Information Extraction from Structured Documents using k-testable Tree Automaton Inference

Raymond Kosala, Hendrik Blockeel, Maurice Bruynooghe

Katholieke Universiteit Leuven

Dept. of Computer Science

Celestijnenlaan 200A

B-3001 Leuven, Belgium

{raymond,maurice,hendrik}@cs.kuleuven.ac.be

Jan Van den Bussche

Limburgs Universitair Centrum

Department WNI

Universitaire Campus

B-3590 Diepenbeek, Belgium

jan.vandenbussche@luc.ac.be

Abstract

Information extraction (IE) addresses the problem of extracting specific inform-

ation from a collection of documents. Much of the previous work on IE from structured documents, such as HTML or XML, uses learning techniques that are based on strings, such as finite automata induction. This paper explores methods that exploit the tree structure of the documents. In particular, our method infers a k-testable tree automaton from a small set of annotated examples and explores various ways to generalize the inferred automaton. Experimental results on the benchmark data sets show that our approach compares favorably to the previous approaches.

1 Introduction

Information extraction (IE) addresses the problem of extracting specific information from a collection of documents. Basically work on IE can be classified into three categories: IE from unstructured texts, IE from semi-structured texts and IE from structured texts [51]. Classical or traditional IE tasks from unstructured natural language texts typically use various forms of linguistic pre-processing. An example domain, which is a task investigated in the Sixth Message Understanding Conference (MUC-6) [37], is "Management Succession". Given an article, the tasks are to extract the name of the new company officers or the old officers, the company name, and the position title succeeded. IE from semi-structured texts arises from the need to extract, for example a date field from online advertisements or announcements. Semi-structured texts typically use ungrammatical language. It requires a non-linguistic approach to extract information from such documents.

With the increasing popularity of the World Wide Web (Web) as a medium for disseminating information and the work on Web information integration [35, 55], there is

a need for IE systems that support the extraction of information from Web documents. Many documents from the Web, which are stored as HTML and/or XML documents, are structured. These documents rely on non-linguistic structures, such as HTML/XML tags, and sometimes use ungrammatical language to convey information, making the methods appropriate for grammatical text unusable.

While there are several query languages supporting the extraction of information from web data [5, 57], their use is time consuming and requires nontrivial skill. As argued for example in [41, 33], there is a need for systems that can learn to extract information from a few annotated examples. The reason is that building IE systems manually is not feasible and scalable for such a dynamic and diverse medium as the Web. The problem, also known as wrapper induction, has already been addressed by several authors. Several machine learning techniques for inducing wrappers have been proposed such as rule learning algorithms, e.g., [17], and multi-strategy approaches [18]. In [41, 20, 19, 51, 16, 27, 10] grammatical inference techniques are used to induce a kind of delimiter-based patterns.

These methods consider the document as a string. However, structured documents such as HTML and XML documents have a tree structure. Therefore it is natural to explore the use of tree automata for IE from structured documents. Indeed, tree automata are well-established and natural tools for processing trees [14]. An advantage of using the more expressive tree formalism is that the extracted field can depend on its structural context in a document. A structural context that is close to the target field in the tree structure of the document can be arbitrarily far away in the string representing the document, making the learning task very difficult and resulting in wrappers with rather poor performance.

The current paper develops a novel wrapper induction method that utilizes the tree structure of the document. Accordingly, it uses tree automata as wrappers. Recent work by Gottlob and Koch [23] shows that all existing wrapper languages for structured document IE can be captured using tree automata. This result provides a strong justification for the use of tree automata instead of string automata.

