Hierarchical Wrapper Induction for Semistructured Information Sources

Ion Muslea, Steven Minton, Craig A. Knoblock
Information Sciences Institute and Integrated Media Systems Center
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292-6695
{*muslea, minton, knoblock*}*@isi.edu*

September 10, 1999

Abstract. With the tremendous amount of information that becomes available on the Web on a daily basis, the ability to quickly develop information agents has become a crucial problem. A vital component of any Web-based information agent is a set of wrappers that can extract the relevant data from semistructured information sources. Our novel approach to wrapper induction is based on the idea of hierarchical information extraction, which turns the hard problem of extracting data from an arbitrarily complex document into a series of simpler extraction tasks. We introduce an inductive algorithm, STALKER, that generates high accuracy extraction rules based on user-labeled training examples. Labeling the training data represents the major bottleneck in using wrapper induction techniques, and our experimental results show that STALKER requires up to two orders of magnitude fewer examples than other algorithms. Furthermore, STALKER can wrap information sources that could not be wrapped by existing inductive techniques.

Keywords: wrapper induction, information extraction, supervised inductive learning, information agents

1. Introduction

With the Web, computer users have gained access to a large variety of comprehensive information repositories. However, the Web is based on a browsing paradigm that makes it difficult to retrieve and integrate data from multiple sources. The most recent generation of *information agents* (e.g., WHIRL (Cohen, 1998), Ariadne (Knoblock et al., 1998), and Information Manifold (Kirk et al., 1995)) address this problem by enabling information from pre-specified sets of Web sites to be accessed via database-like queries. For instance, consider the query "What seafood restaurants in L.A. have prices below $20 and accept the Visa credit-card?" Assume that we have two information sources that provide information about LA restaurants: the Zagat Guide and LA Weekly (see Figure 1). To answer this query, an agent could use Zagat's to identify **seafood** restaurants **under $20** and then use LA Weekly to check which of these accept **Visa**.

© 1999 *Kluwer Academic Publishers. Printed in the Netherlands.*

Information agents generally rely on *wrappers* to extract information from *semistructured* Web pages (a document is semistructured if the location of the relevant information can be described based on a concise, formal grammar). Each wrapper consists of a set of extraction rules and the code required to apply those rules. Some systems, such as TSIMMIS (Chawathe et al., 1994) and ARANEUS (Atzeni et al., 1997) depend on humans to write the necessary grammar rules. However, there are several reasons why this is undesirable. Writing extraction rules is tedious, time consuming and requires a high level of expertise. These difficulties are multiplied when an application domain involves a large number of existing sources or the format of the source documents changes over time.

In this paper, we introduce a new machine learning method for wrapper construction that enables unsophisticated users to painlessly turn Web pages into relational information sources. The next section presents a formalism describing semistructured Web documents, and then Sections 3 and 4 present a domain-independent information extractor that we use as a skeleton for all our wrappers. Section 5 describes STALKER, a supervised learning algorithm for inducing extraction rules, and Section 6 presents a detailed example. The final sections describe our experimental results, related work and conclusions.

2. Describing the Content of a Page

Because Web pages are intended to be human readable, there are some common conventions for structuring HTML documents. For instance, the information on a page often exhibits some hierarchical structure; furthermore, semistructured information is often presented in the form of lists of tuples, with explicit separators used to distinguish the different elements. With these observations in mind, we developed the *embedded catalog* ($\mathcal{EC}$) formalism, which can describe the structure of a wide-range of semistructured documents.

The $\mathcal{EC}$ description of a page is a tree-like structure in which the leaves are the items of interest for the user (i.e., they represent the relevant data). The internal nodes of the $\mathcal{EC}$ tree represent *lists* of k-tuples (e.g., lists of restaurant descriptions), where each item in the k-tuple can be either a leaf l or another list L (in which case L is called an embedded list). For instance, Figure 2 displays the $\mathcal{EC}$ descriptions of the LA-Weekly and Zagat pages. At the top level, an LA-Weekly page is a *list* of 5-tuples that contain the `name, address, phone, review,` and an *embedded list* of `credit cards`. Similarly, a Zagat document can be seen as a 7-tuple that includes a list of addresses,

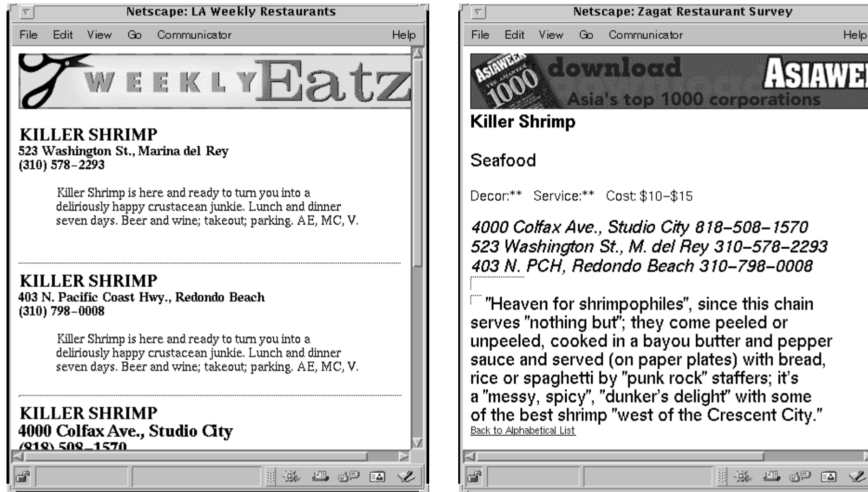

Figure 1. **LA-Weekly and Zagat's Restaurant Descriptions**

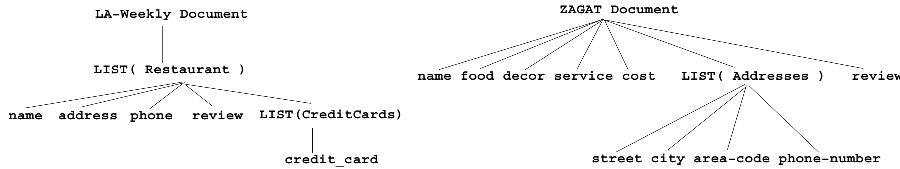

Figure 2. **$\mathcal{EC}$ description of LA-Weekly and ZAGAT pages.**

where each individual address is a 4-tuple `street, city, area-code,` and `phone-number`.

3. Extracting Data from a Document

In order to extract the items of interest, a wrapper uses the $\mathcal{EC}$ description of the document and a set of extraction rules. For each node in the tree, the wrapper needs a rule that extracts that particular node from its parent. Additionally, for each list node, the wrapper requires a *list iteration rule* that decomposes the list into individual tuples. Given the $\mathcal{EC}$ tree and the rules, any item can be extracted by simply determining the path P from the root to the corresponding node and by successively extracting each node in P from its parent. If the parent of a node x is a list, the wrapper applies first the list iteration rule and then it applies x's extraction rule to each extracted tuple.

