Extracting partial structures from HTML documents

Hiroshi Sakamoto Yoshitsugu Murakami Hiroki Arimura Setsuo Arikawa

Department of Informatics, Kyushu University
Hakozaki 6-10-1, Hizashi-ku, Fukuoka-shi 812-8581, Japan
{hiroshi, arim, arikawa}@i.kyushu-u.ac.jp
phone: +82-92-642-2693, fax: +82-92-642-2698

Abstract

The new wrapper model for extracting text data from HTML documents is introduced. In this model, an HTML file is considered as an ordered labeled tree. The learning algorithm takes the sequence of pairs of an HTML tree and a set of nodes The nodes indicate the labels to extract from the HTML tree. The goal of the learning algorithm is to output the wrapper which exactly extracts the labels from the HTML trees.

keywords: information extraction, wrapper induction, semi-structured data, inductive learning

Introduction

The HTML documents currently distributed on the Internet can be regarded as a very large text database and the *information extraction* from the Web is widely studied. The problem of extracting the texts and attributes from HTML documents is difficult because we can not construct the XML like database by only the limited number of HTML tags.

For this purpose Kushmerick introduced the framework of the *wrapper induction* (Kusshmerick 2000). An HTML document is called a *page* and the contents of the page is called the *label*. The goal of the learning algorithm is, given the sequence of examples $\langle P_n, L_n \rangle$ of pages and labels, to output the program W such that $L_n = W(P_n)$ for all n. Other extracting models, for example, are in (Hammer, Garcia-Molina, Cho, and Crespo 1997; Chun-Nan Hsu 1998; Muslea, Minton, Craig, and Knoblock 1998; Freitag 1998).

The program W is called *Wrapper*. Kushmerick defined several classes of Wrappers, in particular, we explain the LR-Wrapper (Kusshmerick 2000). An LR-Wrapper is a sequence $\langle\langle \ell_1, r_1\rangle, \ldots, \langle \ell_K, r_K\rangle\rangle$, where the ℓ_i is called the *left delimiter* and the r_i is called the *right delimiter* for the *i-th attribute* $i = 1, \ldots, K$. The attribute is the unit of extraction data and we assume that HTML page is constructed by the finite repetitions of K attributes.

First, the LR-Wrapper finds the first appearance i of the ℓ_1 in the page P and finds the first appearance j of the r_1 starting from the i. If such i and j are found, it extracts the string between the i and j as the first attribute in P and it continues to extract the next attribute.

The idea of the learning algorithm for LR-Wrapper is to find the ℓ_i as *the longest common suffix* of the strings just before the i-th attribute and the r_i as *the longest common prefix* of the strings immediately after the i-th attribute. Thus, the string is so safe as to be long. However, in the following case, we can not get a sufficiently long delimiters.

```
<h2><a href="www.arim.com">www.arim.com</a>
<br></h2><h3><a href="mailto:arim@arim.com">
arim@arim.com</a><br></h3>
...
<h2><a href="saka.co.jp">saka.co.jp</a><br>
</h2><h3><a href="mailto:saka@saka.co.jp">
saka@saka.co.jp</a><br></h3>
```

Consider the case of extracting the attributes "arim@arim.com" and "saka@saka.co.jp". The learned left and right delimiters for the attributes are the "> and
</h3>, respectively. Now, let us extract the string "arim@arim.com". The first appearance of the the "> is in the first line and the first appearance of
</h3> from this point is in the third line. Thus the extracted string is the "www.arim.com
</h2><h3> <a href="mailto:arim@arim.com">arim@arim.com". The cause in this case is the HTML attribute values of the <a> tags for the email addresses. Since there is no common suffix of the strings, the LR-Wrapper can not determine the correct delimiters.

Thus, for overcome this difficulty, we propose the new data model for the HTML wrapper called *Tee-Wrapper* over the tree structures and present the learning algorithm of the Tree-Wrappers. Moreover, we experiment the prototype of our learning algorithm for more than 1,000 pages of HTML documents.

