Wrapper Generation via Grammar Induction

Boris Chidlovskii[1], Jon Ragetli[2⋆], and Maarten de Rijke[2⋆⋆]

[1] Xerox Research Centre Europe
6, Chemin de Maupertuis, 38240 Meylan, France
chidlovskii@xrce.xerox.com
[2] ILLC, University of Amsterdam
Pl. Muidergracht 24, 1018 TV Amsterdam, The Netherlands
{ragetli,mdr}@wins.uva.nl

Abstract. To facilitate effective search on the World Wide Web, meta search engines have been developed which do not search the Web themselves, but use available search engines to find the required information. By means of wrappers, meta search engines retrieve information from the pages returned by search engines. We present an approach to automatically create such wrappers by means of an incremental grammar induction algorithm. The algorithm uses an adaptation of the string edit distance. Our method performs well; it is quick, can be used for several types of result pages and requires a minimal amount of user interaction.

Keywords: inductive learning, information retrieval and learning, web navigation and mining, grammatical inference, wrapper generation, meta search engines.

1 Introduction

As the amount of information available on the World Wide Web continues to grow, conventional search engines expose limitations when assisting users in searching information. To overcome these limitations, mediators and meta search engines (MSEs) have been developed [2,6,7]. Instead of searching the Web themselves, MSEs exploit existing search engines to retrieve information. This relieves the user from having to contact those search engines manually. Furthermore, the user formulates queries using the query language of the MSE — knowing the native query languages of the connected search engines is not necessary. The MSE combines the results of the connected search engines and presents them in a uniform way.

MSEs are connected to search engines by means of so-called *wrappers*: programs that take care of the source-specific aspects of an MSE. For every search engine connected to the MSE, there is a wrapper which translates a user's query into the native query language and format of the search engine. The wrapper also takes care of extracting the relevant information from the results returned

⋆ Supported by the Logic and Language Links project funded by Elsevier Science.
⋆⋆ Supported by the Spinoza project 'Logic in Action.'

R. López de Mántaras, E. Plaza (Eds.): ECML 2000, LNAI 1810, pp. 96–108, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Search results for query: wrapper

Number One Wrapper Generator

Description: Welcome to the wrapper generating organisation.

1000; http://www.wrapper.org/

Buy our candy bar wrapper collection

Description: An advantageous offer for every candy addict.

774; http://www.candy.com/wrappers/

Maestro's Candy Bar Wrapper Collection

Description: Yes, I devote my otherwise useless life to collecting wrappers.

312; http://www.freehomepages.com/~maestro/

Fig. 1. Sample result page

by the search engine. We will refer to the latter as 'wrapper' and do not discuss the query translation (see [5] for a good overview). An HTML result page from a search engine contains zero or more 'answer items', where an answer item is a group of coherent information making up one answer to the query. A wrapper returns each answer item as a tuple consisting of attribute/value pairs. For example, from the result page in Fig. 1 three tuples can be extracted, the first of which is displayed in Fig. 2. A wrapper discards irrelevant information such as layout instructions and advertisements; it extracts information relevant to the user query from the textual content and attributes of certain tags (e.g., the `href` attribute of the `<A>` tag).

Manually programming wrappers is a cumbersome and tedious task [4], and since the presentation of the search results of search engines changes often, it has to be done frequently. To address this, there have been various attempts to automate this task [3,9,10,12,13]. Our approach is based on a simple incremental grammar induction algorithm. As input, our algorithm requires one result page of a search engine, in which the first answer item is labeled: the start and end of the answer need to be indicated, as well as the attributes to be extracted. After this, the incremental learning of the *item grammar* starts, and with an adapted version of the *edit distance* measure further answer items on the page are found and updates to the extraction grammar are carried out. Once all items have been found and the grammar has been adapted accordingly, some post-processing takes place, and the algorithm returns a wrapper for the entire

```
⟨      url = "http://www.wrapper.org",~title = "Number One Wrapper
Generator",~description = "Welcome to the wrapper generating
organization",~relevance = "1000" ⟩
```

Fig. 2. An item extracted

page. The key features of our approach are limited user interaction (labeling only one answer item) and good performance: for a lot of search engines it generates working wrappers, and it does so very quickly.

