences

![Incorrect Candidate Occurrences](image-url)
Algorithm: Obtain best atomic candidate occurrences for a query result ri.

\[ E_i = \text{obtainBestAtomicCandidateOccurrences}(P, T, ri, \text{maxCandidates}) \]

Input:
- \( P \), a document (or a set of documents considered as if they were part of a single page where their DOM trees were siblings).
- A type \( T \ <\text{field}_1, \ldots, \text{field}_f> \) whose fields are all atomic.
- \( R = \{r_1, \ldots, r_n\} \), where \( r_j \) is a query result of type \( T \) contained in \( P \).
- \( r_j \in \{1..n\} \) is of the form \( \{(\text{field}_1, \text{value}_1), \ldots, (\text{field}_f, \text{value}_f)\} \), where \( \text{value}_k \in \{1..f\} \) denotes the value of \( r_j \) for the \( k \)-th field of \( T \).
- \( \text{maxCandidates} \) is the maximum number of candidate occurrences to be considered in the algorithm.

Output:
- \( E_i \), a set with the best candidate occurrences for \( r_i \) or the empty set if the result is considered to be “obsolete”.

1) Find the occurrences of each value \( \text{value}_k \in \{1..f\} \) in \( P \). This will output a set \( \{O_k\}_{k=1..f} \), where \( O_k \) is the set of occurrences of \( \text{value}_k \) in \( P \).
2) Compute the number of all candidate occurrences of \( r_i \) in \( P \) as \( \text{numCandidateOccurrences} = \prod_{k=1..f} |O_k| \).
3) If \( \text{numCandidateOccurrences} > \text{maxCandidates} \)
   a) Choose the best candidate occurrences guaranteeing \( \text{numCandidateExampleOccurrences} <= \text{maxCandidates} \) (see section 5.5).
4) If the “sibling nodes” heuristic is applicable, sort the candidate occurrences by using the “sibling nodes” heuristic, else sort candidate occurrences by using the “pattern” heuristic.
5) If the “sibling nodes” heuristic is applicable, and all the candidate occurrences have value zero for this heuristic, then return the empty set (the result is considered obsolete and discarded).
6) Sort the occurrences with the same value for the previous (“sibling nodes” or “pattern”) heuristic by using the “proximity” heuristic.
7) If the best candidate occurrence of \( r_i \) overlaps on the page with the best candidate occurrence of another result \( r_j \), then discard the result with lower ranking according to the proximity heuristic.
8) Return \( E_i \), a set with all the occurrences ranking highest.

Fig. 7. Base algorithm
Fig. 8. Sibling nodes heuristic
Fig. 9. Sibling nodes heuristic example
Minimum sub-tree

Fig. 10a. Incorrect candidate occurrence
<table>
<thead>
<tr>
<th>TITLE</th>
<th>FORMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head First Java</td>
<td>Paperback</td>
</tr>
<tr>
<td>Hardcore Java</td>
<td>Paperback</td>
</tr>
</tbody>
</table>

Fig. 10b. Incorrect candidate occurrence
<table>
<thead>
<tr>
<th>Title</th>
<th>Author</th>
<th>Format</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head First Java</td>
<td>Bert Bates</td>
<td>Paperback</td>
<td>27.96</td>
</tr>
<tr>
<td>Hardcore Java</td>
<td>Robert Simmons</td>
<td>Paperback</td>
<td>27.17</td>
</tr>
</tbody>
</table>

Fig. 11. Minimum sub-tree too general
<table>
<tr><td><a>Beginning XML, Second Edition</a></td><td>by David Hunter et al</td><td>Paperback - March 2003</td><td>Price: $27.19</td></tr>
<tr><td><a>Real World XML, Second Edition</a></td><td>by Steven Holzner</td><td>Hardback - September 2004</td><td>Price: $27.19</td></tr>
<tr><td><a>XML Weekend Crash Course</a></td><td>by Kay Ethier et al</td><td>Paperback - September 2001</td><td>Used and new from: $10.99</td></tr>
</table>

Fig. 12. Pattern heuristic
Fig. 13. Detecting obsolete query results
Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Trade</th>
<th>% Chg</th>
<th>Intraday</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORCL</td>
<td>11.24</td>
<td>+1.81%</td>
<td></td>
</tr>
<tr>
<td>IBM</td>
<td>88.25</td>
<td>+0.79%</td>
<td></td>
</tr>
<tr>
<td>AACC</td>
<td>19.57</td>
<td>+0.36%</td>
<td></td>
</tr>
<tr>
<td>AACE</td>
<td>25.15</td>
<td>+1.04%</td>
<td></td>
</tr>
<tr>
<td>AAC</td>
<td>0.63</td>
<td>-3.08%</td>
<td></td>
</tr>
</tbody>
</table>

Fig 14. Dealing with highly time-variable data
Fig. 15a. General algorithm operation
### Table 1

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SPIRIT IN THE DARK</td>
<td>ARETHA FRANKLIN</td>
<td>12/1993</td>
<td>$EDITION1</td>
<td>$EDITION2</td>
<td>$EDITION3</td>
</tr>
<tr>
<td>ARETHA NOW</td>
<td>ARETHA FRANKLIN</td>
<td>07/1993</td>
<td>$EDITION1</td>
<td>$EDITION4</td>
<td></td>
</tr>
</tbody>
</table>

---

**Fig. 15b. General algorithm operation**
**Algorithm:** Obtain best candidate occurrences for a set of results R.

- **R =** $\text{obtainBestCandidateOccurrences (P,T,R,maxCandidates)}$

**Input:**
- $P$, a document (or a set of documents considered as if they were part of a single page where their DOM trees were siblings).
- $R = \{r_1, \ldots, r_n\}$, where $r_j \in T$ is query result of type $T$.
- $\text{maxCandidates}$ is the maximum number of candidate occurrences to be considered in the algorithm at each level of the tree of $T$.

