Wrapper Generation via Grammar Induction

Boris Chidlovskii[1] Jon Ragetli[2] Maarten de Rijke[2]

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

January 7, 2000

Abstract

To facilitate effective search on the World Wide Web, several '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 relevant information from the HTML pages returned by search engines. In this paper we present an approach to create such wrappers automatically 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, it can be used for several types of result pages and it requires a minimal amount of interaction with the user.

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

Type of submission: Full Paper

Contact person: The corresponding author is Jon Ragetli.
Phone: +31 20 525 5235. Fax: +31 20 525 5101.

1 Introduction

As the amount of information available on the World Wide Web continues to grow, conventional search engines, like AltaVista, Yahoo, Lycos and Infoseek, expose certain 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.' A wrapper is a program that takes 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 HTML result page of the search engine. In the following, we will refer to the latter as 'wrapper' and do not discuss the query translation (see [5] for a good overview). As input, wrappers take an HTML result page of a search engine; such pages contain 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 Figure 1 three tuples can be extracted, the first of which is displayed in Figure 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).

The manual programming of 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. Hence, there have been various attempts to automate this task [3, 9, 10, 12, 13].

In this paper we describe our approach, which is based on a simple incremental grammar induction algorithm. As input, our algorithm requires one result page of a search engine, from 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 the help of an adapted version of the *edit distance* measure, further answer items on the page are found. The adapted edit distance method also indicates how to update the current extraction grammar in order to cover newly found items. Once all the items have been found

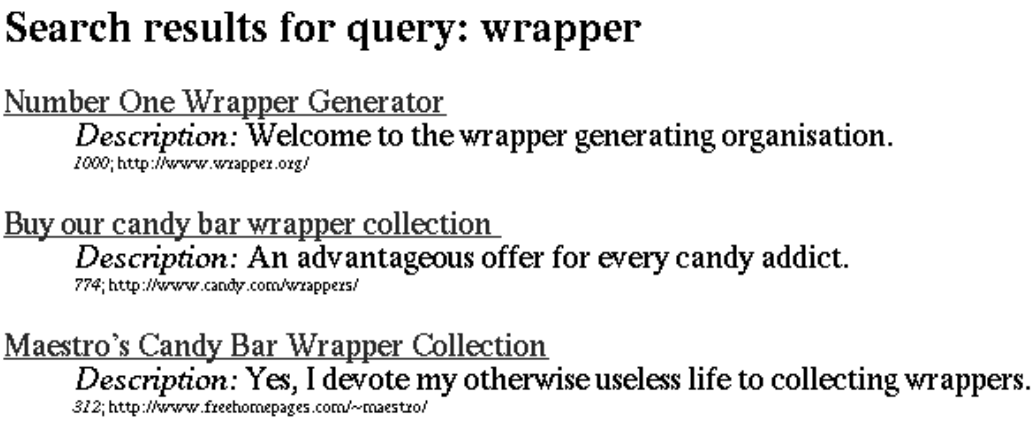

Figure 1: Sample result page

and the grammar has been adapted accordingly, some post-processing takes place, and the algorithm returns a wrapper for the entire page. The key features of our approach are the small amount of 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 remainder of this paper is organized as follows. In the next section show how to use grammar induction for the construction of wrappers. After that we describe our wrapper learning algorithm. Next we discuss experimental results and we conclude with suggestions for improvement and a comparison to alternative approaches. Full details can be found in [14].

2 Using Grammar Induction

In this section we show how grammar induction is used for generating a wrapper. To this end, we view the *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 $(i = 1, 2, \ldots) \in \mathcal{A}$ 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

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

Figure 2: An item extracted

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

might correspond to the string $t\ a_1\ \overline{t}$, where t and $\overline{t}$ are symbols of Σ which denote tags `<title>` and `</title>`. The text 'Wrapper Induction' has to be extracted as the value of attribute $a_1 \in \mathcal{A}$.[1]

The problem that we want to solve can be phrased as follows: to construct a wrapper W that is able to extract all relevant information from a given labeled page and unseen pages from the same source. We solve the problem of generating wrappers by decomposing it into two simpler subtasks. The first task is to find an expression that locates the beginning (`Start`) and the end (`End`) of the list of answer items. The second 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. 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.