In our framework a document is a sequence of tokens S (e.g., words, numbers, HTML tags, etc). It follows that the content of the root node in the $\mathcal{EC}$ tree is the whole sequence S, while the content of each of

```
1:     <p> Name: <b> Yala </b><p> Cuisine: Thai <p><i>
2:     4000 Colfax, Phoenix, AZ 85258 (602) 508-1570
3:     </i> <br> <i>
4:     523 Vernon, Las Vegas, NV 89104 (702) 578-2293
5:     </i> <br> <i>
6:     403 Pico, LA, CA 90007 (213) 798-0008
7:     </i>
```

Figure 3. **A simplified version of a Zagat document.**

its children is a subsequence of S. More generally, the content of an arbitrary node x represents a subsequence of the content of its parent p. A key idea underlying our work is that the extraction rules can be based on "landmarks" (i.e., groups of consecutive tokens) that enable a wrapper to locate the content of x within the content of p.

For instance, let us consider the restaurant descriptions presented in Figure 3. In order to identify the beginning of the restaurant name, we can use the rule

R1 = *SkipTo*(`<b>`)

which has the following meaning: start from the beginning of the document and skip everything until you find the `<b>` landmark. More formally, the rule **R1** is applied to the content of the node's parent, which in this particular case is the whole document; the effect of applying **R1** consists of consuming the *prefix of the parent*, which ends at the beginning of the restaurant name. Similarly, one can identify the end of a node's content by applying a rule that consumes the corresponding suffix of the parent. For instance, in order to find the end of the restaurant name, one can apply the rule

R2 = *SkipTo*(`</b>`)

from the end of the document towards its beginning.

The rules **R1** and **R2** are called *start* and *end rules*, and, in most of the cases, they are not unique. For instance, instead of **R1** we can use

R3 = *SkipTo*(`Name`) *SkipTo*(`<b>`)

or

R4 = *SkipTo*(`Name` *Punctuation HtmlTag*)

R3 has the meaning "ignore everything until you find a `Name` landmark, and then, again, ignore everything until you find `<b>`", while **R4** is interpreted as "ignore all tokens until you find a 3-token landmark that consists of the token `Name`, immediately followed by a punctuation symbol and an HTML tag." As the rules above successfully identify the start of the restaurant name, we say that they *match correctly*. By contrast, the start rules $SkipTo$(`:`) and $SkipTo$(`<i>`) are said to *match incorrectly* because they consume too few or too many tokens, respectively (in STALKER terminology, the former is an *early match*, while the later is a *late match*). Finally, a rule like $SkipTo$(`<table>`) *fails* because the landmark `<table>` does not exist in the document.

To deal with variations in the format of the documents, our extraction rules allow the use of disjunctions. For example, if the names of the recommended restaurants appear in bold, while the other ones are displayed as italic, one can extract all the names based on the disjunctive start and end rules

either $SkipTo$(`<b>`)

or $SkipTo$(`<i>`)

and

either $SkipTo$(`</b>`)

or $SkipTo$(`Cuisine`)$SkipTo$(`</i>`)

Disjunctive rules, which represent a special type of decision lists (Rivest, 1987), are *ordered lists* of individual disjuncts. Applying a disjunctive rule is a straightforward process: the wrapper succesively applies each disjunct in the list until it finds the first one that matches (see more details in the next section's footnote).

To illustrate how the extraction process works for list members, consider the case where the wrapper has to extract all the area codes from the sample document in Figure 3 . In this case, the wrapper starts by extracting the entire list of addresses, which can be done based on the start rule $SkipTo$(`<p><i>`) and the end rule $SkipTo$(`</i>`). Then the wrapper has to iterate through the content of list of addresses (lines `2-6` in Figure 3) and to break it into individual tuples. In order to find the start of each individual address, the wrapper starts from the first token in the parent and repeatedly applies $SkipTo$(`<i>`) to the content of the list (each successive rule-matching starts at the point where the previous one ended). Similarly, the wrapper determines the end of each `Address` tuple by starting from the last token in the parent and repeatedly applying the end rule $SkipTo$(`</i>`). In our example, the

list iteration process leads to the creation of three individual addresses that have the contents shown on the lines `2, 4,` and `6`, respectively. Then the wrapper applies to each address the `area-code` start and end rule (e.g., *SkipTo*('(') and *SkipTo*(')'), respectively).

Now let us assume that instead of the area codes, the wrapper has to extract the ZIP Codes. The list extraction and the list iteration remain unchanged, but the ZIP Code extraction is more difficult because there is no landmark that separates the state from the ZIP Code. Even though in such situations the *SkipTo*() rules are not sufficiently expressive, they can be easily extended to a more powerful extraction language. For instance, we can use

R5 = *SkipTo*(**,**) *SkipUntil*(*Number*)

to extract the ZIP Code from the entire address. The argument of *SkipUntil*() describes *a prefix of the content of the item to be extracted*, and it is not consumed when the rule is applied (i.e., the rule stops *immediately before* its occurrence). The rule **R5** means "ignore all tokens until you find the landmark '**,**', and then ignore everything until you find, but do not consume, a number". Rules like **R5** are extremely useful in practice, and they represent only variations of our *SkipTo*() rules (i.e., the last landmark has a special meaning). In order to keep the presentation simple, the rest of the paper focuses mainly on *SkipTo*() rules. When necessary, we will explain the way in which we handle the *SkipUntil*() construct.

The extraction rules presented in this section have two main advantages. First of all, the *hierarchical* extraction based on the $\mathcal{EC}$ tree allows us to wrap information sources that have arbitrary many levels of embedded data. Second, as each node is extracted independently of its siblings, our approach does not rely on there being a fixed ordering of the items, and we can easily handle extraction tasks from documents that may have missing items or items that appear in various orders. Consequently, in the context of using an inductive algorithm that generates the extraction rules, our approach turns an extremely hard problem into several simpler ones: rather then finding a single extraction rule that takes into account all possible item orderings and becomes more complex as the depth of the $\mathcal{EC}$ tree increases, we create several simpler rules that deal with the easier task of extracting each item from its $\mathcal{EC}$ tree parent.

4. Extraction Rules as Finite Automata

We now introduce two key concepts that can be used to define extraction rules: *landmarks* and *landmark automata*. In the rules described in the previous section, each argument of a $SkipTo()$ function is a *landmark*, while a group of $SkipTo()$ functions that must be applied in a pre-established order represents a landmark automaton. In our framework, a *landmark* is a sequence of tokens and wildcards (a wildcard represents a class of tokens, as illustrated in the previous section, where we used wildcards like $Number$ and $HtmlTag$). Such landmarks are interesting for two reasons: on one hand, they are sufficiently expressive to allow efficient navigation within the $\mathcal{EC}$ structure of the documents, and, on the other hand, as we will see in the next section, there is a simple way to generate and refine them.

Landmark automata ($\mathcal{LA}$s) are nondeterministic finite automata in which each transition $S_i \rightarrow S_j$ $(i \neq j)$ is labeled by a landmark $l_{i,j}$; that is, the transition

$$S_i \stackrel{l_{i,j}}{\rightarrow} S_j$$

takes place if the automaton is in the state S_i and the landmark $l_{i,j}$ matches the sequence of tokens at the input. *Linear landmark automata* are a class of $\mathcal{LA}$s that have the following properties:

- a linear $\mathcal{LA}$ has a single accepting state;

- from each non-accepting state, there are exactly two possible transitions: a loop to itself, and a transition to the next state;

- each non-looping transition is labeled by a landmarks;

- all looping transitions have the meaning "consume all tokens until you encounter the landmark that leads to the next state".