The introduced Tree-Wrapper W is the sequence $\langle EP_1, \ldots, EP_K \rangle$. The EP_i, called the *extraction path*, is the expansion of the notion of path to extract the i-th attributes from the HTML trees. Each EP_i is of the form $\langle ENL_{i_1},\ldots,ENL_{i_\ell}\rangle$, where ENL_{i_1} is called the *extraction node label*. The most simple ENL_{i_j} is the one that consists of only the node name. For a given HTML tree, the Tree-Wrapper tries to find a path *matching with* the path $\langle ENL_{i_1},\ldots,ENL_{i_\ell}\rangle$ and if it is found, then the Wrapper extracts the i-th attributes of the last node which matches with the ENL_{i_ℓ}.

Let us explain the Tree-Wrapper for the example of the HTML document in the above. In this case, the most simple Tree-Wrapper is $W = \langle EP_1\rangle$ and $EP_1 = (<\mathbf{h3}>, <\mathbf{a}>)$. This path matches with only the paths for the email addresses. In the next sections, we introduce the data model for such extraction and present the algorithm to learn more powerful Tree-Wrappers. In the finial section, we show the expressiveness of the Tree-Wrapper by the experimental results.

The Data Model

In this section, we define the HTML tree which is constructed from an HTML file. First, we begin with the notations used in this paper. An *alphabet* Σ is a set of finite symbols. A finite sequence $\langle a_1,\ldots,a_n\langle$ of elements in Σ is called *string* and it is denoted by $w = a_i\cdots a_n$ for short. The *empty string* of length zero is ε. The set of all strings is denoted by Σ^* and let $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. For string w, if $w = \alpha\beta\gamma$, then the string α (β)is called a *prefix* (*suffix*) of w, respectively.

For each tree T, the set of all nodes of T is a subset of $I\!N = \{0,\ldots,n\}$ of natural numbers, where the 0 is the root. A node is called a *leaf* if it has no child and any other node is called an *internal node*. If $n, m \in I\!N$ has the same parent, then n and m are *brother* and n is a *big brother* of m if $n \leq m$. The sequence $\langle n_1,\ldots,n_k\langle$ of nodes of T is called the path if n_1 is the root and n_i is the parent of n_{i+1} for all $i = 1,\ldots,k-1$.

For each node n, the *node label* of n is the triple $NL(n) = \langle N(n), V(n), HAS(n)\rangle$ such that $N(n)$ and $V(n)$are strings called the *node name* and *node value*, respectively, and $HAS(n) = \{HA_1,\ldots,HA_{n_\ell}\}$ is called the set of the *HTML attributes* of n, where each HA_i is of the form $\langle a_i, v_i\rangle$ and a_i, v_i are strings called *HTML attribute name*, *HTML attribute value*, respectively.

If $N(n) \in \Sigma^+$ and $V(n) = \varepsilon$, then the n is called the *element node* and the string $N(n)$ is called the *tag*. If $N(n) = \sharp TEXT$ for the reserved string $\sharp TEXT$ and $V(n) \in \Sigma^+$, then n is called the *text node* and the $V(n)$ called the *text value*. We assume that every node $n \in I\!N$ is categorized to the *element node* or *text node*.

An HTML file is called a page. A page P is corresponding to an ordered labeled tree. For the simplicity, we assume that the P contains no comment part, that is, any string beginning the `<!` and ending the `>` is removed from the P.

Definition 1 For a page P, the P_t is the ordered labeled tree defined recursively as follows.