In Section 2.1 we discuss the input to the grammar induction algorithm and in Section 2.2 the form of the grammar we induce.

2.1 Preprocessing the HTML page

What do we assume about the input for the grammar induction? All known approaches for automatically generating wrappers require as input one or more labeled HTML pages. This means that all or some of the attributes to be extracted from the page have to be indicated by the user, or by some labeling program. Many algorithms require *all* the attributes to be labeled: for all answer items on the result page, the user has to tag every attribute value with the appropriate attribute name [12, 13]. Obviously, this labeling is a boring and time-consuming job, so we have restricted the labeling for our algorithm to a single answer item only.

Let us illustrate the labeling that we require by means of an example. Figure 3 shows the labeled source for the HTML page in Figure 1. The labeling consists of the following:

- an indication of the begin and end of the first answer item (`^BEGIN^` and `^END^`, respectively),

[1] This representation is somewhat simplified. The program can also extract tag attributes, such as the `href` attribute for the `A` tag, by splitting 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>
```

Figure 3: Labeled HTML source of result page

- the names of the attributes (e.g. `^URL^`), and
- the end of the attributes (`^^`).

After the item has been labeled by the user, it is first *abstracted* by our algorithm to turn it into a string over $\Sigma \cup \mathcal{A}$; this is done by a simple tokenizer that distinguishes between different HTML tags and labeled and void attributes.

2.2 The Item Grammar

The item grammar mentioned before 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]. In the following we describe the restricted form of the grammar that we aim to learn. First, we describe it as a simple form of FSA which we call sFSA, where transitions labeled with an a_i from $\mathcal{A}$ (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.

The most simple form of FSA is what we call a *linear FSA*, and is defined below.

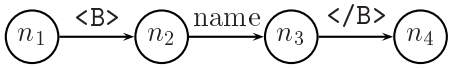

Figure 4: A linear FSA

Definition 2.1 (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 from $\mathcal{A}$, then $e_{i-1,i}$ and $e_{i+1,i+2}$ are labeled with an element from Σ.

The fact that the attribute a in Definition 2.1 is surrounded by HTML tags (from Σ) allows us to extract the attribute. Figure 4 shows a linear FSA that can only extract the attributes from *one* type of item: an item that has an attribute *name* between `<B>` and `</B>` tags.[2] Therefore, it is not very useful. The sFSAs that we will employ to learn the structure of the items, are a bit more complex.

Definition 2.2 (simple FSA) A linear FSA that also has ϵ-transitions $s_{i,j}$ (transitions labeled with $\emptyset$) 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, for example, do not contain cyclic patterns.

[2]The symbols from Σ, like `<B>` and `</B>` in Figure 4, represent tokens for *abstracted tags*. For example, `<font>` would represent any font tag: `<font size = 10>`, `<FONT face = "helvetica">`, etc.

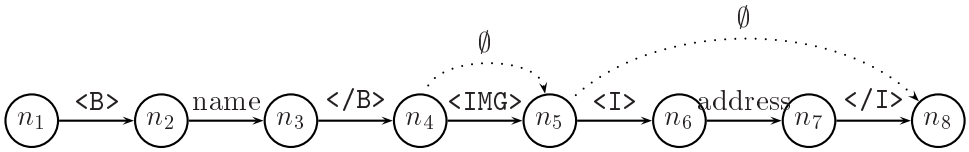

Figure 5: An sFSA

So where do grammars come in? Another way to represent the language defined by an sFSA is by means of a simple kind of regular expression with fixed and optional parts. Using brackets to indicate optional parts, the sFSA of Figure 5 can be represented as

<B>name</B>[<IMG>][<I>address</I>]

By including or discarding optional parts, this expression functions as a grammar defining the same sequences of abstracted tags and content as the sFSA. We will refer to this representation as *item grammar* or simply *grammar*.

Although item grammars are very simple, they are general enough to extract the attributes from the answer items, because the HTML pages for which the grammar has to be learned, are created dynamically and thus have regular structure. Furthermore, after the grammar has been learned, it is processed into a more general form containing repetitions (see Section 5).

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 to calculate *string edit distance* [1]. The simple form of the grammar makes this possible. The edit distance between two strings is defined as follows:

Definition 3.1 (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, $D(abcd, abide) = 3$, because in order to transform *abcd* into *abide* at least three insertion or deletion operations have to be performed: delete the c in *abcd*, and insert an i and an e. 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. We will not discuss this algorithm in more detail; the interested reader is referred to [1, 14].

We have adapted the edit distance algorithm in such a way that it is able to calculate the distance between an item grammar and an item. An item is already denoted by a string of symbols. Thus the adaptation was only needed to cope with the item grammar, which can be viewed as a string of symbols with optional parts.

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: both are strings of symbols. Using the alignment and the remembered position of the brackets, the new grammar is calculated.[3]

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 describe below how the algorithm works by some examples; a more elaborate description can be found in [14].

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

As the item grammar did not contain a c, whereas the string to be covered did, 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.

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

The reason for making the d optional is that the new string shows that it does not occur in every string. The new item grammar now covers the strings ab, abc, abd and $abcd$. This shows that it is a bigger generalization than simply 'remembering' all the examples.

The alignment and the resulting item grammar below require some explanation.

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

The new item grammar $a[b][c][b]d$ is a large generalization; besides abd and acd it covers ad, $abcd$, $acbd$ and $abcbd$ as well, i.e., five other strings besides the original item grammar and the example. The reason we decided to have

[3] Actually, we 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.