The extraction rules presented in the previous section are *ordered lists* of linear $\mathcal{LA}$s. In order to apply such a rule to a given sequence of tokens S, we apply the linear $\mathcal{LA}$s to S in the order in which they appear in the list. As soon as we find an $\mathcal{LA}$ that matches within S, we stop the matching process[1].

[1] Disjunctive *iteration rules* are applied in a slightly different manner. As we already said, iteration rules are applied *repeatedly* on the content of the whole list. Consequently, by blindly selecting the first matching disjunct, there is a risk of skipping over several tuples until we find the first tuple that can be extracted based on that particular disjunct! In order to avoid such problems, a wrapper that uses a disjunctive iteration rule R applies the first disjunct D in R that fulfills the

E1:	513 Pico, Venice, Phone: 1-[800]-555-1515
E2:	90 Colfax, Palms , Phone: ([818]) 508-1570
E3:	523 1st St., LA , Phone: 1-[888]-578-2293
E4:	403 La Tijera, Watts , Phone: ([310]) 798-0008

Figure 4. **Four examples of restaurant addresses.**

In the next section we present the STALKER inductive algorithm that generates rules that identify the start and end of an item x within its parent p. Note that finding a *start rule* that consumes the prefix of p with respect to x (for short $Prefix_x(p)$) is similar to finding an *end rule* that consumes the suffix of p with respect to x (i.e., $Suffix_x(p)$); in fact, the only difference between the two types of rules consists of *how* they are actually applied: the former starts by consuming the first token in p and goes towards the last one, while the later starts at the last token in p and goes towards the first one. Consequently, without any loss of generality, in the rest of this paper we discuss only the way in which STALKER generates start rules.

5. Learning Extraction Rules

The input to STALKER consists of sequences of tokens representing the prefixes that must be consumed by the induced rule. To create such training examples, the user has to select a few sample pages and to use a graphical user interface (GUI) to mark up the relevant data (i.e., the leaves of the $\mathcal{EC}$ tree); once a page is marked up, the GUI generates the sequences of tokens that represent the content of the parent p, together with the index of the token that represents the start of x and uniquely identifies the prefix to be consumed.

Before describing our rule induction algorithm, we will present an illustrative example. Let us assume that the user marked the four area codes from Figure 4 and invokes STALKER on the corresponding four training examples (that is, the prefixes of the addresses $E1$, $E2$, $E3$, and $E4$ that end immediately before the area code). STALKER, which is a sequential covering algorithm, begins by generating a *linear* $\mathcal{LA}$

---

following two criteria. First, D matches within the content of the list. Second, any two disjuncts D_1 and D_2 in R that are applied in succession either fail to match, or match later than D (i.e., one can not generate more tuples by using a combination of two or more other disjuncts).

(remember that each such $\mathcal{LA}$ represents a disjunct in the final rule) that covers as many as possible of the four positive examples. Then it tries to create another *linear* $\mathcal{LA}$ for the remaining examples, and so on. Once STALKER covers all examples, it returns the disjunction of all the induced $\mathcal{LA}$s. In our example, the algorithm generates first the rule **D1** ::= *SkipTo*(`(`), which has two important properties:

- it accepts the positive examples in $E2$ and $E4$;

- it rejects both $E1$ and $E3$ because **D1** can not be matched on them.

During a second iteration, the algorithm considers only the uncovered examples $E1$ and $E3$, based on which it generates the rule

D2 ::= $SkipTo$(`- <b>`).

As there are no other uncovered examples, STALKER returns the *disjunctive rule* **either D1 or D2**.

To generate a rule that extracts an item x from its parent p, STALKER invokes the function **LearnRule()** (see Figure 5). This function takes as input a list of pairs (T_i, Idx_i), where each sequence of tokens T_i is the content of an instance of p, and $T_i[Idx_i]$ is the token that represents the start of x within p. Any sequence $S ::= T_i[1], T_i[2], \ldots, T_i[Idx_i - 1]$ (i.e., any instance of $Prefix_x(p)$) represents a positive example, while any other sub-sequence or super-sequence of S represents a negative example. STALKER tries to generate a rule that accepts all positive examples and rejects all negative ones.

STALKER is a typical *sequential covering* algorithm: as long as there are some uncovered positive examples, it tries to learn a *perfect disjunct* (i.e., a linear $\mathcal{LA}$ that accepts only true positives). When all the positive examples are covered, STALKER returns the solution, which consists of an *ordered* list of all learned disjuncts. The ordering is performed by the function **OrderDisjuncts()** and is based on a straightforward heuristic: the disjuncts with fewer early and late matches should appear earlier; in case of a tie, the disjuncts with more correct matches are preferred to the other ones.

The function **LearnDisjunct()** is a *greedy algorithm* for learning perfect disjuncts: it generates an initial set of *candidates* and repeatedly selects and refines the *best refining candidate* until it either finds a *perfect disjunct*, or runs out of candidates. Before returning a learned disjunct, STALKER invokes **PostProcess()**, which tries to improve the *quality* of the rule (i.e., it tries to reduce the chance that the disjunct will match a random sequence of tokens). This step is necessary because during the refining process each disjunct is kept as general as possible in order to potentially cover a maximal number of examples; once the

LearnRule(*Examples*)
- let *RetVal* be an empty rule
- WHILE $Examples \neq \emptyset$
  - *aDisjunct* =**LearnDisjunct**(*Examples*)
  - remove all examples covered by *aDisjunct*
  - add *aDisjunct* to *RetVal*
- return **OrderDisjuncts**(*RetVal*)

LearnDisjunct(*Examples*)
- let $Seed \in Examples$ be the shortest example
- *Candidates* = **GetInitialCandidates**(*Seed*)
- DO
  - *BestRefiner* = **GetBestRefiner**(*Candidates*)
  - *BestSolution* = **GetBestSolution**($Candidates \cup \{BestSolution\}$)
  - *Candidates* = **Refine**(*BestRefiner*, *Seed*)

  WHILE **IsNotPerfect**(*BestSolution*) AND $BestRefiner \neq \emptyset$
- return **PostProcess**(*BestSolution*)

Refine($C, Seed$)
- let C consist on the consecutive landmarks $l_1, l_2, \cdots, l_n$
- let $TopologyRefs = LandmarkRefs = \emptyset$
- FOR $i = 1$ TO n DO
  - let m be the number of tokens in l_i

  - FOR EACH token t in *Seed* DO
    - in a copy Q of C, add the 1-token landmark t between l_i and l_{i+1}
    - create one such rule for each wildcard that matches t
    - add all these new rules to *TopologyRefs*

  - FOR EACH sequence $s = \overline{t_0 t_1 \cdots t_{m+1}}$ in *Seed* DO
    - IF **Matches**(l_i, s) THEN
      - let $P = Q = C$
      - in P, replace l_i by $\overline{t_0 l_i}$
      - in Q, replace l_i by $\overline{l_i t_{m+1}}$
      - create similar rules for each wildcard that matches t_0 and t_{m+1}
      - add both P and Q to *LandmarkRefs*
- return $TopologyRefs \cup LandmarkRefs$

Figure 5. **The STALKER algorithm.**

refining ends, we post-process the disjunct in order to minimize its potential interference with other disjuncts[2].