1. If P contains an empty tag **<tag>**, P_t has the element node n such that it is a leaf P and $N(n) = tag$.
2. If P contains a string $t_1 \cdot w \cdot t_2$ such that t_1 and t_2 are tags and the w contains no tag, then P_t has the text node n such that it is a leaf P and $V(n) = w$.
3. If P contains a string of the form

$$\texttt{<tag } a_1 = v_1,\ldots,a_\ell = v_\ell\texttt{>}w\texttt{</tag>},$$

then the tree $n(n_1,\ldots,n_k)$ is the subtree of P on n, where $N(n) = tag$, $HAS(n) = \{\langle a_i, v_i\rangle \mid i = 1,\ldots,\ell\}$, and $n_1,\ldots,n_k$ are the trees $t_1,\ldots,t_k$ obtained recursively from the w by the 1, 2 and 3.

Next we define the functions to get the node names, node values, and HTML attributes from given nodes and HTML trees defined above. These functions are useful to explain the algorithms in the next section. These functions return the values indicated below and return *null* if such values do not exist.

- Parent(n): The parent of the node $n \in I\!N$.
- ChildNodes(n): The sequence of all children of n.
- Name(n): The node name $N(n)$ of n.
- Value(n): The concatenation $V(n_1)\cdots V(n_k)$ of all values of the leaves $n_1,\ldots,n_k$ of the subtree on n in the left-to-right order.
- Pos(n): The number of big brothers n_i of n such that $N(n_i) = N(n)$.

The following functions are to get HTML attributes.

- HTMLAttSet(n): The HTML attribute set $HAS(n)$ of the node $n \in I\!N$.
- HTMLAttName(n, i): The HTML attribute name a_i of $\langle a_i, v_i\rangle$ in $HAS(n)$.
- HTMLAttValue(n, i): The HTML attribute value v_i of $\langle a_i, v_i\rangle$ in $HAS(n)$.

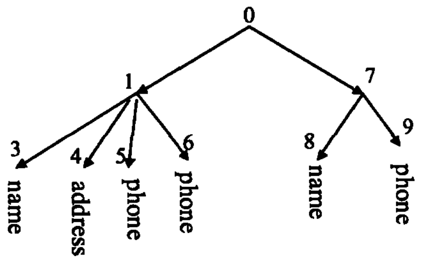
Finally, we define the notion of *common HTML attribute set* of HTML attribute sets and define the function which gets the common HTML attribute set.

Definition 2 Let $S = \{HAS(n_i) \mid i = 1,\ldots,k\}$ and $HAS(n_i) = \{\langle a_{i_1}, v_{i_1}\rangle,\ldots,\langle a_{i_\ell}, v_{i_\ell}\rangle\}$ $(i = 1,\ldots,k)$. The common HTML attribute set of the S, denoted by $CHAS(S)$, is the set $(\cap_{1\leq i\leq k} HAS(n_i)) \cup S'$, where S' is the set of $\langle a, *\rangle$ such that each $HAS(n_i)$ contains an HTML attribute $\langle a, v\rangle$ for the a and $\langle a, v_i\rangle \in HAS(n_i)$, $\langle a, v_j\rangle \in HAS(n_j)$ and $v_i \neq v_j$, where the $*$ is a special symbol not belonging to Σ.

- CommonAttSet($HAS(n_1),\ldots,HAS(n_k)$): The common HTML attribute set of all HTML attribute set $HAS(n_i)$ for $i = 1,\ldots,k$.

What the HTML Wrapper of this paper extracts is the text values of text nodes. These text nodes are called *text attributes*. A sequence of text attributes is called *tuple*. We assume that the contents of a page P is a set of tuple $t_i = \langle ta_{i_1},\ldots,ta_{i_K}\rangle$, where the K is a constant for all pages P. It means that all text attributes in any page is categorized into at most K types. Let us consider an example of HTML document of address list. This list contains three types of attributes, name, address, and phone number. Thus, a tuple is of the form $\langle name, address, phone\rangle$. However, this tuple can not handle the case that some elements contain more than two values such as some one has two phone numbers. Thus, we expand the notion of tuple to a sequence of a set of text attributes, that is $t = \langle ta_1, \ldots, ta_K\rangle$ and $ta_i \subseteq I\!N$ for all $1 \leq i \leq K$. The set of tuples of a page P is called the *label* of P.