We will use the k-testable tree automaton inference algorithm [46], an algorithm for grammatical inference that is able to identify in the limit [22] any k-testable tree language (in the strict sense) from positive examples only. Informally, a k-testable tree language is a language that can be determined just by looking at all the subtrees of length k. The amount of generalization occurring when learning from the positive examples of the k-testable tree algorithm is mainly determined by the value of k; it decreases with increasing k. It is an algorithm for ranked trees, while documents are unranked trees. The simplest way to apply the algorithm is to convert web documents into (ranked) binary trees. As the extraction is based on some structural context, k must be large enough such that the field to be extracted and its structural context are covered in the same subtree. Because of the binarisation, the value of k needed for capturing the structural (or distinguishing) context tends to be rather large. Consequently, the generalization tends to be rather low, often resulting in rather poor recall (as we reported in [31]).

To overcome this problem, we have experimented with two generalizations of the k-testable algorithm, namely, the g-testable and gl-testable algorithms. In the g-testable algorithm [29], the generalization is parameterized by l. It considers generalizations of states (which are trees) where the state labels at the lowest l levels are replaced by wildcards. The gl-algorithm, which is introduced in this paper, considers another generalization and uses the partial order between different generalizations to limit the search.

Experiments show that these generalizations improve the performance of the induced wrappers.

Not only does our method exploit the tree structure, it also requires very little user intervention. The user only has to annotate the field to be extracted in a few representative examples. Previous approaches required substantially more user intervention such as splitting the document in small fragments, and selecting some of them for use as a training example, e.g., [51]; the manual specification of the length of a window for the prefix, suffix and target fragments [20, 19], and of the special tokens or landmarks such as ">" or ";" [19, 41].

Preliminary version of parts of this article appeared in conference proceedings (k-testable algorithm [31], g-testable algorithm [29]).

The rest of the paper is organized as follows. Section 2 provides some background on tree automata and their use for IE. Section 3 describes our methodology, the k-testable algorithm and its generalisations: the g-testable and gl-testable algorithms. Experimental setting and results are described in Section 4 and related work in Section 5. In Section 6 we conclude by summarizing the contributions of this paper.

2 Preliminaries

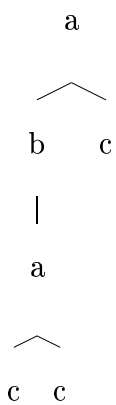
2.1 Grammatical inference

Grammatical inference refers to the process of learning rules from a set of labeled examples. It belongs to a class of inductive inference problems [2] in which the target domain is a formal language (a set of strings over some alphabet Σ) and the hypothesis space is a family of grammars. It is also often referred to as automata induction, gram-

mar induction, or automatic language acquisition. It is a well-established research field in AI that goes back to Gold's work [22]. The inference process aims at finding a minimum automaton (the *canonical automaton*) that is *compatible* with the examples. The compatibility with the examples depends on the applied quality criterion. Quality criteria that are generally used are exact learning in the limit of Gold [22], query learning of Angluin [1] and probably approximately correct (PAC) learning of Valiant [53]. There is a large body of work on grammatical inference, for excellent surveys see, e.g., [38, 49, 44].

In regular grammar inference, we have a finite alphabet Σ and a regular language $L \subseteq \Sigma^*$. Given a set of examples that are in the language (S^+) and a (possibly empty) set of examples not in the language (S^-), the task is to infer a deterministic finite automaton (DFA) A that accepts the examples in S^+ and rejects the examples in S^-.

2.2 Tree automata



Assume given a finite set V of labels, each with an associated rank (or arity; a natural number). Trees labeled by V are formally defined as terms over V, as follows: a label of rank 0 ($f/0$ or just f) is a tree; and if f/n is a label of rank $n > 0$ and $t_1, \ldots, t_n$ are trees, then $f(t_1, \ldots, t_n)$ is a tree. For example the term $a(b(a(c, c)), c)$ with $a/2, b/1, c/0 \in V$, represents the tree on the left.

A deterministic tree automaton (DTA) M is a quadruple (V, Q, Δ, F), where V is a set of ranked labels, Q is a finite set of states, $F \subseteq Q$ is a set of final (accepting) states, and $\Delta : \bigcup_k V_k \times Q^k \to Q$ is the transition function. Here, V_k denotes the subset of V of labels of rank n. For example, $(v, q_1, \ldots, q_k) \to q$, where $v/k \in V_k$ and $q, q_i \in Q$, represents a transition.