Both the initial candidates and their refined versions are generated based on a *seed example*, which is the shortest uncovered example (i.e., the example with the smallest number of tokens in $Prefix_x(p)$). For each token t that ends the seed example and for each wildcard w_i that "matches" t, STALKER creates an initial candidate that is a 2-state $\mathcal{LA}$. In each such automaton, the transition $S_0 \rightarrow S_1$ is labeled by a landmark that is either t or one of the wildcards w_i. The rationale behind this choice is straightforward: as disjuncts have to completely consume each positive example, it follows that any disjunct that consumes a t-ended prefix must end with a landmark that consumes the trailing t.

Before describing the actual refining process, let us present the main intuition behind it. If we reconsider now the four training examples in Figure 4, we see that STALKER starts with the initial candidate *SkipTo*(( ), which is a perfect disjunct; consequently, STALKER removes the covered examples ($E2$ and $E4$) and generates the new initial candidate **R0**::=*SkipTo*(`<b>`). Note that **R0** matches *early* in both uncovered examples $E1$ and $E3$ (that is, it does not consume the whole $Prefix_x(p)$), and, even worse, it also matches within the two already covered examples! In order to obtain a better disjunct, STALKER refines **R0** by adding more terminals to it. During the refining process, we search for new candidates that consume more tokens from the prefixes of the uncovered examples and fail on all other examples. By adding more terminals to a candidate, we hope that its refined versions will eventually turn the early matches into correct ones, while the late matches[3], together with the ones on the already covered examples, will become failed matches. This is exactly what happens when we refine **R0** into **R2** ::= *SkipTo*(`- <b>`): the new rule does not match anymore on $E2$ and $E4$, and **R0**'s early matches on $E1$ and $E3$ become correct matches for **R2**.

[2] We perform three types of post processing operations: replacing wildcards with tokens, merging landmarks that match immediately after each other, and adding more tokens to the short landmarks (e.g., *SkipTo*(`<b>`) is likely to match in most HTML documents, while *SkipTo*(`Maritime Claims : <b>`) matches in significantly fewer). The last operation has a marginal influence because it improves the accuracies of only three of the rules discussed in Section 7.

[3] As explained in Section 3, a disjunct D that consumes more tokens than $Prefix_x(p)$ is called a late match on p. It is easy to see that by adding more terminals to D we can not turn it into an early or a correct match (any refined version of D is guaranteed to consume at least as many tokens as D itself). Consequently, the only hope to avoid an incorrect match of D on p is to keep adding terminals until it fails to match on p.

The **Refine()** function in Figure 5 tries to obtain (potentially) better disjuncts either by making its landmarks more specific (*landmark refinements*), or by adding new states in the automaton (*topology refinements*). In order to perform a refinement, STALKER uses a *refining terminal*, which can be either a token or a wildcard (besides the nine predefined wildcards *Anything, Numeric, AlphaNumeric, Alphabetic, Capitalized, AllCaps, HtmlTag, NonHtml,* and *Punctuation*, STALKER can also use domain specific wildcards that are defined by the user). A straightforward way to generate the refining terminals consists of using all the tokens in the *seed example*, together with the wildcards that match them.[4].

Given a disjunct D, a landmark l from D, and a refining terminal t, a *landmark refinement* makes l more specific by concatenating t either at the beginning or at the end of l. By contrast, a *topology refinement* adds a new state S and leaves the existing landmarks unchanged. For instance, if D has a transition

$$A \xrightarrow{l} B$$

(i.e., the transition from A to B is labeled by the landmark l), then given a refining terminal t, a *topology refinement* creates a new disjunct in which the transition above is replaced by

$$A \xrightarrow{t} S \xrightarrow{l} B.$$

As one might have noted already, **LearnDisjunct()** uses different heuristics for selecting the best refining candidate and the best current solution, respectively. This fact has a straightforward explanation: as long as we try to further refine a candidate, we do not care how well it performs the extraction task. In most of the cases, a good refining candidate *matches early* on as many as possible of the uncovered examples; once a refining candidate extracts correctly from some of the training examples, any further refinements are used mainly to make it fail on the examples on which it still matches incorrectly.

Both sets of heuristics are described in Figure 6. As we already said, **GetBestRefiner()** prefers candidates with a larger potential coverage (i.e., as many as possible early and correct matches). At equal coverage, it prefers a candidate with more early matches because, at the intuitive level, we prefer the most "regular" features in a document: a candidate

[4] In the current implementation, STALKER uses a more efficient approach: for the refinement of a landmark l, we use only the tokens from the seed example that are located *after* the point where l currently matches within the seed example.

BestRefiner()	**BestSolution()**
Prefer candidates that have:	Prefer candidates that have:
- larger coverage	- more correct matches
- more early matches	- more failures to match
- more failed matches	- fewer tokens in *SkipUntil*()
- fewer wildcards	- fewer wildcards
- shorter unconsumed prefixes	- longer end-landmarks
- fewer tokens in *SkipUntil*()	- shorter unconsumed prefixes
- longer end-landmarks	

Figure 6. **The STALKER heuristics.**

that has only early matches is based on a regularity shared by *all* examples, while a candidate that also has some correct matches creates a dichotomy between the examples on which the existing landmarks work perfectly and the other ones. In case of a tie, STALKER selects the disjunct with more failed matches because the alternative would be late matches, which will have to be eventually turned into failed matches by further refinements. All things being equal, we prefer candidates that have fewer wildcards (a wildcard is more likely than a token to match by pure chance), fewer unconsumed tokens in the covered prefixes (after all, the main goal is to fully consume each prefix), and fewer tokens from the content of the slot to be extracted (the main assumption in wrapper induction is that all documents share the same underlying structure; consequently, we prefer extraction rules based on the document template to the ones that rely on the structure of a particular slot). Finally, the last heuristic consists of selecting the candidate that has longer landmarks closer to the item to be extracted; that is, we prefer more specific "local context" landmarks.

In order to pick the best current solution, STALKER uses a different set of criteria. Obviously, it starts by selecting the candidate with the most correct matches. If there are several such disjuncts, it prefers the one that fails to match on most of the remaining examples (remember that the alternatives, early or late matches, represent incorrect matches!). In case of a tie, for reasons similar to the ones cited above, we prefer candidates that have fewer tokens from the content of the item, fewer wildcards, longer landmarks closer to the item's content, and fewer unconsumed tokens in the covered prefixes (i.e., in case of incorrect match, the result of the extraction contains fewer irrelevant tokens).

Finally, STALKER can be easily extended such that it also uses *SkipUntil*() constructs. The rule refining process remains unchanged (after all, *SkipUntil*() changes only the meaning of the last landmark

in a disjunct), and the only modification involves **GenerateInitial-Candidates()**. More precisely, for each terminal t that matches the first token in an instance of x (including the token itself), STALKER also generates the initial candidates $SkipUntil(t)$.

