Automatic Wrapper Generation for Web Search Engines

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, 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 algorithm to create such wrappers automatically, based on an adaptation of the *string edit distance*. Our algorithm 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.

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, 6–8]. Instead of searching the Web themselves, MSEs exploit existing search engines to retrieve relevant information and combine it in a way which better satisfies the user's needs. This relieves the user from having to contact those search engines manually and knowing their native query languages; knowledge of the MSE's query language suffices. 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*: software modules that take care of the source-specific aspects of the 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 extracts the relevant information from the HTML result page of 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

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

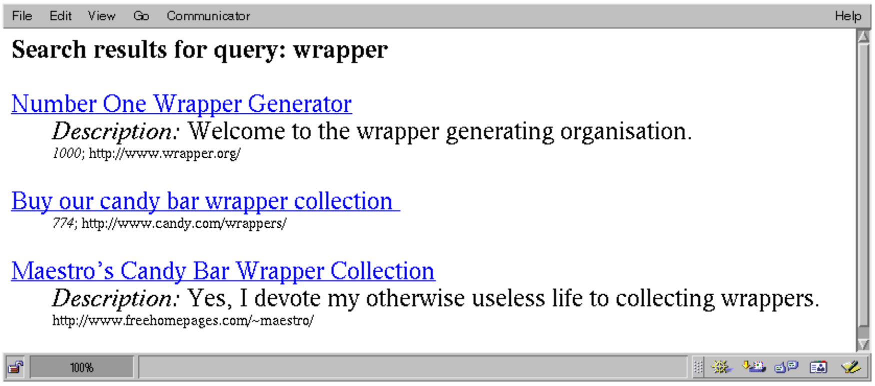

Fig. 1. Sample result page

a search engine contains zero or more *answer items*; an answer item is a group of coherent information making up one answer to the query. A wrapper extracts each answer item from the textual content and attributes of certain tags on the page 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. Like most result pages, the page in Fig. 1 shows variation in the items, as the second item lacks a description and the third a relevance ranking.

Manually programming wrappers is a cumbersome and tedious task [4], and since the presentation of the search results of search engines often changes, it has to be done frequently. Hence, there have been various attempts to automate this task [3, 10, 11, 13, 14]. The approach we describe is based on a simple incremental grammar induction algorithm. As input, it requires one result page of a search engine, from which the first answer item is labeled: the start and end of the item need to be indicated, as well as the attributes to be extracted. After this, the incremental learning of the *item grammar* starts, and using an adapted version of the *edit distance* measure, further answer items on the page are found and updates to the grammar are performed. Once this process is ready, the algorithm returns a wrapper for the entire page after some post-processing.

The key features of our approach are limited user interaction (labeling only one answer item) and good performance: for many search engines it generates working wrappers, and it does so very quickly. In the following sections, we will first discuss our wrapper generating algorithm. After this, experimental results

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

Fig. 2. An item extracted

```
<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>
             ⋮
  </dl>
</BODY></HTML>
```

Fig. 3. Labeled HTML source of result page

will be described, and we compare our method to other approaches. Finally, we conclude and discuss future work.

2 The Wrapper Generator

In this section we discuss the input and output of the Wrapper Generator (WG), after which we give an overview of its algorithm. In the next section we will go into more detail. All known approaches for automatically generating wrappers require as input some labeled HTML pages: all or some of the attributes to be extracted from the page have to be marked — manually, or automatically by a labeling program. As it is hard to create labeling programs for the heterogeneous set of search engines 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 of the HTML page in Fig. 1. The labeling consists of an indication of the begin and end of the first answer item (`^BEGIN^` and `^END^`, respectively), the attribute names (e.g. `^URL^`), and the end of the attributes (`^^`).

The output of the WG is a wrapper, whose task is to output zero or more *tuples* consisting of relevant information. Each element of the tuple is an attribute/value pair; the attribute names are provided by the human who labeled the page. Figure 2 shows the first tuple extracted from the result page in Fig. 1.