The paper is organized as follows. In the next section we show how to use grammar induction for the construction of wrappers. After that we describe our wrapper learning algorithm. We then present experimental results, comparisons and conclusions. Full details can be found in [14].

2 Using Grammar Induction

We view *labeled* HTML files as strings over the alphabet $\Sigma \cup \mathcal{A}$, where every $\sigma \in \Sigma$ denotes an HTML tag, and every $a_i \in \mathcal{A}$ $(i = 1, 2, \ldots)$ denotes an attribute to be extracted. The symbol a_0 in $\mathcal{A}$ represents the special attribute `void`, that should *not* be extracted; Σ and $\mathcal{A}$ are disjoint. For example, the HTML fragment

```
<title>Wrapper Induction</title>
```

might correspond to the string $t a_1 \bar{t}$, where t and $\bar{t}$ are symbols of Σ which denote tags `<title>` and `</title>`, respectively. The text `Wrapper Induction` has to be extracted as the value of attribute $a_1 \in \mathcal{A}$.[1]

We aim to construct a wrapper that is able to extract all relevant information from a given labeled page and unseen pages from the same source. We solve the problem by decomposing it into two simpler subtasks. The first one is to find an expression that locates the beginning (`Start`) and the end (`End`) of the list of answer items. The second subtask is to induce a grammar `Item` that can extract all the relevant information from every single item on the page. The grammar describing the entire page will then be of the form `Start (Item)`* `End`. The `Start` and `End` expressions can easily be found. The grammar induction takes place when the grammar for the items is generated. Here, the item grammar is learned from a number of samples from $(\Sigma \cup \mathcal{A})^+$, corresponding to the answer items on the page. Besides learning the grammar, our algorithm also *finds* the samples on the HTML page that it uses to learn the grammar.

Preprocessing the HTML Page. All known approaches for automatically generating wrappers require as input one or more labeled HTML pages: all or some of the attributes to be extracted have to be marked by the user or some labeling program. As it is hard to create labeling programs for the heterogeneous set of search engines that an MSE must be connected to, and the labeling is a boring and time-consuming job, we have restricted the labeling for our algorithm to a single answer item only. Figure 3 shows the labeled source for the HTML page in Fig. 1. The labeling consists of an indication of the begin (`^BEGIN^`) and end (`^END^`) of the first answer item, the names of the attributes (e.g. `^URL^`), and the end of the attributes (`^^`). After the item has been labeled, it is *abstracted* by our algorithm to turn it into a string over $\Sigma \cup \mathcal{A}$.

[1] This representation is somewhat simplified. The program can also extract tag attributes, such as the `href` attribute for the `A` tag, or split element contents with conventional string separators. Due to space limitations, we omit details.

```
<HTML><HEAD><TITLE>Search results for query: wrapper</TITLE></HEAD>
<BODY bgcolor = "white" text= "black">
<H3>Search results for query: wrapper</H3>
  <dl>
     ^BEGIN^ <dt> ^URL^ <a href="http://www.wrapper.org/"> ^^
     ^TITLE^ Number One Wrapper Generator ^^ </a><br>
      <dd><i>Description:</i> ^DESCR^ Welcome to the wrapper
      generating organisation. ^^ <br>
      <font size="-3"><I> ^REL^ 1000 ^^ </I>;
      http://www.wrapper.org/</font> ^END^
  </dl>
  <dl>
    <dt><a href="http://www.candy.com/wrappers/">
    Buy our candy bar wrapper collection </a><br>
             ⋮
     <font size="-3"><I>312</I>;
     http://www.freehomepages.com/~maestro/</font>
  </dl>
</BODY></HTML>
```