A DTA processes trees bottom up. Given a leaf labeled $v/0$ and a transition $(v) \to q$, the state q is assigned to it. Given a node labeled v/k with children in state $q_1, \ldots, q_k$ and a transition $(v, q_1, \ldots, q_k) \to q$, the state q is assigned to it. We say that a tree is accepted if the state assigned to its root is accepting, i.e., belongs to F.

Grammatical inference can be generalized from string languages to tree languages. Rather than a set of strings over an alphabet Σ given as example, we are now given a set of trees over a ranked alphabet V. Rather than inferring a standard finite automaton compatible with the string examples, we now want to infer a compatible tree automaton. Tree automata are the natural generalisation of string automata. Typically algorithms for tree automata induction are developed by upgrading the existing algorithms for string automata induction (e.g., [46, 48]).

2.3 Information extraction by grammatical inference

If we model structured documents as trees over some ranked alphabet V as above, an IE task can be reduced to a grammatical inference task (as noted by Freitag [16]). Specifically, suppose the IE task consists of selecting certain nodes from a tree. We are given a set of examples, each consisting of a tree and a selected node. By adding for each label $v \in V$ a new label (v, x), where x is a new "target" symbol, we can represent such examples as trees over the new alphabet $V' = V \cup (V \times \{x\})$, where the label of precisely one node, namely the selected one, is in $V \times \{x\}$, and the other labels are in V as before.

We can now try to infer a grammar for the obtained set of example trees, producing a DTA M over V'. If successful, we can use M to perform the original IE task simply by selecting each node, one by one, relabeling it to (v, x) if its original label is x, and verifying whether M accepts the thus relabeled tree. If so, the selected node is extracted.

In what follows, we will focus on applications where only leaf nodes are to be extracted, and where all these nodes have a fixed known label. We can therefore simplify the setting a bit by labeling selected nodes simply by the target symbol x instead of (v, x), because v is fixed and known and therefore uninformative. The new alphabet V' then simply becomes $V \cup \{x\}$.

3 Approach and algorithms

Structured documents in HTML, or, more generally, XML format, can be readily represented as trees, where internal nodes represent the elements, and are labeled by tags, and leaf nodes represent the text content. Before we can use grammatical inference to perform IE on such trees, as described above, we must deal with two issues:

1. How do we deal with text content?

2. Tags are not ranked. For example, in HTML, an `<ul>` element can have an arbitrary number of `<li>` subelements, and more generally, in XML documents, there is no bound on the number of subelements an element can have.

In the next two subsections, we will deal with these two issues. After that, we summarize our general approach. Finally, we introduce the various concrete grammatical inference algorithms we will use for IE from structured documents.

3.1 Preprocessing

Figure 1 shows a simplified view of a representative document. (The real documents, as used in the experiments by us and the other authors, are more complex.) In this document,

the fields to be extracted are the fields following the '`Alt. Name`' and '`Organization`' fields. A document consists of a variable number of records. As we can see, in each record the number of occurrences of the fields to be extracted is also variable (from zero to several occurrences). Also the position where they occur is not fixed. There is evidence that extracting this kind of information is a difficult task [26, 40].