Our wrapper generation algorithm, shown in Fig. 4, works as follows. First, the result page is abstracted to a sequence of tokens, after which an *item grammar* is initialized. Then, using this item grammar, other items on the page are

```
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 := grammar-whole-page(G, AP)
6. W := translate-to-wrapper(GP)
7. return W
```

- AP is the abstracted page
- LP is the HTML page labeled by the user
- G is the item grammar
- I is an item on the abstracted page
- GP is a grammar for the whole page in an abstract format
- W is the same grammar, translated into a working wrapper

Fig. 4. The wrapper generating algorithm

found, and the grammar itself is updated to cover those items. Once all the items have been found, the grammar is extended to cover the entire page, and finally translated into a working wrapper.

Steps 2, 3 and 4 show that the grammar building algorithm is incremental: the grammar G is updated whenever a new item is encountered. The fact that the algorithm itself finds the items on the page saves the user the burden of labeling every item on the page, a feature that other approaches (e.g. [3, 13, 14]) lack. Another feature is that the user does not have to indicate the begin and end of the item precisely. When an item has been indicated more narrow than it actually is, step 5 takes care of finding the real item size. In Fig. 3 the item is indeed indicated too narrow; the fragment of HTML between `<DL>` and `</DL>` is repeated on the page, instead of the smaller item indicated there.

3 Details of the Algorithm

For presentational purposes we do not describe the steps in the same order as the algorithm performs them. The paragraphs describe steps of the algorithm as well as underlying theory.

Step 1: Abstract. In the `abstract` step, the entire HTML page is transformed into a string of tokens. Each tag is abstracted to one token for that tag,[1] and everything else on the page is abstracted to the token C (for Content). By representing each token by a symbol, we can now view the HTML page as a string of symbols. This allows us to use an algorithm for comparing strings, as described below, to find the items on the page other than the labeled first one.

[1] This abstraction generalizes over the contents of the tag. E.g. both tags `<font face="Arial">` and `<font face="Helvetica">` are abstracted to `<FONT>`.

The Item Grammar. The item grammar is an important foundation of the WG algorithm. It both represents the structure of the items already encountered on the page, and it is used to find the next item on the page. It is defined as follows:

Definition 1 (Item Grammar).

1. *A symbol is an item grammar.*
2. *If G_1 and G_2 are item grammars, then G_1G_2 is an item grammar, meaning G_1 concatenated to G_2.*
3. *If G is an item grammar without substrings of the form '$[S]$', where S can be any string, $[G]$ is an item grammar.*

Since an HTML page is abstracted into a string of symbols, the item grammar represents sequences of content and HTML tags. The square brackets represent optionality; e.g. $a[b]c$ covers both the strings abc and ac. The third clause of the definition prevents optional parts from being nested. By including or discarding optional parts, the item grammar defines sequences of HTML and content that can appear on the result page. Although item grammars are less expressive than regular grammars, they are expressive enough to grasp variations in the item structure.

Step 2: Initialize. In the `initialize` step, the item grammar is initialized by making it equal to the string of symbols that represents the first, labeled item on the page. The user has labeled the attributes in the first item with a name; the algorithm remembers their position and name.

Edit Distance. The simple form of the item grammar makes it possible to use the grammar to find other items on the page in `find-next-item`, using an adapted form of *edit distance*. We will first recall the edit distance algorithm for strings [1]. Next, we discuss our adapted form of edit distance.

Definition 2 (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. Algorithms calculating the edit distance can also return the difference between the two strings in the form of a so called *alignment*. This difference is used in `incorporate-item` to indicate how to adapt the item grammar on the basis of the new item (see the next subsection). As an example, for $abcd$ and $abide$ the alignment is the following:

$$\begin{array}{cccccc} \hline a & b & c & - & d & - \\ a & b & - & i & d & e \\ \hline \end{array}$$

Here we omit the details of the edit distance and alignment algorithm and refer the interested reader to [1, 15].

Let G_i denote the item grammar constructed for the first i items. In steps 3 and 4 of our algorithm (Fig. 4), we calculate the distance between a grammar

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. 5. Three alignments

G_i and a string representing the $i+1$-th item, in order to construct the item grammar G_{i+1} covering all $i+1$ items. Therefore, we have adapted a method to calculate edit distance to work for an *item grammar* and a string instead of only for strings. 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, as will be described next.[2]

Step 4: Incorporate-item. The algorithm detects and processes different cases in the alignment between G_i and $i+1$-th item. Since the full algorithm description is extensive and space is limited, we can only illustrate how it works by means of some examples (Fig. 5). A more elaborate description is given in [15].