Fig. 3. Labeled HTML source of result page

The Item Grammar. The item grammar has to be learned from merely *positive* examples; this cannot be done efficiently for regular expressions with the full expressive power of Finite State Automata (FSAs) [15]. We aim to learn a very restricted kind of grammar, which we will first describe as a simple form of FSA, called sFSA, where transitions labeled with an attribute $a_i \in \mathcal{A}$ (except a_0) also produce output: the attribute name and the token consumed. After that we show how those sFSAs correspond with a simple form of regular expression. We start by defining an extremely simple class of FSAs.

Definition 1 (Linear FSA). A sequence of nodes $n_1 \ldots n_m$, where every node n_i $(1 \leq i < m)$ is connected to n_{i+1} by one edge $e_{i,i+1}$ labeled with elements from $\Sigma \cup \mathcal{A}$, is a *linear FSA* if it is the case that whenever $e_{i,i+1}$ is labeled with an element $a \in \mathcal{A}$, then $e_{i-1,i}$ and $e_{i+1,i+2}$ are labeled with elements from Σ.

The fact that the attribute a in Definition 1 is surrounded by HTML tags (from Σ) allows us to extract the attribute. Fig. 4 shows a linear FSA that can only extract the attributes from *one* type of item: an item that has an attribute

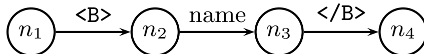

Fig. 4. A linear FSA

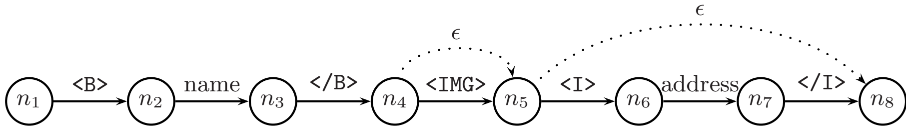

Fig. 5. An sFSA

name between <B> and </B> tags (symbols in Σ, like <B> and </B> in Fig. 4, represent tokens for *abstracted tags*). Therefore, it is not very useful. The sFSAs that we employ to learn the structure of items, are a bit more complex.

Definition 2 (simple FSA). A linear FSA that also has ϵ-transitions $s_{i,j}$ (transitions labeled with ϵ) from node n_i to node n_j $(i < j)$ is called a *simple FSA (sFSA)* if

- whenever there is an ϵ-transition $s_{i,j}$ there is no ϵ-transition $s_{k,l}$ with $i \leq k \leq j$, or $i \leq l \leq j$, and
- whenever there is an ϵ-transition $s_{i,j}$, and $e_{j,j+1}$ is labeled with an element from $\mathcal{A}$, $e_{i-1,i}$ is labeled with an element from Σ.

The first condition demands that ϵ-transitions do not overlap or subsume each other. The second condition states that when an ϵ-transition ends at a node with an outgoing edge with a label from $\mathcal{A}$ (i.e., the abstracted content), it has to start at a node with an incoming edge with a label from Σ (i.e., an abstracted HTML tag). The latter guarantees that an attribute is always surrounded by HTML tags, no matter what path is followed through the automaton.

Figure 5 shows an sFSA that can extract names and addresses from items, where some items do not contain the address between <I> and </I>, and there may be an image (<IMG>) after the name that is enclosed by <B> and </B> tags. The ϵ-transitions of the sFSA make it more expressive than a linear FSA, but sFSAs are less expressive than FSAs, since sFSAs do not contain cyclic patterns.

Where do grammars come in? One can represent the language defined by an sFSA by a simple kind of regular expression with fixed and optional parts. Using brackets to indicate optional parts, the sFSA of Fig. 5 can be represented as <B>name</B>[<IMG>][<I>address</I>]. This expression acts as a grammar defining the same sequences of abstracted tags and content as the sFSA. We refer to this representation as *item grammar* or simply *grammar*. The grammar can be this simple, because the HTML pages for which they are created are created dynamically upon user requests and therefore have a regular structure.