Figure 1: The tree of the text attributes, *name*, *address*, and *phone*.



The Fig.1 denotes the tree containing the text attributes *name*, *address*, and *phone*. The first tuple is $t_1 = \langle\{3\},\{4\},\{5,6\}\rangle$ and the second tuple is $t_2 = \langle\{8\},\{\},\{9\}\rangle$. The third attribute of t_1 contains two values and the second attribute of t_2 contains no values.

The Learning Algorithm for the Tree-Wrappers

In this section, we give the two algorithm. The first algorithm execT(P_t,W) extracts the text value of the text attributes from the page P_t using given the Tree-Wrapper W. The second algorithm learnT(E) finds the Tree-Wrapper W for the sequence $E = \ldots, \langle P_n, L_n\rangle, \ldots$, where L_n is the label of the page P_n. A pair $\langle P_n, L_n\rangle$ is called an *example*.

The Tree-Wrapper

Definition 3 The *extraction node label* is a triple $ENL = \langle N, Pos, HAS\rangle$, where N is a node name, $Pos \in I\!N \cup \{*\}$, HAS is an HTML attribute set. The *extraction path* is a sequence $EP = \langle ENL_1, \ldots, ENL_\ell\rangle$.

An $ENL = \langle N, Pos, HAS\rangle$ of a node n is considered as the generalization of which contains the node name, node value, and the value of the function Pos. The first task of execT is to find a path in P_t which matches with the given EP and to extract the text value of the last node of the path. The following function and definition gives the semantics of the matching.

---

boolean **isMatchENL**(n, ENL)
/*input: node n, $ENL = \langle N, Pos, HAS\rangle$*/
/*output: true or false*/
if(N==Name(n) && (Pos==Pos(n) || Pos==$*$) &&
  isMatchHAS(n,HAS)) == true;
else return false;

boolean **isMatchHAS**(n, HAS)
/*input: node n, HAS=$\langle HA_1, \ldots, HA_m, \ldots, HA_M\rangle$,*/
/*HA_m=(a_m, v_m)*/
/*output: true or false*/
for(m=1; $m \leq M$; m++){
  if(HTMLAttValue(n,m) $\neq v_m$ && $v_m \neq *$)
    return false;
} return true;

Definition 4 Let ENL be an extraction node label and n be a node of a page P_t. The ENL *matches* with the n if the function isMatchENL(n, ENL) returns true. Moreover, let $EP = \langle ENL_1, \ldots, ENL_\ell\rangle$ be an extraction path and $p = \langle n_1, \ldots, n_\ell\rangle$ be a path of a page P_t. The EP *matches* with the p if the ENL_i matches with n_i for all $i = 1, \ldots, \ell$.

Definition 5 The Tree-Wrapper is the sequence $W = \langle EP_1, \ldots, EP_K\rangle$ of extraction paths $EP_i = \langle ENL_1^i, \ldots, ENL_{\ell_i}^i\rangle$, where each ENL_j^i is an extraction label.

The algorithm execT is given as follows.
Algorithm **execT**(P_t, W)
/* input: $W = \langle EP_1, \ldots, EP_K\rangle$ and P_t */
/* output: The label $L_t = \{t_1, \ldots, t_m\}$ */

1. For each EP_i ($i = 1, \ldots, K$), find all path $p = \langle n_1, \ldots, n_\ell\rangle$ of P_t such that EP_i matches with p and add the pair $\langle i, n_\ell\rangle$ into the set *Att*. /* The n_ℓ is a candidate for the i-th text attribute.*/
2. Sort all elements $\langle i, n_\ell\rangle \in Att$ in the increasing order of i. Let $LIST$ be the list and $j = 1$.
3. If the length of $LIST$ is 0 or $j > m$, then halt. If not, find the longest prefix *list* of $LIST$ such that all element is in non-decreasing order of i of $\langle i, n\rangle$ and for all $i = 1, \ldots, K$, compute the set $ta_i = \{n \mid \langle i, n\rangle \in list\}$. If the *list* is empty, then let $ta_i = \emptyset$.
4. Let $t_j = \langle ta_1, \ldots, ta_K\rangle$, $j = j + 1$, remove the *list* from $LIST$ and go to 3.