In Fig. 5 (a), the original item grammar does not contain a c, while the string to be covered does. Therefore, the new 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, as in Fig. 5 (b). The reason for making the d optional is that the new string shows that it does not occur in every string. Note that the new item expression now covers the strings ab, abc, abd and $abcd$, which is a larger generalization than the simple memorization of all the examples.

The situation in Fig. 5 (c) is less straightforward, as the new item grammar $a[b][c][b]d$ is a large generalization; besides the original examples abd and acd it covers ad, $abcd$, $abbd$, $acbd$ and $abcbd$ as well. The reason for this 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.

Step 3: Find-next-item. Incorporate-item generates an item grammar for a set of items on the page. The WG also uses the item grammar for finding these items. The finding mechanism is iterative, it starts with the first labeled item and uses grammar G_i to find $i+1$-th item. The finding of the items is crucial for our algorithm, as it allows us to have only one item labeled on the search result page. We have implemented three different strategies for finding the answer items, but as space is limited we only describe the one that works for most sources: the *Local Optimum Method* (LOM). The other two are simpler and mostly quicker methods, but even with the LOM wrappers are generated quickly; see Section 4.

[2] We have also adapted the edit distance algorithm to deal with labeled attributes in the grammar, that correspond with unlabeled content in the item.

1. $D_{newlocal} := 998$, $D_{local} := 999$, $D_{best} := 1000$
2. i_b, $i_e := 0$
3. `local-best-item` $:= \emptyset$, `best-item` $:= \emptyset$

`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` $:= \emptyset$
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. 6. The Local Optimum Method

All our methods for finding items are based on an important assumption: *all items on the page have the same begin and end tag(s).* Consequently, the task of finding items on the page is reduced to finding substrings (below the last found or labeled item) that have the same start and end delimiters. The user can decide for how many tags this assumption holds by setting the parameter *SeparatorLength.* If SeparatorLength is increased, it will be easier to find the items on the page; the chance of finding for example a sequence of two tags is smaller than that of finding one tag. However, setting the parameter too high will result in not finding all items.

The LOM finds 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. The outer *while* loop makes the start of potential items vary, while the inner while loop does the same for the end of the potential items. Thus, several combinations of begin and end tags are considered, and the one that is not far below the previous found item (*local*) and has a low distance to the grammar (*optimal*) is selected to be incorporated into the grammar.

In step 11, if the distance of the best candidate item to the item grammar exceeds *Threshold*, the algorithm will return $\emptyset$ instead of the item, thus preventing the grammar from adjustment, and the process of finding the items stops (see Fig. 4). *Threshold* depends on two parameters: *HighDistance* and *Variation*. *HighDistance* is the maximum distance from the grammar evaluated among all items incorporated previously. The initial value of *HighDistance* is set by user, and it is incremented whenever a new incorporated item has a distance higher than *HighDistance*. *Variation* is set by user as well, but it does not change during the process of finding the items. Combined together in *Threshold*, *HighDistance* and *Variation* form a flexible way for finding and incorporating new items.

Step 5: Making a Grammar for the Entire Page. Once the WG has found all the items on the page, and adjusted the item grammar accordingly, the item grammar can only extract the useful information from one item — not yet from the entire page. Here we recall that HTML pages for which a wrapper is generated, are assumed to contain a sequence of items, possibly mixed with irrelevant information. Above and below the sequence there might be irrelevant data as well. This assumption translates in a natural way into a skeleton for a wrapper:

```
1. skip the top of the page
WHILE there is another item
   2. parse item
   3. skip irrelevant data if present
```

It is not necessary to recognize the irrelevant data at the bottom of the page explicitly; the wrapper stops when it does not recognize any further items. For step 2, parsing the items, we already have the item grammar. But, as mentioned before, 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 item grammar) will also be too small. Therefore, the item grammar will be extended by finding common prefixes and suffixes of the HTML between the found items.