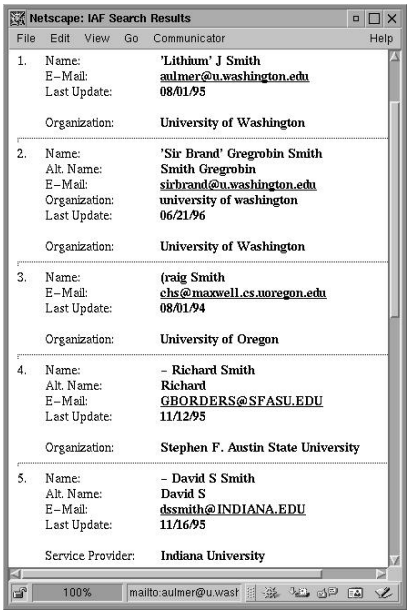
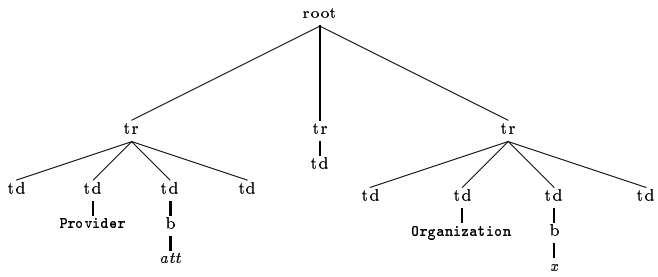


Figure 1: An example of a HTML document

An important issue is how to deal with the various text nodes in the document. Treating every piece of text as a distinct label is unacceptable as it results in too specific automata. Labeling all text nodes (but the node to be extracted which is labeled `x`) by some fixed label `CDATA`, as in XML DTD's [56], is also unacceptable, as this results in too general automata. Indeed, consider the following fragment of a document tree that could originate from the document shown above:



Suppose the target field `x` is always preceded by a field labeled `Organization`. If the labels `Provider` and `Organization` are both replaced by `CDATA` then any automaton that extracts the `x` node will likely also extract the `att` node when it is replaced by `x`. Hence we

should not replace the field `Organization` by CDATA. Fields such as `Organization` and `Alt.Name` are called *distinguishing contexts* (or structural context). Roughly speaking, a distinguishing context is the text content of a tree node that is useful for the identification of the field of interest. However, not every field of interest has a unique distinguishing context.

In our experiments, we consistently used the following procedure to determine the distinguishing context. We look for the invariant text label that is nearest to the field of interest and occurs at the same distance from the field of interest in all examples. For example, the text 'Organization:' is an invariant text label that is nearest to the organization name in HTML document figure at the beginning of this Section. If no such text is found, no context is used and all text is turned into CDATA. If there are several possibilities, one is chosen at random. As distance measure, we use the length of the shortest path in the document tree (for example the distance of a node to its parent is one; to its sibling, two; to its uncle, three).

3.2 Conversion to ranked trees

Existing tree automata inference algorithms expect ranked trees. The simplest way to apply them on HTML or XML documents, which are unranked trees, is to transform the latter into (ranked) binary trees. This is the approach that we follow in this paper.

Using the symbol T to denote unranked trees and F to denote a sequence of unranked trees (a forest), the following grammar defines unranked trees:

$$T ::= a(F),\ a \in V \qquad\qquad F ::= \epsilon$$

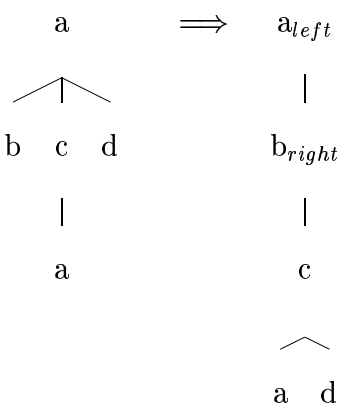
$$F ::= T, F$$

The transformation we use can be formally defined with the following recursive function *encode* (with $encode_f$ for the encoding of forests):

$$encode(T) \stackrel{\text{def}}{=} encode_f(T)$$

$$encode_f(a(F_1), F_2) \stackrel{\text{def}}{=} \begin{cases} a & \text{if } F_1 = F_2 = \epsilon \\ a_{right}(encode_f(F_2)) & \text{if } F_1 = \epsilon, F_2 \neq \epsilon \\ a_{left}(encode_f(F_1)) & \text{if } F_1 \neq \epsilon, F_2 = \epsilon \\ a(encode_f(F_1), encode_f(F_2)) & \text{otherwise} \end{cases}$$