The learning algorithm

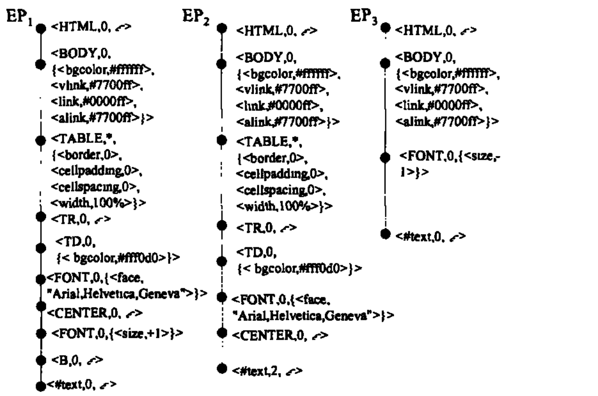
Given a pair $\langle P_n, L_n\rangle$, the learning algorithm learnT calls the function learnExPath which finds the extraction path EP_i^n for the i-th text attributes and $i = 1, \ldots, K$ and it computes the composite $EP_i \cdot EP_i^n$, where EP_i is the extraction path for the i-th text attribute found so far. The definition of the *composite* $EP_i \cdot EP_i^n$ is given as follows and Fig.2 is an example for a composite of two extraction path.

Definition 6 Let ENL_1 and ENL_2 be any extraction node labels. The composite $ENL_1 \cdot ENL_2$ is the extraction node label $ENL = \langle N, Pos, HAS \rangle$ such that

1. $N = N_1$ if $N_1 = N_2$ and ENL is undefined otherwise,
2. $Pos = Pos_1$ if $Pos_1 = Pos_2$, and $Pos = *$ otherwise,
3. $HAS = \text{CommonAttSet}(HAS_1, HAS_2)$.

Definition 7 Let $EP_1 = \langle ENL^1_n, \ldots, ENL^1_1 \rangle$ and $EP_2 = \langle ENL^2_m, \ldots, ENL^2_1 \rangle$ be extraction paths. The $EP = EP_1 \cdot EP_2$ is the longest sequence $\langle ENL^1_\ell \cdot ENL^2_\ell, \ldots, ENL^1_1 \cdot ENL^2_1 \rangle$ such that all $ENL^1_i \cdot ENL^2_i$ are defined for $i = 1, \ldots, \ell$, where $\ell \leq min\{n, m\}$.

Figure 2: The composite of extraction paths.



The learnExPath calls the function getPath(n) which finds the path p from the root to the node n and for each node n_i of $p = \langle n_\ell, \ldots, n_1 \rangle$ ($n_1 = n$), it computes the ENL_i and returns the $EP = \langle EP_\ell, \ldots, EP_1 \rangle$. Finally, the complete description of the learning algorithm learnT is given as follows.

Experimental Results

We equip the learning algorithm by Java language and experiment with this prototype for HTML documents. For parsing HTML documents, we use the OpenXML 1.2 (http://www.openxml.org) which is a validating XML parser written in Java. It can also parse HTML and supports the HTML parts of the DOM (http:www.w3.org/DOM).

The experimental data of HTML pages is collected by the citeseers which is a scientific literature digital library (http://citeseers.nj.nec.com). The data consists of 1,300 HTML pages. We choose att_1 = "the title", att_2 = "the name of authors", and att_3 = "the abstract" as the text attributes and practice the learnT.