```
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
```

Figure 6: The Local Optimum Method

a large generalization is that based on the examples we can conclude at least that the b and c are optional. Moreover, they may also occur at the same time and in any order. That is what we capture with this generalization.

4 Finding Answer Items

In the previous section, we have discussed the learning of the grammar based on the answer items on the HTML page. However, as only the first answer item on the page has been indicated by its labeling, the other answer items have to be found. For this, we use the distance as 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 quicker methods, but even with the LOM a wrapper for a standard HTML page 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 6 shows the algorithm. In the first three steps, a number of variables are initialized. Let us look at the outermost *while* loop now. Once the previous item has been found, or the first labeled item, the LOM looks for the next occurrence of the begin delimiter. After that 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. With the same begin delimiter as before, 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 the occurrence of the begin delimiter after the begin delimiter considered previously will be treated as the potential start of another item. In other words, potential items a bit lower on the page are considered now. 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 an item that was incorporated previously. Its initial value is set by the user, and it is incremented whenever an item is incorporated whose distance is higher than *HighDistance*. *Variation* is a value that is not adapted during the process of finding the items.

Example 4.1 Consider Figure 7. Suppose that the item grammar does not have any optional parts yet, and is of the form *abcdefgh*. Values for *HighDistance*, *Variation* and *Threshold* are 2, 1.6, and 3.2, respectively. Now LOM comes up with an item *aefgh*. The distance between the item grammar and the string is 3, which is smaller than *Threshold*, and hence the string will be incorporated. The new item grammar is *a*[*bcd*]*efgh*, and *HighDistance* now equals 3. Suppose the next string LOM comes up with is *aefh*. In fact this string is almost covered by the item grammar, except for the missing *g*. Therefore, it probably really is an item, although the distance calculated is quite high: 4. If *HighDistance* had not been increased in the previous update of the item grammar, this potential item would not have been incorporated, simply because the distance calculating algorithm does not take into account that *bcd* is an optional part of the item grammar. So

									HD	Var	Thr
grammar	a	b	c	d	e	f	g	h	2	1.6	3.2
item	a	–	–	–	e	f	g	h	2	1.6	3.2
grammar	a	$[b$	c	$d]$	e	f	g	h	3	1.6	4.8
item	a	–	–	–	e	f	–	h	3	1.6	4.8

Figure 7: Example of the use of *HighDistance* and *Variation*

```
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

Figure 8: The wrapper generating algorithm

an increase in *HighDistance* compensates for the simplicity of the distance measure.