Informally, the first child of a node v in an unranked tree T is encoded as the left child of the corresponding node v' of T', the binary encoding of T, while the right sibling of a node v in tree T is encoded as the right child of v' in T'. To distinguish between a node with one left child and a node with one right child, the node is annotated with left and right respectively. For example, the unranked tree $a(b, c(a), d)$ is encoded into the binary tree $a_{left}(b_{right}(c(a, d)))$ as pictorially shown below. Note that the binary tree has exactly the same number of nodes as the original tree.



An advantage of transforming the HTML/XML document trees into (ranked) binary trees is that we can directly use the tree automata inference algorithms that have been

proposed in the literature, such as [21, 46, 7, 3]. In this paper we explore the application of the k-testable algorithm developed in [21, 46]. We choose the k-testable algorithm because it requires fewer examples than the other tree-based algorithms, and hence less effort from the user (who needs to provide these labeled examples). A drawback of converting an unranked tree to a binary tree is that the distance between the distinguishing context and the target node can increase. A so far less explored alternative is to work directly with unranked trees. Unranked tree automata formalism exist, *e.g.*, [43, 52]. They have transition rules of the form $(v, e) \rightarrow q$, where e is a regular expression that describes a sequence of states. [30] is a first step in this direction.

3.3 Approach

Our approach for information extraction has the following characteristics:

- Strings stored at the nodes are treated as a whole. If extracted, the whole node is returned.
- One automaton is learned for one type of field to be extracted, e.g., the field following "Organization".
- In the examples used during learning, one target field is replaced by x. When a document contains several fields of the same type, then several examples are created from it, one for each occurrence of the target field.

One characteristic of our tree automata wrappers is that they do single-slot (or single-field) extraction. A single-slot IE system extracts isolated facts from the text, while a multi-slot IE system groups the related extracted fields (called a case frame) together into correctly ordered multi-slot facts. There are some domains where multi-slot extraction is a necessity. For example, a webpage may contain a list of house addresses with their cor-

responding prices. Unless the address and the price are combined in a pair, the extracted information is rather useless.

Multi-slot extraction can be achieved by extracting also the location of the extracted fields. Knowing the locations, one can combine the extracted fields into the correct tuples. This simple post-processing method works for structured documents such as HTML/XML documents, since the order of the fields in a case frame typically follows the order of their position in the document. However, we did not collect the extracted fields in a case frame in our experiments.

The learning procedure is as follows:

1. Replace in the examples the target field by "x", the distinguishing context (if present) by "ctx" and all other text fields by `CDATA`.

2. Convert the example trees to binary trees.

3. Run a tree automaton inference algorithm on the examples and return the inferred automaton.

The extraction procedure is as follows:

1. Replace the distinguishing context (if present) by "ctx" and all other text fields by `CDATA`.

2. Convert the tree to a binary tree.

3. Repeat for all `CDATA` nodes:
   - Replace the label of one `CDATA` node by the special label 'x'.
   - Run the inferred tree automaton.

- If the tree is accepted by the automaton, then extract the original text of the node labeled with x.

The automaton can succeed for zero, one or more text nodes. The text nodes for which it succeeds are the extracted fields. Only the first step of the learning procedure requires user intervention. The second step of the learning procedure and the whole extraction procedure are done automatically. The above procedures are repeated for each field of interest in the dataset. If one wants to do a novel extraction task on a novel dataset, then the learning procedure above has to be done for this novel task. That is, the user should mark the fields of interest and distinguishing context if they exist, then run the learning algorithm on the novel data.

3.4 Tree automaton inference algorithms

The k-testable algorithm is a basic tree automaton inference algorithm. As we will see in Section 4, the algorithm is precise, but is sometimes too specific, as indicated by the low recall. Hence we develop two generalisation algorithms: g-testable and gl-testable. We start with some definitions.