Two parts of the wrapper are still missing: a part that skips the top of the page, and a part that skips irrelevant data within the item list. For skipping the top of the page and recognizing the start of the list of items, the smallest fragment of HTML just above the first item is taken that does not occur higher on the page as well. For skipping the useless HTML between the items in the list, another kind of item grammar is constructed — the *trash grammar*. The indices of the (extended) items that were found have been stored, so this process is a straightforward repetition of `incorporate-item`. Once this trash grammar has been constructed, it is appended to the end of the item grammar. When the item and trash grammars have been generated, the WG will detect repetitive patterns in them and generalize them accordingly. This kind of generalization is appropriate, as certain fragments can re-occur arbitrarily often on result pages, like author names with enclosing tags.

After all these processing steps the WG translates the abstract grammar into a working wrapper. In our implementation this is a JavaCC parser [12], for the Knowledge Brokers meta search engine developed at Xerox Research Centre Europe is programmed in Java. However, our WG is not restricted to generate JavaCC parsers; the translation step can easily be replaced.

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 (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

* NI is the number of items.
** Only accessible to members of the Calliope library group.

4 Experimental Results

We have tested our wrapper generator on 22 search engines, a random selection of sources to which Knowledge Brokers had already been connected manually. It was quite successful, as it created working wrappers for 16 of the 22 sources (73%). For 2 other sources the generated incorrect wrappers could easily be corrected. Since working wrappers were created with only one answer item labeled, good generalizations are made when the item grammar is induced: labeling only *one* item of *one* page is sufficient to create wrappers for many other items and pages of the same source. Table 1 summarizes the experimental results; the times were measured on a modest computer (PC AMD 200MMX/32 Mb RAM).

The time needed to generate a wrapper is very short; the maximum time is 32.5 seconds, the average 7.8 seconds. Together with the small amount of labeling that has to be done, this makes our approach very rapid. In general, it is not the case that it takes more time to create a wrapper for larger pages than for smaller ones. A large page may contain a lot of irrelevant data at the top, where no items are sought, thus not increasing the time to create a wrapper. The same holds for pages with more items versus pages with less items. For a page with a lot of items that has very simple structure (for example when the begin and end delimiters only occur for real items) a wrapper can be learned more quickly than for a more complex page with less items.

Increasing the *SeparatorLength* parameter (see Section 3) speeds up our algorithm, as fewer fragments of HTML are considered. 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. For our wrappers, little is allowed to change in the list with search results, because the wrapper for that list is generated by making not too large generalizations. But even if the wrappers are not very robust, it is easy to create a new one whenever the search engine's result pages change, since the algorithm is fast and requires limited user interaction.

Incorrect Wrappers. 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. After manually correcting the wrapper, it worked properly. A similar correction produced a working wrapper for IICM. For the Library of Congress, the fact that the WG only distinguishes between tags and textual content caused it to fail, as the attributes on the result pages were only separable by textual separators.

The WG did not create a working wrapper for the Computer Science Bibliography (Univ. of Trier) and FtpSearch because the right items were not found due to too much variation in the items, causing the distance between the item grammar and the item found to be too high. Increasing the parameters *HighDistance* or *Variation* could not change this, because then fragments of HTML that did not correspond to an item were incorrectly incorporated in the grammar. The problem with the Computer Science Bibliography (Univ. of Karlsruhe) concerned the detection of repetitions in the item expression. Complex repetitions on the page make the wrapper generator create a repetitive part with only optional parts. This would cause the wrapper to enter an infinite loop. Another reason was that not all information belonging to an answer item was located with the item itself; some of it was shown in a header above a number of results.

5 Comparison to Other Approaches

In [9], Hammer et al. present a simple approach to semi-automatically generating wrappers that lies in between hand-coding the wrappers and creating them fully automatically: specifying them at a high level.

Kushmerick et al. [13] present a template-based approach for building wrappers for HTML sources. They use recognizers to label the page automatically. That is very useful, as their algorithm requires entirely labeled pages .