6. Example of Rule Induction

Let us consider again the restaurant addresses from Figure 4. In order to generate an extraction rule for the `area-code`, we invoke STALKER with the training examples $\{E1, E2, E3, E4\}$. During the first iteration, **LearnDisjunct()** selects the shortest prefix, $E2$, as seed example. The last token to be consumed in $E2$ is "`(`", and there are two wildcards that match it: *Punctuation* and *Anything*; consequently, STALKER creates three initial candidates:

R1 = $SkipTo$(`(`)

R2 = $SkipTo$(*Punctuation*)

R3 = $SkipTo$(*Anything*)

As **R1** is a *perfect disjunct*[5], **LearnDisjunct()** returns **R1** and the first iteration ends.

During the second iteration, **LearnDisjunct()** is invoked with the uncovered training examples $\{E1, E3\}$; the new seed example is $E1$, and STALKER creates again three initial candidates:

R4 = $SkipTo$(`<b>`)

R5 = $SkipTo$(*HtmlTag*)

R6 = $SkipTo$(*Anything*)

As all three initial candidates match early in all uncovered examples, STALKER selects **R4** as the best possible refiner because it uses no wildcards in the landmark. By refining **R4**, we obtain the three landmark refinements

R7 = $SkipTo$(`- <b>`)

R8 = $SkipTo$(*Punctuation* `<b>`)

R9 = $SkipTo$(*Anything* `<b>`)

R10: *SkipTo*(Venice) *SkipTo*()
R11: *SkipTo*() *SkipTo*(<b>)
R12: *SkipTo*(:) *SkipTo*(<b>)
R13: *SkipTo*(-) *SkipTo*(<b>)
R14: *SkipTo*(,) *SkipTo*(<b>)
R15: *SkipTo*(Phone) *SkipTo*(<b>)
R16: *SkipTo*(1) *SkipTo*(<b>)
R17: *SkipTo*(*Numeric*) *SkipTo*(<b>)
R18: *SkipTo*(*Punctuation*)*SkipTo*(<b>)
R19: *SkipTo*(*HtmlTag*) *SkipTo*(<b>)
R20: *SkipTo*(*AlphaNum*) *SkipTo*(<b>)
R21: *SkipTo*(*Alphabetic*) *SkipTo*(<b>)
R22: *SkipTo*(*Capitalized*) *SkipTo*(<b>)
R23: *SkipTo*(*NonHtml*) *SkipTo*(<b>)
R24: *SkipTo*(*Anything*) *SkipTo*(<b>)

Figure 7. **All 21 topology refinements of R4.**

along with the 21 topology refinements shown in Figure 7.

At this stage, we have already generated several *perfect disjuncts*: **R7**, **R11**, **R12**, **R13**, **R15**, **R16**, and **R19**. They all match correctly on $E1$ and $E3$, and fail to match on $E2$ and $E4$; however, STALKER dismisses **R19** because it is the only one using wildcards in its landmarks. Of the remaining six candidates, **R7** represents the best solution because it has the longest end landmark (all other disjuncts end with a 1-token landmark). Consequently, **LearnDisjunct()** returns **R7**, and because there are no more uncovered examples, STALKER completes its execution by returning the disjunctive rule **either R1 or R7**.

7. Experimental Results

In order to evaluate STALKER's capabilities, we tested it on the 30 information sources that were used as application domains by WIEN (Kushmerick, 1997), which was the first wrapper induction system[6]. To make the comparison between the two systems as fair as possible, we did not use any domain specific wildcards, and we tried to follow the exact experimental conditions used by Kushmerick. For all 21 sources for which WIEN had labeled examples, we used the exact same data; for the remaining 9 sources, we worked closely with Kushmerick to reproduce the original WIEN extraction tasks. Furthermore, we also used WIEN's experimental setup: we start with one randomly chosen training example, learn an extraction rule, and test it against all the unseen examples. We repeated these steps 30 times, and we average the number of test examples that are correctly extracted. Then we

[5] Remember that a *perfect disjunct* correctly matches at least one example (e.g., $E2$ and $E4$) and rejects all other ones.

[6] All these collections of sample documents, together with a detailed description of each extraction task, can be obtained from the RISE repository, which is located at `http://www.isi.edu/~muslea/RISE/index.html`.

repeated the same procedure with 2, 3, ..., and 10 training examples. As opposed to WIEN, we do not train on more than 10 examples because we noticed that, in practice, a user rarely has the patience of labeling more than 10 training examples.

This section has four distinct parts. We begin with an overview of the performance of STALKER and WIEN over the 30 test domains, and we continue with an analysis of STALKER's ability to learn list extraction and iteration rules, which are key components in our approach to hierarchical wrapper induction. Then we compare and contrast STALKER and WIEN based on the number of examples required to wrap the sources, and we conclude with the main lessons drawn from this empirical evaluation.

7.1. Overall Comparison of STALKER and WIEN

The data in Table I provides an overview of the two systems' performance over the 30 sources. The first four columns contain the source name, whether or not the source has missing items or items that may appear in various orders, and the number of embedded lists in the $\mathcal{EC}$ tree. The next two columns specify how well the two systems performed: whether they wrapped the source perfectly, imperfectly, or completely failed to wrap it. For the time being, let us ignore the last two columns in the table.

In order to better understand the data from Table I, we have to briefly describe the type of wrappers that WIEN generates (a more technical discussion is provided in the next section). WIEN uses a fairly simple extraction language: it does not allow the use of wildcards and disjunctive rules, and the items in each k-tuple are assumed to be always present and to always appear in the same order. Based on the assumptions above, WIEN learns a unique *multi-slot* extraction rule that extracts all the items in a k-tuple at the same time (by contrast, STALKER generates several *single-slot* rules that extract each item independently of its siblings in the k-tuple). For instance, in order to extract all the addresses and area codes from the document in Figure 3, a hypothetical WIEN rule does the following: it ignores all characters until it finds the string "`<p><i>`" and extracts as `Address` everything until it encounters a "`(`". Then it immediately starts extracting the `AreaCode`, which ends at "`)`". After extracting such a 2-tuple, the rule is applied again until it does not match anymore.

Out of the 30 sources, WIEN wraps *perfectly* 18 of them, and *completely fails* on the remaining 12. These complete failures have a straightforward explanation: if there is no perfect wrapper in WIEN's language (because, say, there are some missing items), the inductive algorithm

Table I. **Test domains for WIEN and STALKER: a dash denotes failure, while $\surd$ and $\simeq$ mean *perfectly* and *imperfectly* wrapped, respectively.**