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 Figure 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 2.1. The second step initializes the grammar G to the first, labeled item. In the third step, `find-next-item` is the algorithm for finding the 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 the 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 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 2.2, 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. As useless fragments in a page do not contain attributes to be extracted, the trash grammar will consist of symbols in $\Sigma \cup \{a_0\}$. The indices of the items that were found were stored, so this process is a straightforward repetition of `incorporate-item`. When 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. For example, when we have a grammar like $abcd[bcd][bcd]ef$, it is generalized to $a(bcd)^+ef$. For reasons of efficiency, the generalization is postponed until *after* the item and trash grammars have been constructed. Observe that our specific form of generalization (i.e., detecting repetitions) is appropriate here, because parts can re-occur arbitrarily often on search result pages; for instance, the output of a bibliographical search engine may be a list author names, often enclosed between `<a href = "...">` and `</a>` tags.

After all these processing steps, we have an abstract wrapper of the form:

$$\texttt{Start}\ (\texttt{Item Trash})^+$$

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 Figure 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

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 (Univ. Trier)	www.informatik.uni-trier.de/~ley/db/index.html
Library of Congress	lcweb.loc.gov
FtpSearch	shin.belnet.be:8000/ftpsearch
CS Bibliography (Univ. Karlsruhe)	liinwww.ira.uka.de/bibliography/index.html
IICM	www.iicm.edu

* Only accessible to members of the Calliope library group.

Table 1: Experimental results

had already been connected manually. It was quite successful, as it created working wrappers 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.

We will now discuss our experimental results in more detail.

6.1 Speed of the Wrapper Generator

Observe that 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 for a large file with a large number of items. The average time needed was 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 parameter *SeparatorLength* (see Section 4) makes our algorithm faster, as fewer fragments of HTML are taken into account. For example, 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 only 17.1 seconds.

6.2 Robustness of the Wrappers

An important aspect of the generated wrappers is their robustness, that 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. Not much 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. The top of the page can change as long as it does not contain an initial segment of the item list. Finally, the HTML code at the bottom of the page can change freely, as this part will never be parsed. 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.

6.3 Incorrect Wrappers

We will now discuss why our algorithm does not work for the six sources mentioned in Table 1.

The wrapper generated for Excite did not work because the code to extract the URL from `<a href = "...">` tags was not general enough and did not recognize the unquoted URL in the Excite answer page. We have corrected the wrapper manually, and after this it worked properly. A similar correction produced a working wrapper for IICM.

For the Library of Congress, there were more serious reasons for our failure to generate a working wrapper. As should be clear from the exposition of the algorithm, our wrapper generator distinguishes between HTML tags, represented as symbols from Σ and text, represented as symbols from $\mathcal{A}$. On the Library of Congress result pages, however, the attributes were only separable by textual separators and not by HTML tags, making it impossible to create a wrapper for it with our algorithm.

For the Computer Science Bibliography (University of Trier) and FtpSearch the algorithm did not create a working wrapper because the right items were not found due to too much variation in the items, causing the distance between the item expression and the item found to be too high. Starting the algorithm with a higher value of *HighDistance* or *Variation* did not improve this situation, because in that case fragments of HTML that did

not correspond to an item were incorrectly incorporated in the grammar.