3 Inducing the Item Grammar

Our grammar induction algorithm is incremental; item grammar G_n, based on the first n items, is adapted on encountering item $n+1$, resulting in grammar G_{n+1}. The update of the grammar is based on an algorithm calculating the *string edit distance* [1].

item grammar	a	b	$-$	d
string	a	b	c	d
new item gr.	a	b	$[c]$	d

(a)

item grammar	a	b	$[c]$	d
string	a	b	c	$-$
new item gr.	a	b	$[c]$	$[d]$

(b)

item grammar	a	b	$-$		d
string	a	$-$	c		d
new item gr.	a	$[b]$	$[c]$	$[b]$	d

(c)

Fig. 6. Three alignments

Definition 3 (Edit distance). The *edit distance* $D(s_1, s_2)$ between two strings of symbols s_1 and s_2 is the minimal number of insertions or deletions of symbols, needed to transform s_1 into s_2.

For example, $D(abcd, abide) = 3$: to transform *abcd* into *abide* at least three insertion or deletion operations have to be performed. Here, and in the examples below, the characters are symbols from $\Sigma \cup \mathcal{A}$. The algorithm that we use to calculate the edit distance also returns a so-called *alignment*, indicating the differences between the strings. For *abcd* and *abide* the alignment is the following:

a	b	c	$-$	d	$-$
a	b	$-$	i	d	e

The dashes indicate the insertion and deletion operations; see [1,14] for more details. We have adapted the edit distance algorithm in a way that permits to calculate the distance between an item grammar — a string of symbols with optional parts — and an item. The adaptation amounts to first simplifying the item grammar by removing all brackets, while remembering their position. Now the edit distance between the item and the simplified grammar can be calculated as usual. Using the alignment and the remembered position of the brackets, the new grammar is calculated. We have also adapted the edit distance algorithm to deal with labeled attributes in the grammar, that correspond with unlabeled content in the item; we omit details here.

The algorithm detects and processes different cases in the alignment between G_n and the $n+1$-th item. Since the full algorithm description is extensive and space is limited, we can only indicate how it works with the some examples. As the item grammar in Fig. 6 (a) does not contain c, whereas the string to be covered does, the resulting item grammar has an optional c in it, so that it covers both *abd* and *abcd*. Now suppose the string *abc* has to be covered by the new item grammar (Fig. 6 (b)). The reason for making d optional is that it does not occur in the new string. The new item grammar covers *ab*, *abc*, *abd* and *abcd*, which is a larger generalization than simply 'remembering' the examples. In Fig. 6 (c), the new item grammar $a[b][c][b]d$ is a large generalization; besides *abd* and *acd* it covers *ad*, *abcd*, *acbd* and *abcbd*, i.e., five other strings besides the original item grammar and the example. The reason we decided to have a large generalization is that based on the examples we can at least conclude that b and c are optional, but they may co-occur in any order.

```
1. D_newlocal := 998, D_local := 999, D_best := 1000
2. i_b, i_e := 0
3. local-best-item := ∅, best-item := ∅
WHILE D_local < D_best and not at end of page
   4. D_best := D_local
   5. best-item := local-best-item
   6. i_b := next occurrence begin tag(s)
   WHILE D_newlocal < D_local
      7. D_local := D_newlocal
      8. local-best-item := (i_b, i_e)
      9. i_e := next occurrence end tag(s)
      10. D_newlocal := D(item grammar, (i_b, i_e))
11. IF D_best > Threshold THEN best-item := ∅
12. return best-item and D_best
```