3.4.1 Definitions

With t a tree, height(t) is the number of nodes on the longest path from the root. The *k-root* $r_k(t)$ of a tree t is the tree of height at most k obtained from t by cutting off branches longer than k; the set $f_k(t)$ of *k-forks* is the set of all trees of height k obtained from t by taking all subtrees of height at least k and cutting off branches longer than k; finally the set $s_k(t)$ of *k-subtrees* is the set of all subtrees at the bottom of t of height at most k. Formally:

$$r_k(v(t_1,\ldots,t_m)) = \begin{cases} v & \text{if } k = 1 \\ v(r_{k-1}(t_1),\ldots,r_{k-1}(t_m)) & \text{if } k > 1 \end{cases} \quad (1)$$

$$f_k(v(t_1,\ldots,t_m)) = \begin{cases} \emptyset & \text{if height}(v(t_1,\ldots,t_m)) < k \\ \bigcup_{j=1}^{m} f_k(t_j) \cup \{r_k(v(t_1,\ldots,t_m))\} & \text{otherwise} \end{cases} \quad (2)$$

$$s_k(v(t_1,\ldots,t_m)) = \bigcup_{j=1}^{m} s_k(t_j) \cup \begin{cases} \emptyset & \text{if height}(v(t_1,\ldots,t_m)) > k \\ v(t_1,\ldots,t_m) & \text{otherwise} \end{cases} \quad (3)$$

Note that the k-root and the k-subtrees have height at most k, and that the k-forks have height exactly k.

Example 1 *Suppose* $t = a(b(a(b,x)),c)$ *then* $r_2(t) = \{a(b,c)\}$; $f_2(t) = \{a(b,c), b(a), a(b,x)\}$; *and* $s_2(t) = \{a(b,x), b, x, c\}$.

The *level* of a node is defined as the number of edges on the path from the node to the root. The skeleton of tree t, $skeleton(t)$, is defined as t with all of its labels, except the root label, changed to a wildcard $*$. For example, $skeleton(a(b(d),c(e,f))) = a(*(*),*(*,*))$. A *partition* of a set S is a set of disjoint nonempty subsets of S (called *classes*) such that the union of the subsets is S. The *children* of a tree $v(t_1,\ldots,t_m)$ are $t_1,\ldots,t_m$. A tree t *covers* a tree t' if t' can be derived from t by replacing some of the wildcards in t.

3.4.2 The k-testable algorithm

The k-testable algorithm [46] is parameterized by a natural number k; its name comes from the notion of a "k-testable tree language". Informally, a tree language (set of trees) is k-testable if membership of a tree in the language can be determined just by looking at its $(k-1)$-root, k-forks, and $(k-1)$-subtrees. The k-testable algorithm is capable

of identifying in the limit any k-testable tree language from positive examples only. We have selected it because the information to be extracted typically has a locally testable character. Intuitively, given an example, the right value of k is the minimal value that ensures that the target x and the distinguishing context are in the same fork.

The choice of k is performed automatically using cross-validation, choosing the smallest k giving the best results. Our cross-validation approach takes randomly one half of the dataset for training and uses the rest for testing. First calculating a score for $k = 2$, the value for k is increased until the score shows a decrease. The least k-value giving a maximal score is then selected as the best value. As argued in Section 4 the F1-score we use first increases and then decreases, hence this is a good approach.

The procedure to learn the tree automaton [46] is shown in Algorithm 1. The algorithm uses the $(k-1)$-roots, k-forks and $(k-1)$-subtrees occurring in the examples to derive states and transitions. Note that the algorithm uses trees as states of the inferred automaton: the $(k-1)$-roots, the $(k-1)$-subtrees and the $(k-1)$-roots of the k-forks all become states. It is a simple way to ensure that the state associated with a node not only depends on the label, but also on the states of the children.