SRC	Miss	Perm	Embd	WIEN	STALKER		
						ListExtr	ListIter
S1	-	-	-	$\surd$	$\simeq$	1 / 100%	1 / 100%
S2	yes	-	-	-	$\simeq$	1 / 100%	1 / 100%
S3	-	-	-	$\surd$	$\simeq$	1 / 100%	5 / 100%
S4	-	-	-	$\surd$	$\surd$	7 / 100%	1 / 100%
S5	-	-	-	$\surd$	$\surd$	1 / 100%	1 / 100%
S6	yes	-	-	-	$\simeq$	1 / 100%	8 / 100%
S7	yes	-	-	-	$\surd$	6 & 1 / 100%	2 & 7/ 100%
S8	-	-	-	$\surd$	$\surd$	1 / 100%	1 / 100%
S9	yes	-	1	-	$\simeq$	10 / 100%	10 / 96%
						10 / 94%	7 / 100%
S10	-	-	-	$\surd$	$\surd$	1 / 100%	1 / 100%
S11	yes	yes	-	-	$\simeq$	1 / 100%	3 / 100%
S12	-	-	-	$\surd$	$\surd$	1 / 100%	1 / 100%
S13	-	-	-	$\surd$	$\surd$	1 / 100%	1 / 100%
S14	-	-	-	$\surd$	$\surd$	1 / 100%	1 / 100%
S15	-	-	-	$\surd$	$\surd$	1 / 100%	1 / 100%
S16	yes	-	-	-	$\surd$	-	-
S17	-	-	1	-	$\surd$	1 & 1 / 100%	4 & 1 / 100%
S18	yes	-	-	-	$\surd$	1 & 4 / 100%	1 & 1 / 100%
S19	-	-	-	$\surd$	$\surd$	1 / 100%	1 / 100%
S20	-	-	-	$\surd$	$\surd$	1 / 100%	1 / 100%
S21	yes	yes	2	-	-	-	-
S22	-	-	-	$\surd$	$\surd$	1 / 100%	1 / 100%
S23	-	-	-	$\surd$	$\surd$	1 / 100%	1 / 100%
S24	yes	-	1	-	$\simeq$	1 / 100%	1 / 100%
						1 / 100%	10 / 91%
S25	-	-	-	$\surd$	$\surd$	1 / 100%	1 / 100%
S26	-	-	1	-	$\simeq$	1 & 1 / 100%	1 & 1 / 100%
S27	-	-	-	$\surd$	$\surd$	-	-
S28	-	-	-	$\surd$	$\surd$	1 / 100%	1 / 100%
S29	yes	yes	-	-	-	-	-
S30	-	-	-	$\surd$	$\surd$	1 / 100%	1 / 100%

does not even try to generate an imperfect rule. It is important to note that WIEN fails to wrap all sources that include *embedded lists* (remember that embedded lists are at least two levels deep) or items that are *missing* or appear in *various orders*.

On the same test domains, STALKER wraps perfectly 20 sources and learns 8 additional imperfect wrappers. Out of these last 8 sources, in 4 cases STALKER generates "high quality" wrappers (i.e., wrappers in which most of the rules are 100% accurate, and no rule has an accuracy below 90%). Finally, two of the 30 sources, **S21** and **S29**, can not be wrapped by STALKER.[7] In order to wrap all 28 sources, STALKER induced 206 different rules, out of which 182 (i.e., more than 88%) had 100% accuracy, and another 18 were at least 90% accurate; in other words, only six rules, which represents 3% of the total, had an accuracy below 90%. Furthermore, as we will see later, the perfect rules were usually induced based on just a couple of training examples.

7.2. Learning List Extraction and Iteration Rules

As opposed to WIEN, which performs an *implicit* list iteration by repeatedly applying the same multi-slot extraction rule, STALKER learns *explicit* list extraction and iteration rules that allow us to navigate within the $\mathcal{EC}$ tree. These types of rules are crucial to our approach because they allow us to decompose a difficult wrapper induction problem into several simpler ones in which we always extract one individual item from its parent. To estimate STALKER's performance, we have to analyze its performance at learning the 32 list extraction and 32 list iteration rules that appeared in the 28 test domains above.

The results are shown in the last two columns of Table I, where we provide the number of training examples and the accuracy for each such rule. Note that there are some sources, like **S16**, that have no lists at all. At the other end of the spectrum, there are several sources that include two lists[8].

[7] The documents in **S21** are difficult to wrap because they include a *heterogeneous list* (i.e., the list contains elements of several types). As each type of element uses a different kind of layout, the iteration task is extremely difficult. The second source, **S29**, raises a different type of problem: some of the items have just a handful of occurrences in the collection of documents, and, furthermore, about half of them represent various types of formatting/semantic errors (e.g., the date appearing in the location of the price slot, and the actual date slot remaining empty). Under these circumstances, we decided to declare this source unwrappable by STALKER.

[8] For sources with multiple lists, we present the data in two different ways. If all the learned rules are perfect, the results appear on the same table line (e.g., for **S7**, the list extraction rules required 6 and 1 examples, respectively, while the list iteration rules required 2 and 7 examples, respectively). If at least one of the rules

The results are extremely encouraging: only one list extraction and two list iteration rules were not learned with a 100% accuracy, and all these imperfect rules have accuracies above 90%. Furthermore, out of the 72 rules, 50 of them were learned based on a *single* training example! As induction based on a single example is quite unusual in machine learning, it deserves a few comments. STALKER learns a perfect rule based on a single example whenever one of the initial candidates is a perfect disjunct. Such situations are frequent in our framework because the hierarchical decomposition of the problem makes most of the subproblems (i.e., the induction of the individual rules) straightforward. In final analysis, we can say that independently of how difficult it is to induce all the extraction rules for a particular source, the list extraction and iteration rules can be usually learned with a 100% accuracy based on just a few examples.

7.3. Efficiency Issues

In order to easily compare WIEN's and STALKER's requirements in terms of the number of training examples, we divided the sources above in three main groups:

- sources that can be perfectly wrapped by both systems (Table II)
- sources that can be wrapped perfectly only by one system (Tables III and IV)
- sources on which WIEN fails completely, while STALKER generates imperfect wrappers (Table V).

For each source that WIEN can wrap (see Tables II and IV), we provide two pieces of information: the number of *training pages* required by WIEN to generate a correct wrapper, and the total number of item occurrences that appear in those pages. The former is taken from (Kushmerick, 1997) and represents the *smallest* number of completely labeled training pages required by one of the six wrapper classes that can be generated by WIEN. The latter was obtained by multiplying the number above by the average number of item occurrences *per page*, computed over all available documents.

For each source that STALKER wrapped perfectly, we report four pieces of informations: the minimum, maximum, mean, and median number of training examples (i.e., item occurrences) that were required

has an accuracy below 100%, the data for the different lists appear on successive lines (see, for instance, the source **S9**).

Table II. **Sources Wrapped Perfectly by Both Systems.**

SRC	WIEN		STALKER (number of examples)			
	Docs	Exs	Min	Max	Mean	Median
S4	2.0	20.0	1.0	7.0	2.0	1.0
S5	2.0	14.4	1.0	3.0	1.5	1.0
S8	2.0	43.6	1.0	2.0	1.2	1.0
S10	3.9	39.0	1.0	2.0	1.2	1.0
S12	2.0	88.8	1.0	1.0	1.0	1.0
S13	2.0	20.0	1.0	3.0	1.5	1.0
S14	7.0	679.7	1.0	2.0	1.4	1.0
S15	2.0	70.8	1.0	1.0	1.0	1.0
S19	2.0	40.0	1.0	1.0	1.0	1.0
S20	2.0	65.4	1.0	1.0	1.0	1.0
S22	2.0	200.0	1.0	1.0	1.0	1.0
S23	2.0	109.0	1.0	7.0	1.6	1.0
S25	2.0	65.4	1.0	1.0	1.0	1.0
S27	2.0	2.0	1.0	4.0	2.2	1.5
S28	2.0	150.0	1.0	1.0	1.0	1.0
S30	5.3	15.9	1.0	9.0	2.4	1.0

to generate a correct rule[9]. For the remaining 8 sources from Tables IV and V, we present an individual description for each learned rule by providing the reached accuracy and the required number of training examples.