• $D_{newlocal}$ stores the distance of the item grammar to the part of the page between the latest found occurrence of the begin and end tag(s)
• D_{local} stores the distance of the item grammar to `local-best-item`
• D_{best} stores the distance of the item grammar to `best-item`
• i_b, i_e are the indexes of the begin and end of a (potential) item
• `local-best-item` stores the potential item starting at i_b that has the lowest distance to the item grammar of the potential items starting at i_b
• `best-item` stores the potential item that has the smallest distance to the item grammar so far

Fig. 7. The Local Optimum Method

4 Finding Answer Items

So far, we have discussed the learning of the grammar based on the answer items on the HTML page. As only the first answer item on the page has been indicated by its labeling, the other items have to be found. For this, we use the distance calculated by the adapted edit distance algorithm. We have implemented three different strategies for finding the answer items on the page, but as space is limited we will only describe the best and most general one: the *Local Optimum Method* (LOM). The other two are simpler and usually quicker, but even with the LOM a wrapper is quickly generated; see Section 6.

Our methods for finding items are based on an important assumption: *all items on the page have the same begin and end tag(s).* As a consequence we can view the task of finding items on a page as finding substrings on the page below the labeled item that start and end with the same delimiters as the first labeled item. The user can decide for how many tags this assumption holds by setting the parameter *SeparatorLength*. If more begin or end tags are used, it will be easier to find the items on the page; there is less chance of finding for example a sequence of two tags than only one tag. However, setting the parameter too high will result in too simple a grammar without any variation.

The LOM tries to find items on the page that are *local*, i.e., below and close to the item that was found last, and *optimal* in the sense that their distance to the item grammar is low. Figure 7 shows the algorithm. In the first three steps, a number of variables are initialized. As to the outermost *while* loop, once the previous item has been found, or the first labeled item, the LOM looks for the next occurrence of the begin delimiter, and then it looks for the first occurrence of the end delimiter. Material between those delimiters is a potential item; this is checked by calculating its edit distance to the item grammar. Below the last found end delimiter, the LOM looks at the next occurrence of the end delimiter. This is a new potential item to consider, so the distance between the item grammar and this potential item is measured. If this distance is lower than the previous distance, another occurrence of the end delimiter is considered. If not, the previous potential item is stored as the local-best-item, and potential items a bit lower on the page are considered next. The process of considering new end delimiters starts again, resulting in a new local-best-item. Now the two local-best-items are compared. If the second one was better than the first one, LOM will seek the next occurrence of the begin delimiter. If not, the previous local-best-item is returned as the local-optimal item.

In step 11 of the algorithm, a *Threshold* is mentioned. If the distance of the best candidate item exceeds *Threshold*, the algorithm will return $\emptyset$ instead of this item; this prevents the grammar to be adjusted to cover the item, and the process of finding the item stops. *Threshold* is the product of two values: *HighDistance* and *Variation*. *HighDistance* is the maximum distance of any item incorporated so far. Its initial value is set by the user, and it is incremented whenever an item is incorporated whose distance is higher than *HighDistance*; it can be used to compensate for the simplicity of the distance measure. *Variation* is a value that is not adapted during the process of finding the items.

5 The Entire Wrapper Generating Algorithm

We have discussed the two most important components of our wrapper generator: learning the grammar, and finding the items. In Fig. 8 the entire wrapper generating algorithm is described; below we discuss some components.

The first step, `abstract`, abstracts LP, the page labeled by the user, into a sequence of symbols $AP \in (\Sigma \cup \mathcal{A})^+$; see Section 2. The second step initializes the grammar G to the first, labeled item. In the third step, `find-next-item` is the algorithm for finding items, as described in Section 4; in the fourth step `incorporate-item` adjusts the grammar in the way we described in Section 3. In the fifth step, the grammar G is used to make a grammar for the whole page. The user might have labeled the first item smaller than it actually is. By the assumption that all items on the page have the same begin and end tags, the found items (and the resulting grammar) will also be too small. Therefore, the item grammar will be extended if possible. If there is a common suffix of the HTML between the items covered and the HTML before the first item, this suffix is appended to the beginning of the item grammar. If there is a common prefix

```
1. AP := abstract(LP)
2. G := initialize(AP)
REPEAT
   3. I := find-next-item(AP, G)
   4. IF I ≠ ∅ THEN G := incorporate-item(G, I)
UNTIL I = ∅
5. GP := expression-whole-page(G, AP)
6. W := translate-to-wrapper(GP)
7. return W
```