Algorithm 1 k-testable

Input: A set T of positive examples (ranked trees over V) and a positive integer k.
Output: A tree automaton (V, Q, Δ, FS).

1: $\mathcal{F} := \bigcup\{f_k(t) \mid t \in T\}$
2: $\mathcal{S} := \bigcup\{s_{k-1}(t) \mid t \in T\}$
3: $FS := \{r_{k-1}(t) \mid t \in T\}$
4: $Q := \mathcal{S} \cup FS \cup \{r_{k-1}(f) \mid f \in \mathcal{F}\}$
5: $\Delta := \{(v, t_1, \ldots, t_m) \to v(t_1, \ldots, t_m) \mid v(t_1, \ldots, t_m) \in \mathcal{S}\}$
6: $\Delta := \Delta \cup \{(v, t_1, \ldots, t_m) \to r_{k-1}(v(t_1, \ldots, t_m)) \mid v(t_1, \ldots, t_m) \in \mathcal{F}\}$

3.4.3 The g-testable algorithm

The basic idea of the g-testable algorithm is to generalize the transitions originating from forks that are not important for the extraction. Important forks are those that contain the target label x. In Algorithm 2, they are collected in the set $\mathcal{TF}$ (target forks), the other ones in the set $\mathcal{OF}$ (other forks). The generalisation is parameterised by a level l. It replaces the label of a node by a wildcard when its level is greater or equal to l. The algorithm uses a function $gen(f, l)$ for this. Figure 2 shows a fork f (left), $gen(f, 1)$ (middle) and $gen(f, 2)$ (right). To prevent overgeneralisation, we require that the generalisation of a fork does not cover any target fork. The value of parameters k and l are determined by the same cross-validation method as explained above.

The meaning of a generalized fork is the set of all trees that can be obtained by instantiating labels for the wildcards. Generalized forks yield transitions with wildcards. For example, with $k = 3$ and $l = 2$, the fork $gen(f, 2)$ from the Figure would yield the transition $a(b, c(*, *)) \rightarrow a(b, c)$, which on a 5-label alphabet $V = \{a, b, c, d, e\}$ effectively stands for the $5^2 = 25$ possible transitions obtained by instantiating labels for the wildcards. Similarly, with $k = 3$ and $l = 1$, the fork $gen(f, 1)$ from the Figure would yield the transition $a(*, *(*, *)) \rightarrow a(*, *)$, which then stands for 5^4 possible transitions obtained by instantiating labels for the wildcards on the left-hand side of the transition. Since the right-hand side stands for the 2-root of the fork, the wildcards on the right-hand side are instantiated in accordance with the left-hand side. Some concrete example instantiations of the transition are:

- $a(b, c(d, e)) \rightarrow a(b, c)$
- $a(d, e(b, c)) \rightarrow a(d, e)$

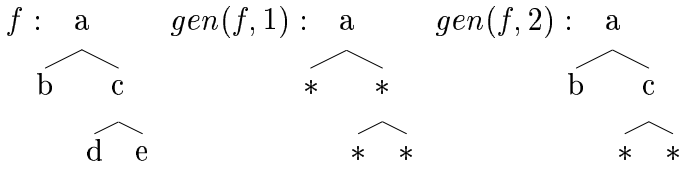


Figure 2: Generalizing a fork.

- $a(a, a(b, b)) \to a(a, a)$

The detailed g-testable procedure is shown in Algorithm 2. Note that the $(k-1)$-subtrees of k-forks are explicitly added as states. In the k-testable algorithm, they were already present as $(k-1)$-subtrees or as $(k-1)$-roots of other forks. For generalized k-forks, this is no longer the case, hence their subtrees are explicitly added. Note also that with $l = k$, the g-testable algorithm will output exactly the same automaton as the k-testable algorithm.