By analyzing the data from Table II, we can see that for the 16 sources that both systems can wrap correctly, STALKER requires up to two orders of magnitude fewer training examples. STALKER requires no more than 9 examples for any rule in these sources, and for more than half of the rules it can learn perfect rules based on a single example (similar observations can be made for the four sources from Table III).

[9] We present the empirical data for the perfectly wrapped sources in such a compact format because it is more readable than a huge table that provides detailed information for each individual rule. Furthermore, as 19 of the 20 sources from Tables II and III have a median number of training examples equal to one, it follows that more than half of the individual item data would read "item X required *a single training example* to generate a 100% accurate rule."

Table III. **Source on which WIEN fails completely, while STALKER wraps them perfectly.**

SRC	WIEN	STALKER (number of examples)			
		Min	Max	Mean	Median
S7	-	1.0	7.0	2.3	1.0
S16	-	1.0	10.0	3.4	1.0
S17	-	1.0	4.0	1.5	1.0
S18	-	1.0	4.0	1.4	1.0

Table IV. **Sources on which WIEN outperforms STALKER.**

SRC	WIEN		STALKER		
	Docs	Exs	Task	Accuracy	Exs
S1	2.0	80.6	Price	100%	1
			URL	91%	10
			Product	92%	10
			Manufacturer	100%	3
			List Extraction	100%	1
			List Iteration	100%	1
S3	2.0	34.8	URL	100%	8
			Title	100%	1
			Abstract	100%	1
			Size	100%	1
			Date	98%	10
			Time	97%	10
			List Extraction	100%	1
			List Iteration	100%	5

As the main bottleneck in wrapper induction consists of labeling the training data, the advantage of STALKER becomes quite obvious.

Table IV reveals that despite its advantages, STALKER may learn imperfect wrappers for sources that pose no problems to WIEN. The explanation is quite simple and is related to the different ways in which the two systems define a training example: WIEN's examples are *entire documents*, while STALKER uses fragments of pages (each parent of an item

Table V. **Sources on which WIEN fails, and STALKER wraps imperfectly.**

SRC	Task	Accur.	Exs	SRC	Task	Accur.	Exs
S2	URL	100%	1	**S6**	Faculty	100%	4
	Source	99%	10		University	93%	10
	Title	88%	10		Attention	100%	2
	Date	84%	10		Address	100%	2
	ListExtr	100%	1		City	100%	1
	ListIter	100%	1		ZIP	100%	1
S9	Date	100%	1		Province	100%	1
	Price	100%	1		Country	100%	1
	Airline	100%	1		Phone	100%	1
	Flight	100%	1		Fax	96%	10
	DepartCity	100%	1		URL	97%	10
	DepartCode	100%	1		UpdateOn	100%	1
	ArriveCity	99%	10		UpdateBy	100%	1
	ArriveCode	100%	2		ListExtr	100%	1
	DepartTime	100%	3		ListIter	100%	8
	ArriveTime	92%	10	**S11**	Name	94%	10
	Availab.	95%	10		Email	98%	10
	Food	100%	6		Update	66%	10
	Plane	100%	3		Organiz.	48%	10
	ListExtr-1	100%	10		Alt. Name	100%	1
	ListIter-1	94%	10		Provider	100%	1
	ListExtr-2	96%	10		ListExtr	100%	1
	ListIter-2	100%	1		ListIter	100%	3
S24	Language	100%	1	**S26**	House	100%	1
	URL	91%	10		Number	100%	1
	Image	100%	6		Price	97%	10
	Translat.	89%	10		Artist	100%	1
	ListExtr-1	100%	1		Album	71%	1
	ListIter-1	100%	1		ListExtr-1	100%	1
	ListExtr-2	100%	1		ListIter-1	100%	1
	ListIter-2	91%	10		ListExtr-2	99%	10
					ListIter-2	100%	1

is a fragment of a document). This means that for sources in which *each document contains all possible variations* of the main format, WIEN is *guaranteed* to see all possible variations! On the other hand, STALKER has practically no chance of having all these variations in *each* randomly chosen training set. Consequently, whenever STALKER is trained only on a few variations, it will generate an imperfect rule. In fact, the different types of training examples lead to an interesting trade-off: by using only fragments of documents, STALKER may learn perfect rules based on significantly fewer examples than WIEN. On the other hand, there is a risk that STALKER may induce imperfect rules; we plan to fix this problem by using active learning techniques (RayChaudhuri and Hamey, 1997) to identify all possible types of variations.

Finally, in Table V we provide detailed data about the learned rules for the six most difficult sources. Besides the problem mentioned above, which leads to several rules of 99% accuracy, these sources also contain missing items and items that may appear in various orders. Out of the 62 rules learned by STALKER for these six sources, 42 are perfect and another 14 have accuracies above 90%. Sources like **S6** and **S9** emphasize another advantage of the STALKER approach: one can label just a few training examples for the rules that are easier to learn, and than focus on providing additional examples for the more difficult ones.

7.4. Lessons

Based on the results above, we can draw several important conclusions. First of all, compared with WIEN, STALKER has the ability to wrap a larger variety of sources. Even though not all the induced wrappers are perfect, an imperfect, high accuracy wrapper is to be preferred to no wrapper at all.

Second, STALKER is capable of learning most of the extraction rules based on just a couple of examples. This is a crucial feature because from the user's perspective it makes the wrapper induction process both fast and painless. Our hierarchical approach to wrapper induction played a key role at reducing the number of examples: on one hand, we decompose a hard problem into several easier ones, which, in turn, require fewer examples. On the other hand, by extracting the items independently of each other, we can label just a few examples for the items that are easy to extract (as opposed to labeling every single occurrence of each item in each training page).

Third, by using single-slot rules, we do not allow the harder items to affect the accuracy of the ones that are easier to extract. Consequently, even for the most difficult sources, STALKER is typically capable of learning perfect rules for several of the relevant items.

Last but not least, the fact that even for the hardest items STALKER usually learns a correct rule (in most of the cases, the lower accuracies come from averaging correct rules with erroneous ones) means that we can try to improve STALKER's behavior based on active learning techniques that would allow the algorithm to select the few relevant cases that would lead to a correct rule.

8. Related Work

Research on learning extraction rules has occurred mainly in two contexts: creating wrappers for information agents and developing general purpose information extraction systems for natural language text. The former are primarily used for semistructured information sources, and their extraction rules rely heavily on the regularities in the structure of the documents; the latter are applied to free text documents and use extraction patterns that are based on linguistic constraints.

With the increasing interest in accessing Web-based information sources, a significant number of research projects depend on wrappers to retrieve the relevant data. A wide variety of languages have been developed for manually writing wrappers (i.e., where the extraction rules are written by a human expert), from procedural languages (Atzeni and Mecca, 1997) and Perl scripts (Cohen, 1998) to pattern matching (Chawathe et al., 1994) and LL(k) grammars (Chidlovskii et al., 1997). Even though these systems offer fairly expressive extraction languages, the manual wrapper generation is a tedious, time consuming task that requires a high level of expertise; furthermore, the rules have to be rewritten whenever the sources suffer format changes. In order to help the users cope with these difficulties, Ashish and Knoblock (Ashish and Knoblock, 1997) proposed an expert system approach that uses a fixed set of heuristics of the type "look for bold or italicized strings."