- *LP* is the labeled HTML page
- *AP* is the abstracted page
- *G* is the item grammar
- *I* is an item
- *GP* is a grammar for the entire page
- *W* is the same grammar, translated into a working wrapper

Fig. 8. The wrapper generating algorithm

of the HTML between the items, and the HTML below the last found item, it is appended to the end of the item grammar. Besides this, expressions for `Start` and `End`, as discussed in Section 2, are also generated in this fifth step. This is easy: the expression for `Start` is the smallest fragment of *AP* just before the labeled item that does not occur before in *AP*. `End` is recognized implicitly, by the fact that no items can be recognized anymore.

For skipping the useless HTML in the item list, another grammar is constructed — the *Trash* grammar. It does not contain attributes to be extracted, so the trash grammar will consist of symbols in $\Sigma \cup \{a_0\}$. The indices of the items found have been stored, so this process is a repetition of `incorporate-item`. Once the trash grammar has been constructed, it is appended to the end of the item grammar. Once the item and trash grammars have been generated, our algorithm will detect repetitions, and it will generalize the grammars accordingly.

After all these processing steps, we have an abstract wrapper of the form `Start` (`Item Trash`)$^+$, that is an expression for the beginning of the item list, followed by one or more repetitions of a sequence of the item grammar and the trash grammar. The last step of the algorithm in Fig. 8 is the conversion of the abstract grammar into a working wrapper. In our implementation we translate the abstract grammar into a JavaCC parser [11], as the meta searcher Knowledge Brokers, developed at Xerox Research Centre Europe, is programmed in Java.

6 Experimental Results

We have tested our wrapper generating algorithm on 22 different search engines. This is a random selection of sources to which Knowledge Brokers had already been connected manually. It was quite successful, as it created working wrappers

Table 1. Experimental results

Successfully generated wrappers				
source	URL	size (kB)	NI	time (sec)
ACM	www.acm.org/search	12	10	8.0
Elsevier Science	www.elsevier.nl/homepage/search.htt	11	11	2.6
NCSTRL	www.ncstrl.org	9	8	32.5
IBM Patent Search	www.patents.ibm.com/boolquery.html	19	50	5.3
IEEE	computer.org/search.htm	26	20	3.7
COS U.S. Patents	patents.cos.com	17	25	5.4
Springer Science Online	www.springer-ny.com/search.html	36	100	32.1
British Library Online	www.bl.uk	5	10	2.6
LeMonde Diplomatique	www.monde-diplomatique.fr/md/index.html	6	4	2.5
IMF	www.imf.org/external/search/search.html	10	50	5.3
Calliope	sSs.imag.fr*	22	71	4.1
UseNix Association	www.usenix.org/Excite/AT-usenixquery.html	16	20	4.3
Microsoft	www.microsoft.com/search	26	10	4.5
BusinessWeek	bwarchive.businessweek.com	13	20	3.9
Sun	www.sun.com	20	10	3.7
AltaVista	www.altavista.com	19	10	4.1

Sources for which the algorithm failed to generate a wrapper	
source	URL
Excite	www.excite.com
CS Bibliography (Trier)	www.informatik.uni-trier.de/~ley/db/index.html
Library of Congress	lcweb.loc.gov
FtpSearch	shin.belnet.be:8000/ftpsearch
CS Bibliography (Karlsruhe)	liinwww.ira.uka.de/bibliography/index.html
IICM	www.iicm.edu

* Only accessible to members of the Calliope library group.