All pages are indexed to be $P_1, \ldots, P_{1300}$ in the order of the file size. The training example is $E = \{\langle P_i, L_i \rangle \mid i = 1, \ldots, 10\}$, where the L_i is the label made from the P_i in advance. The result is shown in Fig. 3 which is the Tree-Wrapper W found by learnT(E).

---

learnT(E)
/*input: $E = \{\ldots, \langle P_n, L_n \rangle, \ldots\}$*/
/*output: Tree-Wrapper $\langle P_n, L_n \rangle$ */
/* execT(P_n, W)=L_n Tree-Wrapper W*/
$L_n = \{t_i \mid 1 \leq i \leq M_n\}$;
$t_i = \langle ta^i_1, \ldots, ta^i_K \rangle$;
$NODE(n,k) = \cup_{1 \leq i \leq M_n} ta^i_k$;
n=1;
do while($\langle P_{n+1}, L_{n+1} \rangle$){
  for(k=1; $k \leq K$; k++){
    if($NODE(n,k)$ &&
      EP^n_k=leanExPath($NODE(n,k)$) &&
      $EP_k = \varepsilon$) $EP_k = EP^n_k$;
    if($NODE(n,k)$ &&
      EP^n_k=leanExPath($NODE(n,k)$)&&
      EP_k != ε) $EP_k = EP_k \cdot EP^n_k$;
    else ;
  } $n = n + 1$;
} return $W = \langle EP_1, \ldots, EP_K \rangle$;

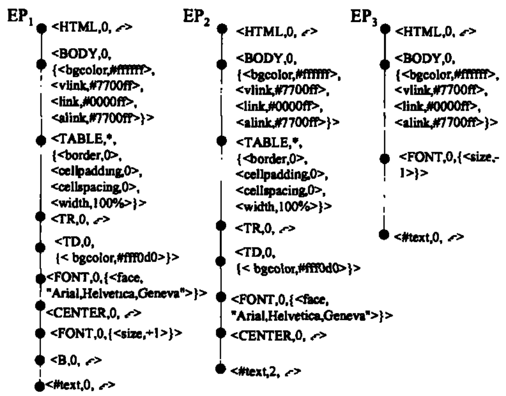
---

learnExPath($NODE(n,k)$)
/* input: $NODE(n,k) = \{n_1, \ldots, n_m\}$ */
/* output: extraction path EP */
if(m==1) return EP=getPath(n_1);
else j=1;
for(i=1; $i \leq m$; i++) $current_i = n_i$;
while($\forall i$ Name($current_i$)==Name($current_1$)){
  $ENL_j = \langle N_j, Pos_j, HAS_j \rangle$;
    N_j=Name($current_1$);
    if($\forall i$(Pos($current_i$)==Pos($current_1$)))
      Pos_i=Pos($current_1$);
    else $Pos_i = *$;
    $HAS_j = CHAS(S)$;
      S={HTMLAttSet($current_i$) | $i = 1, \ldots, m$};
  for(i=1; $i \leq m$; i++) $current_i$=Parent($current_i$);
  $j = j + 1$;
}
if(j==1) return false;
else return $EP_k = \langle ENL_{j-1}, \ldots, ENL_1 \rangle$;

---

getPath(n)
/* input: node n */
/* output: extraction path EP from the root to n */
$current = n$; i=1;
do while(Parent($current$)){
  $ENL_i = \langle N_i, Pos_i, HAS_i \rangle$;
    N_i=Name($current$);
    Pos=Pos($current$);
    HAS_i=HTMLAttSet($current$);
  $current$=Parent($current$); $i = i + 1$;
} return $EP = \langle ENL_{i-1}, \ldots, ENL_1 \rangle$;

Figure 3: The Tree-Wrapper found by learnT



Next, we practice the execT(P_i, W) for the remained pages P_i $(i = 11, \ldots, 1300)$ to extract all tuples $\langle att_1, att_2, att_3 \rangle$ from P_i. The three pages can not be extracted. The P_{1095} is one of the pages. We explain the reason by this page. In Fig. 3, we can find that the first extraction path EP_1 contains the extraction node label for the TABLE tag. The HTML attribute set HAS of this node contains the attribute "cellpadding" whose value is 0. However, the corresponding node in P_{1095} has the HTML attribute value "cellpadding= 1". Thus, the EP_1 does not match with the path.