**Output:**
- $bco = \{\text{the set containing the best candidate occurrences for the query results } r_1, \ldots, r_n\}$

1) Build $S$ the tree of $T$.
2) $k = \text{depth}(S)$.
3) While ($k > 0$)
   a) $k = k - 1$
   b) $bco = \emptyset$ (the empty set)
   c) For every sub-result of $k$-th level $se$ of every result $r_j \in R$
      i) $SE = \{\text{all sub-results of the same type of } se \text{ in any result contained in } R\}$
      ii) Obtain $bco_{se} = \text{obtainBestAtomicCandidateOccurrence (P, SE, se, maxCandidates)}$, the set of the best candidate occurrences for $se$.
      iii) $bco = bco \cup bco_{se}$
   d) if ($k > 0$)
      i) Substitute in $P$ each candidate occurrence $c \in bco$ by a unique markup generated based on the data values that form it. If two candidate occurrences $c_1$ and $c_2$ are formed by the same data values, then the markup generated for them will be equal.
      ii) Substitute in every $r_j \in R$ every non-atomic field of level $k$ by as many atomic fields as sub-results in the non-atomic field, and assign as value to each added atomic field the markup generated for the corresponding sub-result.
4) Return $bco$.

Fig. 16. General algorithm
<table>
<thead>
<tr>
<th>Title</th>
<th>Artist</th>
<th>Date</th>
<th>Format/Price</th>
</tr>
</thead>
</table>
| Don't Turn Me From Your Door | John Lee Hooker | 2/1992 | CD/SEP.07  
|                     |               |       | W/IA/83     |
| In Through The Out Door    | Led Zeppelin  | 08/1984 | CD/EP.05  
|                     |               |       | LP/EP.29.9  
|                     |               |       | W/EP.11.05  |

**Fig. 17. Two search results**
<table>
<thead>
<tr>
<th>Album title</th>
<th>Artist</th>
<th>Date</th>
<th>Format/price</th>
</tr>
</thead>
<tbody>
<tr>
<td>DON'T TURN ME FROM YOUR DOOR</td>
<td>JOHN LEE HOOKER</td>
<td>2/1992</td>
<td>CD / £9.07, MC / £7.93</td>
</tr>
<tr>
<td>IN THROUGH THE OUT DOOR</td>
<td>LED ZEPPELIN</td>
<td>8/1994</td>
<td>CD / £11.85, LP / £29.28, MC / £11.05</td>
</tr>
</tbody>
</table>

Fig 18. HTML Code
ALBUMS {
  EXTRACTION {
    ALBUM {
      EXTRACTION {
        PATTERN {
          SEPARATORS (ANCHOR, TAB)
          ANCHOR \[($TITLE)\] TAB ANCHOR \[($ARTIST)\] TAB \[($DATE)\]
        }
      }
    }
  }
  EDITION {
    EXTRACTION {
      PATTERN {
        SEPARATORS (EOL)
        \[$(FORMAT) \"/\" $(PRICE)\]
      }
    }
  }
  UNTIL {
    PATTERN {
      SEPARATORS (EOL)
      EOL EOL
    }
  }
  }
  }
  }
}

Fig. 19. DEXTL Program
**Algorithm:** Generate a set of DEXTL patterns to extract occurrences of $T$.

- $EP = obtainAtomicPattern(P, T, S, E)$

**Inputs:**
- $P$, a document (or a set of documents considered as if they were part of a single page where their DOM trees were siblings).
- A type $T<field_1, …, field_r>$ whose fields are all atomic.
- $S$, the current set of tag type separators.
- $E = \{e_1, …, e_n\}$, where $e_i \in \{1..n\}$ is a labeled example of type $T$ contained in $P$.
- $e_i \in \{1..n\}$ is of the form $\{(v_{i1}, pos_{i1}), …, ((v_{if}, pos_{if}))\}$ where $(v_k, pos_k) k \in \{1..f\}$ denotes the $pos_k$-th occurrence in $P$ of the string $v_k$, which is the value of $e_i$ for the $k$-th field of $T$.

**Outputs:**
- $EP = \{ep_1, …, ep_t\}$ the set of generated extraction patterns.

1) $EP = \emptyset$; $UnmatchedExamples = E$;
2) While ($UnmatchedExamples \neq \emptyset$)
   a) Choose any $e_j \in UnmatchedExamples$.
   b) Generate a candidate pattern $cp_i$ from $e_i$ (see Fig 22).
      $cp_i = generateCandidatePattern(P, T, S, E, e_i)$
   c) Compute the number $m_i$ of matchings for $cp_i$ in $P$, and the number $me_i$ of matchings which correspond with some $e_j \in E$.
   d) Let $MatchedExamples$ be the set of examples matched by $cp_i$.
      $UnmatchedExamples = UnmatchedExamples – MatchedExamples$.
3) End While
4) Now $EP$ contains a set of extraction patterns $\{ep_1, …, ep_s\}$. Remove from $EP$ the patterns such as all the examples recognized by them are recognized by other patterns.
5) Disambiguate the patterns of $EP$. For each pattern add tokens at the end and the beginning while it recognizes the same set of samples ($me_i$). Get as result the smaller patterns with the fewer $(m_i-m)$. 
6) $EP$ contains the extraction patterns for $T$.

Fig. 20. Basic algorithm for induction
Fig. 21. Generating candidate patterns
A new method of diagnosing heart disease takes only seconds to conduct, but critics say the technique is ripe for overuse. Finding the wreck of a 19th-century steamer in Florida has taken special technology and the results have been extraordinary.

Fig. 22. Inducing new tag-separators