for 16 of the 22 sources. For 2 other sources the generated incorrect wrappers could easily be corrected. The working wrappers were created with only one answer item labeled. This means that good generalizations are being made when inducing the grammar for the items; labeling only *one* item of *one* page is sufficient to create wrappers for many other items and pages. Table 1 summarizes our experimental results; the fourth column, labeled NI, contains the total number of items on the page. The times displayed in Table 1 were measured on a modest computer (PC AMD 200MMX/32 RAM). Still, the time to generate a wrapper is very short; it took at most 32.5 seconds, with the average time being 7.8 seconds. Together with the small amount of labeling that has to be done, this makes our approach to generating wrappers a very rapid one.

Increasing the *SeparatorLength* value (see Section 4) makes our algorithm faster, as fewer fragments of HTML are taken into account. For NCSTRL, the time to generate a wrapper is shown with a *SeparatorLength* of 1 (32.5 seconds), as 1 is the default *SeparatorLength*. However, with a *SeparatorLength* of 2, it takes 22.5 seconds, with 3 it takes 21.4 seconds, and with 4 17.1 seconds.

Robustness of the Wrappers. An important aspect of the generated wrappers is the extent to which the result pages of the search services may change without the wrapper breaking down. The wrappers we generate are not very robust. Little is allowed to change in the list with search results, because the wrapper for that list is generated so as to closely resemble the original HTML code. But even if

the wrappers are not very robust, it is easy to create a new wrapper whenever the search engine's result pages change. Our algorithm is fast and does not need much interaction, which makes it unproblematic to generate a new wrapper.

Incorrect Wrappers. There are various reasons why our algorithm failed to produce working wrappers for the six sources mentioned in Table 1. In some cases the HTML of the result pages was incorrect (Excite, IICM). In another case attributes were only separated by textual separators and not by HTML tags, making it impossible to create a wrapper for it with our algorithm (Library of Congress). In some cases the algorithm failed to create a working wrapper because the right items were not found due to too much variation in the items (Computer Science Bibliography Trier, FtpSearch). And in another case there was too much variation in the items and in the hierarchical way in which they were presented (Computer Science Bibliography Karlsruhe).

Grammar Evaluation. How do we determine that an induced grammar is correct? Like in all other approaches, the grammar induction is called *successful* if the grammar extracts correctly all items from the example page. For certain sources, one result page was insufficient and more pages were needed to learn all structural variations and induce the working wrappers. However, in all these cases, once the grammar was successfully induced for the initial, labeled page, it was always possible to extend it to new result pages, without additional labeling.

In the general case, the Probably Approximate Correct (PAC) technique is used to estimate the grammar accuracy; however, since our method is really fast at incorporating new structural variations, we found that it is easier to keep incorporating forever; we omit details here.

Comparison to Other Approaches. Most alternative approaches differ from ours in significant ways. Some are far simpler [8], or specify wrappers manually at a far more abstract level [6]. Others differ in that they are based on static templates instead on learning the structure of result pages [12]. Still others are based on assumptions about the structure and lay-out cues [3,16]. Some approaches need much more user interaction as the user has to label several entire pages [13]. The approach of Hsu, Chang and Dung [9,10] is the one that is most similar to ours. Their finite-state transducers, called *single-pass SoftMealy extractors*, resemble the grammars that we generate, although they abstract pages in a more fine-grained way. In their approach, textual content is further divided, e.g., in numeric strings and punctuation symbols. The approach seems to create more robust wrappers than ours, but at the price of more extensive user input. Further comparisons — empirical and analytic — of the approaches are needed to understand the trade-off between user interaction and quality of the wrappers.