The problem with the Computer Science Bibliography (University of Karlsruhe) concerned the detection of repetitions in the item expression. Complex repetitions on the page make the wrapper generator create a repetitive part of the form $([a][b][c][b])^+$, an expression that cannot be translated into a working wrapper, as it would enter an infinite loop. Another problem was that the result pages are divided in several sections, each of which is headed by the source from which the answers in the section were retrieved. As this source is not consistently mentioned with every answer, we cannot extract it.

7 Further Work

Although our wrapper generator works well, it can be extended and improved in several ways. We will now discuss a number of those extensions and improvements.

Improved user-interface. At present, the user has to label the attributes by inserting special tags in the HTML page with some text editor. It would be better, and require less effort from the user, if he or she could label it in a nice graphical interface, so that the HTML code can be hidden.

No result pages. The page most search engines return when having found no appropriate search results is very different from the ones returned when they did find relevant results. At the moment, the code to recognize them has to be manually inserted in the parser code. It is not difficult to write this code, so it is easy to automate this as well.

Relax HTML separability. One of the assumptions underlying our wrapper generator is that all attributes can be separated by HTML tags. As we noted before, not all HTML pages satisfy this requirement. By making our abstraction more fine-grained, we should be able to generate wrappers for such pages. Text should be divided in more tokens; for instance, on some pages a '–' is used to separate attributes, so this should be a separate abstraction, and as a consequence, a separate token in the generated grammar. If the user indicates which textual separators to use before the wrapper will be generated, those can be integrated quite easily in the algorithm.

On the other hand, the HTML separability causes the wrapper generator not to rely on specific textual content on the pages. That makes this approach language independent, a good property of the wrapper generator.

Recognizers. If a lot of search engines for a specific domain (e.g. scientific publications) have to be connected to a meta searcher, it can be worthwhile

to create *recognizers* (see for example [12]), software modules able of finding and labeling the attributes on the page. This might make it possible that the user does not have to label anything at all.

More sample items. We have intentionally investigated the power of our method with minimal user input, but conjecture that labeling more answer items and selecting them carefully improves performance. Further research is needed to verify this hypothesis.

8 Comparison to Other Approaches

In [8], a simple approach to semi-automatically generating wrappers is presented that lies in between hand-coding the wrappers and creating them fully automatically. Wrappers in the TSIMMIS project [6] are made by specifying them at a high level. Thus, their approach is simpler than ours.

Kushmerick et al. [12] present a template-based approach for building wrappers for HTML sources. They use recognizers to label the page automatically, which relieves the user from doing it. That is very useful, as their algorithm requires pages that are entirely labeled. The user only has to select the recognizers to be used to label the page.

In [3], Ashish and Knoblock present quite a different approach to generating wrappers. They focus on building wrappers that make static HTML pages queriable. Their wrappers are constructed not with the help of any labeling, but by *structuring* the page. This structuring is based on certain assumptions about how the nesting hierarchy within a page is reflected in the layout. A user does not need to do anything as long as those assumptions are true for the page considered. If they are not true, the user has to correct the wrong structuring performed by their algorithm. Their algorithm is not well-suited for making wrappers for our domain (pages with search results), because we think it is not possible to find general heuristics applicable to multiple search engines.

Ashish and Knoblock [3], Soderland [16] uses lay-out cues to construct wrappers. Furthermore, Soderland's system uses a semantic lexicon which makes the approach very different from ours. Besides automatically generated pages (like the search engines connected to Knowledge Brokers), his domain consists of less structured hand-crafted pages.

Muslea, Minton and Knoblock [13] discuss the automatic generation of hierarchical wrappers. A drawback of their approach is that the user has to label several entire pages, although it is an advantage to our approach that this can be done in a graphical user interface. The hierarchical wrappers do not suffer from the problem we mentioned with respect to the Computer Science Bibliography at the University of Karlsruhe, that attributes belonging to several different items cannot be extracted. The hierarchical form of the

wrappers makes it possible to decompose the computationally hard problem of generating wrappers for entire result pages into smaller problems. While our approach is bottom-up, the approach of Muslea, Minton and Knoblock is top-down; we start with the HTML code and generalize it when finding other items on the page, whereas they start with the attributes on the labeled page and try to find the unique start and end tags of the attribute within the parent.