Any other pages are exactly extracted, thus, we conclude that this algorithm is effective for this site.

Next, we construct the following XML like database from the extracted contents of the pages and run the LR-Wrapper to divide the titles and the years. This XML file C_{11} corresponds to the content of the P_{11}.

```
<contents>
<tupple>
<attr1>A System for Induction of Oblique Decision Trees(1994)</attr1>
<attr2>Sreerama K. Murthy, Simon Kasif, Steven Salzberg</attr2>
<attr3>This article describes a new system for induction of oblique decision trees. This system, OC1, combines...
</attr3>
</tupple>
</contents>
```

The learned LR-Wrapper is as follows. On some pages, the published year are omitted. In such case, the LR-Wrapper can not extract the exact texts.

```
⟨ ('<contents> ↓ <tupple> ↓ <attr1>', '('),
 ('(', ')</attr1> ↓ <attr2>'),
 (')</attr1> ↓ <attr2>', '</attr2> ↓ <attr3>'),
 ('</attr2> ↓ <attr3>',
  '</attr3> ↓ </tupple> ↓ </contents> ↓ ') ⟩
```

The future work of this study is to expand the Tree-Wrapper so that it can extract the HTML attributes rather than the text attributes. We also expand the prototype to extract substrings of the text values for overcome the above difficulty.

References

Abiteboul, S., Buneman, P., and Suciu, D. 2000. Data on the Web: From relations to semistructured data and XML, Morgan Kaufmann, San Francisco, CA, 2000.

Angluin, D. 1988. Queries and concept learning. *Machine Learning* 2:319–342.

Cohen, W. W. and Fan, W. 1999. Learning Page-Independent Heuristics for Extracting Data from Web Pages, *Proc. WWW-99.*

Craven, M., DiPasquo, D., Freitag, D., McCallum, A., Mitchell, T., Nigam, K., and Slattery, S., 2000. Learning to construct knowledge bases from the World Wide Web, *Artificial Intelligence* 118:69–113.

Freitag, D. 1998. Information extraction from HTML: Application of a general machine learning approach. *Proc. the Fifteenth National Conference on Articial Intelligence*, pp. 517-523.

Hirata, K, Yamada, K., and Harao, M. 1999. Tractable and intractable second-order matching problems. *Proc. 5th Annual International Computing and Combinatorics Conference.* LNCS 1627:432–441.

Hammer, J., Garcia-Molina, H., Cho, J., and Crespo, A. 19967. Extracting semistructured information from the Web. *Proc. the Workshop on Management of Semistructured Data*, pp. 18–25.

Hsu, C.-N. 1998. Initial results on wrapping semistructured web pages with finite-state transducers and contextual rules. *In papers from the 1998 Workshop on AI and Information Integration*, pp. 66–73.

Kamada, T. 1998. Compact HTML for small information appliances. *W3C NOTE 09-Feb-1998.* www.w3.org/TR/1998/NOTE-compactHTML-19980209

Kushmerick, N. 2000. Wrapper induction: efficiency and expressiveness. *Artificial Intelligence* 118:15–68.

Muslea, I., Minton, S., and Knoblock, C. A. 1998. Wrapper induction for semistructured, web-based information sources. Proc. *the Conference on Automated Learning and Discovery.*

Sakamoto, H., Arimura, H., and Arikawa, S. 2000. Identification of tree translation rules from examples. *Proc. 5th International Colloquium on Grammatical Inference.* LNAI 1891:241–255.

Valiant, L. G. 1984. A theory of the learnable. *Commun. ACM* 27:1134–1142.