7 Conclusion and Further Work

We have presented an approach to automatically generate wrappers. Our method uses grammar induction based on an adapted form of the *edit distance*. Our

wrapper generator is language independent, because it relies on the structure of the HTML code to build the wrappers. Experimental results show that our approach is accurate — 73% (allowing minor modifications: 82%) of the wrappers generated are correct. Our generator is quick, as it takes less than 10 seconds to generate a wrapper for most sources. The most important advantage of our approach is that it requires minimal user input; it suffices to label only one item on the page for which the wrapper has to be generated; the other items are found by the wrapper generator itself.

Although our wrapper generator works well, it can be extended and improved in several ways. For a start, it would be useful if the user could label attributes in a graphical interface that hides the HTML code. Second, we need to extend the wrapper to generate code to handle no result pages. Also, we would like to experiment with relaxing our assumption that all attributes are separated by HTML tags. Further, if a lot of search engines for a specific domain have to be connected to a meta searcher, it may be worthwhile to create *recognizers* [12], modules that find and label the attributes on the page. Finally, we have deliberately investigated the power of our method with minimal user input, but conjecture that labeling more answer items and selecting them carefully improves performance.

References

1. Aho, Alfred V. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 255–300, Elsevier, 1990. 100, 101
2. Andreoli, J.-M., Borghoff, U., Chevalier, P-Y., Chidlovskii, B., Pareschi, R., and Willamowski, J. The Constraint-Based Knowledge Broker System. *Proc. of the 13th Int'l Conf. on Data Engineering*, 1997. 96
3. Ashish, N., and Knoblock, C. Wrapper Generation for Semi-structured Internet Sources. *SIGMOD Record* 26(4):8–15, 1997. 97, 106
4. Chidlovskii, B., Borghoff, U., Chevalier, P.-Y. Chevalier. Toward Sophisticated Wrapping of Web-based Information Repositories. *Proc. 5th RIAO Conference, Montreal, Canada*, pages 123–135, 1997. 97
5. Florescu, D., Levy, A., and Mendelzon, A. Database techniques for the World-Wide Web: A Survey. *SIGMOD Record 27(3)*:59–74, 1998. 97
6. Garcia-Molina, H., Hammer, J., and Ireland, K. Accessing Heterogeneous Information Sources in TSIMMIS. *AAAI Symp. Inform. Gathering*, pages 61–64, 1995. 96, 106
7. Gauch, S., Wang, G., Gomez, M. ProFusion: Intelligent Fusion from Multiple Distributed Search Engines. *J. Universal Computer Science*, 2(9): 637–649, 1996. 96
8. Hammer, J. Garcia-Molina, H., Cho, J., Aranha, R., and Crespo, A. Extracting Semistructured Information from the Web. *Proceedings of the Workshop on Management of Semistructured Data*, 1997. 106
9. Hsu, C.-N., and Chang, C.-C. Finite-State Transducers for Semi-Structured Text Mining. *Proc. IJCAI-99 Workshop on Text Mining*, 1999. 97, 106
10. Hsu, C.-N., and Dung, M.-T., Generating finite-state transducers for semistructured data extraction from the web. *Information Systems*, 23(8):521–538, 1998. 97, 106

11. JavaCC – The Java parser generator. URL: http://www.metamata.com/JavaCC/. 104
12. Kushmerick, N., Weld, D.S., and Doorenbos, R., Wrapper Induction for Information Extraction. *Proc. IJCAI-97*: 729–737, 1997. 97, 106, 107
13. Muslea, I., Minton, S., Knoblock, C. STALKER. *AAAI Workshop on AI & Information Integration*, 1998. 97, 106
14. Ragetli, H.J.N. Semi-automatic Parser Generation for Information Extraction from the WWW. *Master's Thesis, Faculteit WINS, Universiteit van Amsterdam*, 1998. 98, 101
15. Sakakibara, Y. Recent advances of grammatical inference. *Theoretical Computer Science* 185:15–45, 1997. 99
16. Soderland, S. Learning to Extract Text-based Information from the World Wide Web. *Proc. KDD-97*, pages 251–254, 1997. 106