The wrapper induction techniques introduced in WIEN (Kushmerick, 1997) are a better fit to frequent format changes because they rely on learning techniques to generate the extraction rules. Compared to the manual wrapper generation, Kushmerick's approach has the advantage of dramatically reducing both the time and the effort required to wrap a source; however, his extraction language is significantly less expressive than the ones provided by the manual approaches. In fact, the WIEN extraction language can be seen as a non-disjunctive STALKER rules that use just a single $SkipTo()$ and do not allow the use of wildcards. There are several other important differences between STALKER and WIEN. First, as WIEN learns the landmarks by searching *common prefixes* at the *character level*, it needs more training examples than STALKER.

Second, WIEN cannot wrap sources in which some items are missing or appear in various orders. Last but not least, STALKER can handle $\mathcal{EC}$ trees of arbitrary depths, while WIEN's approach to nested documents turned out to be impractical: even though Kushmerick was able to manually write 19 perfect "nested" wrappers, none of them could be learned by WIEN.

SoftMealy (Hsu and Dung, 1998) uses a wrapper induction algorithm that generates extraction rules expressed as finite transducers. The SoftMealy rules are more general than the WIEN ones because they use wildcards and they can handle both missing items and items appearing in various orders. Intuitively, SoftMealy's rules are similar to the ones used by STALKER, except that each disjunct is either a single $SkipTo()$ or a $SkipTo()SkipUntil()$ in which the two landmarks must match *immediately* after each other. As SoftMealy uses neither multiple $SkipTo()$s nor multiple $SkipUntil()$s, it follows that its extraction rules are strictly less expressive than STALKER's. Finally, SoftMealy has one additional drawback: in order to deal with missing items and various orderings of items, SoftMealy may have to see training examples that include *each possible ordering* of the items.

In contrast to information agents, most general purpose information extraction systems are focused on unstructured text, and therefore the extraction techniques are based on linguistic constraints. However, there are three such systems that are somewhat related to STALKER: WHISK (Soderland, 1999), Rapier (Califf and Mooney, 1999), and SRV (Freitag, 1998). The extraction rules induced by Rapier and SRV can use the landmarks that immediately precede and/or follow the item to be extracted, while WHISK is capable of using multiple landmarks. But, similarly to STALKER and unlike WHISK, Rapier and SRV extract a particular item independently of the other relevant items. It follows that WHISK has the same drawback as SoftMealy: in order to handle correctly missing items and items that appear in various orders, WHISK must see training examples for each possible ordering of the items. None of these three systems can handle embedded data, though all use powerful linguistic constraints that are beyond STALKER's capabilities.

9. Conclusions and Future Work

The primary contribution of our work is to turn a potentially hard problem – learning extraction rules – into a problem that is extremely easy in practice (i.e., typically very few examples are required). The number of required examples is small because the $\mathcal{EC}$ description of a page simplifies the problem tremendously: as the Web pages are

intended to be human readable, the $\mathcal{EC}$ structure is generally reflected by actual landmarks on the page. STALKER merely has to find the landmarks, which are generally in the close proximity of the items to be extracted. In other words, the extraction rules are typically very small, and, consequently, they are easy to induce.

We plan to continue our work on several directions. First, we plan to use *unsupervised learning* in order to narrow the landmark search-space. Second, we would like to use *active learning* techniques to minimize the amount of labeling that the user has to perform. Third, we plan to provide PAC-like guarantees for STALKER.

Acknowledgments

This work was supported in part by USC's Integrated Media Systems Center (IMSC) - an NSF Engineering Research Center, by the National Science Foundation under grant number IRI-9610014, by the U.S. Air Force under contract number F49620-98-1-0046, by the Defense Logistics Agency, DARPA, and Fort Huachuca under contract number DABT63-96-C-0066, and by research grants from NCR and General Dynamics Information Systems. The views and conclusions contained in this paper are the authors' and should not be interpreted as representing the official opinion or policy of any of the above organizations or any person connected with them.

References

Ashish, N. and C. Knoblock: 1997, 'Semi-automatic wrapper generation for Internet information sources'. In: *Proceedings of Cooperative Information Systems*. pp. 160–169.

Atzeni, P. and G. Mecca: 1997, 'Cut and paste'. In: *Proceedings of the 16th ACM SIGMOD Symposion on Principles of Database Systems*. pp. 144–153.

Atzeni, P., G. Mecca, and P. Merialdo: 1997, 'Semi-structured and structured data in the Web: going back and forth'. In: *Proceedings of ACM SIGMOD Workshop on Management of Semi-structured Data*. pp. 1–9.

Califf, M. and R. Mooney: 1999, 'Relational Learning of Pattern-Match Rules for Information Extraction'. In: *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*. pp. 328–334.

Chawathe, S., H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom.: 1994, 'The TSIMMIS project: integration of heterogeneous information sources'. In: *Proceedings of the 10th Meeting of the Information Processing Society of Japan*. pp. 7–18.

Chidlovskii, B., U. Borghoff, and P. Chevalier: 1997, 'Towards sophisticated wrapping of Web-based information repositories'. In: *Proceedings of 5th International RIAO Conference*. pp. 123–35.

Cohen, W.: 1998, 'A Web-based Information System that Reasons with Structured Collections of Text'. In: *Proceedings of the Second International Conference on Autonomous Agents (AA-98)*. pp. 400–407.

Freitag, D.: 1998, 'Information Extraction From HTML: Application of a General Learning Approach'. In: *Proceedings of the 15th Conference on Artificial Intelligence (AAAI-98)*. pp. 517–523.

Hsu, C. and M. Dung: 1998, 'Generating Finite-State Transducers for Semistructured data extraction from the Web'. *Journal of Information Systems* **23(8)**, 521–538.

Kirk, T., A. Levy, Y. Sagiv, and D. Srivastava: 1995, 'The Information Manifold'. In: *Proceedings of the AAAI Spring Symposium: Information Gathering from Heterogeneous Distributed Environments*. pp. 85–91.

Knoblock, C., S. Minton, J. Ambite, N. Ashish, J. Margulis, J. Modi, I. Muslea, A. Philpot, and S. Tejada: 1998, 'Modeling Web Sources for Information Integration'. In: *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*. pp. 211–218.

Kushmerick, N.: 1997, 'Wrapper induction for information extraction'. Ph.D. thesis, Dept. of Computer Science, U. of Washington, TR UW-CSE-97-11-04.

RayChaudhuri, T. and L. Hamey: 1997, 'Active learning-approaches and issues'. *Journal of Intelligent Systems* **7**, 205–243.

Rivest, R. L.: 1987, 'Learning Decision Lists'. *Machine Learning* **2**(3), 229–246.

Soderland, S.: 1999, 'Learning Information Extraction Rules for Semi-structured and Free Text'. *Machine Learning* **34**(1/2/3), 233–272.