The approach of Hsu, Chang and Dung [9, 10] is 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, in numeric strings and punctuation symbols, amongst others. This finer grained distinction makes their approach applicable to other text than HTML pages. Their tuple transducers make it possible to represent disjunctions more straightforwardly than in our approach. Experimental results show that their approach, like ours, does not need many labeled items, albeit more labeled items than ours. It seems that their approach can handle differences in the order of the attributes better than ours, but we have not fully tested this. A further investigation of the differences between the two approaches, and the tradeoff between user input and quality of the wrappers should make this clear.

9 Conclusion

In this paper we have presented an approach to automatically generate wrappers. For this purpose, our method used grammar induction based on an adapted form of *edit distance*, a method to compare strings. 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% of the wrappers generated is correct; when small adjustments to the generated wrappers are allowed, 82% of them works correctly. Furthermore, 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 the user does not have to do much labeling; 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. Comparing our algorithm to others, we conclude that it creates good wrappers with little user interaction. The approach of Hsu et al. [9, 10] seems to create better wrappers, but at the price of more extensive user input.

Acknowledgments. Jon Ragetli was supported by the Logic and Language Links project funded by Elsevier Science Publishers. Maarten de Rijke was supported by the Spinoza project 'Logic in Action.'

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.

[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.

[3] Ashish, N., and Knoblock, C. Wrapper Generation for Semi-structured Internet Sources. *ACM SIGMOD Workshop on Management of Semistructured Data*, 1997.

[4] Chidlovskii, B., Borghoff, U., Chevalier, P.-Y. Chevalier. Toward Sophisticated Wrapping of Web-based Information Repositories. *Proc. 5th RIAO Conference, Montreal, Canada*, 1997.

[5] Florescu, D., Levy, A., and Mendelzon, A. Database techniques for the World-Wide Web: A Survey. *SIGMOD Record 27(3)*, pages 59–74, 1998.

[6] Garcia-Molina, H., Hammer, J., and Ireland, K. Accessing Heterogeneous Information Sources in TSIMMIS. *Proc. AAAI Symp. on Information Gathering*, 1995.

[7] Gauch, S., Wang, G., Gomez, M. ProFusion: Intelligent Fusion from Multiple Distributed Search Engines. *Journal of Universal Computer Science*, 2(9), 1996.

[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.

[9] Hsu, C.-N., and Chang, C.-C. Finite-State Transducers for Semi-Structured Text Mining. *Proc. IJCAI-99 Workshop on Text Mining: Foundations, Techniques and Applications*, 1999.

[10] Hsu, C.-N., and Dung, M.-T., Generating finite-state transducers for semistructured data extraction from the web. *Information Systems*, 23(8), 1998.

[11] JavaCC – The Java parser generator. URL: `http://www.metamata.com/JavaCC/`.

[12] Kushmerick, N., Weld, D.S., and Doorenbos, R., Wrapper Induction for Information Extraction. *Proc. IJCAI-97*, 1997.

[13] Muslea, I., Minton, S., Knoblock, C. STALKER: Learning Extraction Rules for Semistructured, Web-based Information Sources. *AAAI Workshop on AI & Information Integration*, 1998.

[14] Ragetli, H.J.N. Semi-automatic Parser Generation for Information Extraction from the WWW. *Master's Thesis, Faculteit WINS, Universiteit van Amsterdam*, 1998.

[15] Sakakibara, Y. Recent advances of grammatical inference. *Theoretical Computer Science* 185:15–45, 1997.

[16] Soderland, S. Learning to Extract Text-based Information from the World Wide Web. *Proc. KDD-97*, 1997.