In [3], Ashish and Knoblock present quite a different approach to generating wrappers, focusing on making static HTML pages queriable. Their wrappers are constructed without labeling, but by *structuring* the page, using certain assumptions about how the nesting hierarchy within a page is reflected in the layout. Their algorithm is not well-suited for making wrappers for our domain (pages with search results), because there are no general heuristics applicable to multiple search engines. Soderland [16] also 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, his domain consists of less structured hand-crafted pages.

Muslea et al. [14] discuss the automatic generation of hierarchical wrappers. A drawback of their approach is that the user has to label several pages entirely, albeit in a graphical interface. The hierarchical wrappers do not suffer from the problem we mentioned with respect to the Computer Science Bibliography at the Univ. of Karlsruhe, that attributes belonging to several different items cannot be extracted. The hierarchical form of the wrappers makes it possible to decompose the problem of generating wrappers for entire result pages into smaller problems. While our approach is bottom-up, the approach of Muslea et al. is top-down.

The approach of Hsu et al. [10, 11] is similar to ours. Their finite-state transducers, called *single-pass SoftMealy extractors* resemble the grammars that we generate. Their abstraction of textual content on the pages is a more fine-grained. This makes their approach more widely applicable than to HTML pages. Experimental results show that their approach does not need many labeled items, albeit more than ours. It seems that their approach is better in handling differences in the order of the attributes, but we have not fully tested this. Further investigation — both empirical and analytical — of the differences between the two approaches should make this clear.

6 Conclusion and Further Work

We have presented an approach to automatically generate wrappers, which uses grammar induction based on an adapted form of *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% of the wrappers generated is correct (allowing minor modifications: 82%). Furthermore, our generator is quick, as the average time to generate a wrapper is less than 8 seconds.

The major advantage of our approach is the small amount of labeling by the user: labeling only one item suffices. 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. [10, 11] seems to create better wrappers, but at the price of more extensive user input.

Although the Wrapper Generator performs well, several improvements and extensions are possible. For example, a program with a graphical interface can simplify the labeling, that is currently performed with a text editor. One of the assumptions underlying our wrapper generator is that all attributes can be separated by HTML tags, but not all result pages satisfy this assumption. By making finer-grained abstractions, we should be able to generate wrappers for such pages. 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 natural language independent.

If many search engines for one domain have to be connected to a meta searcher, it is worthwhile to create *recognizers* [13] that find and label the attributes automatically. Finally, we have deliberately investigated the power of our method with minimal user input, but further research is needed to clarify the trade-off between user interaction and quality of the generated wrappers.

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. *SIGMOD Record* 26(4):8–15, 1997.
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.
5. Florescu, D., Levy, A., and Mendelzon, A. Database techniques for the World-Wide Web: A Survey. *SIGMOD Record 27(3)*:59–74, 1998.
6. Garcia-Molina, H., Hammer, J., and Ireland, K. Accessing Heterogeneous Information Sources in TSIMMIS. *AAAI Symp. Inform. Gathering*, pages 61–64, 1995.
7. Gauch, S., Wang, G., Gomez, M. ProFusion: Intelligent Fusion from Multiple Distributed Search Engines. *J. Universal Computer Science*, 2(9): 637–649, 1996.
8. Gravano, L., Papakonstantinou, Y. Mediating and Metasearching on the Internet. *Data Engineering Bulletin 21(2)*, pages 28–36, 1998.
9. 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.
10. Hsu, C.-N., and Chang, C.-C. Finite-State Transducers for Semi-Structured Text Mining. *Proc. IJCAI-99 Workshop on Text Mining*, 1999.
11. 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.
12. JavaCC – The Java parser generator. URL: http://www.metamata.com/JavaCC/.
13. Kushmerick, N., Weld, D.S., and Doorenbos, R., Wrapper Induction for Information Extraction. *Proc. IJCAI-97*: 729–737, 1997.
14. Muslea, I., Minton, S., Knoblock, C. STALKER. *AAAI Workshop on AI & Information Integration*, 1998.
15. Ragetli, H.J.N. Semi-automatic Parser Generation for Information Extraction from the WWW. Master's Thesis, Faculteit WINS, Universiteit van Amsterdam, 1998.
16. Soderland, S. Learning to Extract Text-based Information from the World Wide Web. *Proc. KDD-97*, pages 251–254, 1997.