Algorithm 2 g-testable algorithm

Input: A set T of positive examples, parameters k and l.
Output: A tree automaton (V, Q, Δ, FS)

1: $\mathcal{F}_0 := \bigcup\{f_k(t) \mid t \in T\}$
2: $\mathcal{TF} := \{f \in \mathcal{F}_0 \mid f \text{ contains } x\}$
3: $\mathcal{OF}_0 := \mathcal{F}_0 - \mathcal{TF}$
4: $\mathcal{OF}_{\text{nogen}} := \{f \in \mathcal{OF}_0 \mid gen(f, l) \text{ covers one of } \mathcal{TF}\}$
5: $\mathcal{OF}_{\text{gen}} := \mathcal{OF}_0 - \mathcal{OF}_{\text{nogen}}$
6: $\mathcal{F} := \mathcal{TF} \cup \{gen(f, l) \mid f \in \mathcal{OF}_{\text{gen}}\} \cup \mathcal{OF}_{\text{nogen}}$
7: $\mathcal{S} := \bigcup\{s_{k-1}(t) \mid t \in T\}$
8: $FS := \{r_{k-1}(t) \mid t \in T\}$
9: $Q := \mathcal{S} \cup FS \cup \{r_{k-1}(f) \mid f \in \mathcal{F}\} \cup \bigcup\{s_{k-1}(f) \mid f \in \mathcal{F}\}$
10: $\Delta := \{(v, t_1, \ldots, t_m) \to v(t_1, \ldots, t_m) \mid v(t_1, \ldots, t_m) \in \mathcal{S}\}$
11: $\Delta := \Delta \cup \{(v, t_1, \ldots, t_m) \to r_{k-1}(v(t_1, \ldots, t_m)) \mid v(t_1, \ldots, t_m) \in \mathcal{F}\}$

3.4.4 The gl-testable algorithm

As the k-testable algorithm, the gl-testable algorithm has a single parameter k whose optimal value is determined by the same cross-validation method. As the g-testable algorithm, the gl-testable algorithm divides forks in *target forks* $\mathcal{TF}$ and *other forks* $\mathcal{OF}$

and generalises the other forks. However, the amount of generalisation is not determined by a second parameter l, but by a more exhaustive exploration of possible generalisations.

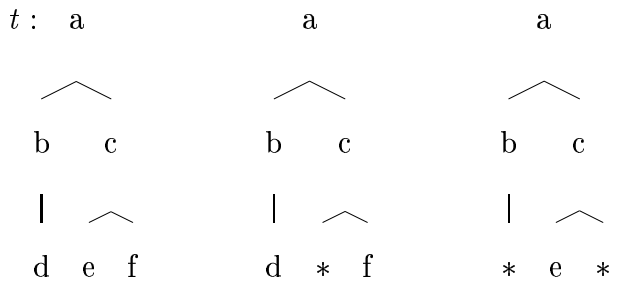
To check that some generalisation is not overly general, we perform a cover test against the target forks (as in the g-testable algorithm) but also another test. This second test checks that the children of a fork (which are also states) do not cover states containing the target x. Such states can originate from subtrees of height $k-1$. These states (target subtrees) are collected in the set $\mathcal{TS}$.

To avoid an exhaustive search over all possible generalisations of a fork, some heuristics are used. As a first heuristic, the other forks are partitioned according to their skeleton by means of a procedure *partition* (not shown). For instance, with $\mathcal{OF} = \{a(d), a(c,d), a(b), a(c), a(b,c), a(c,d,e)\}$, $partition(\mathcal{OF}) = \{\{a(b), a(c), a(d)\}, \{a(b,c), a(c,d)\}, \{a(c,d,e)\}\}$. Then the procedure *pgen* (also not shown) computes a single generalisation of the forks in the same class (with the same skeleton): common labels are preserved but all other labels are replaced by a wildcard $*$. For instance, $pgen(\{a(b(c),d), a(b(c),e), a(b(f),e)\}) = a(b(*),*)$. If this generalisation is not overly general, it is used as initial value for a search of further generalisations. Otherwise, each fork of the partition is considered for generalisation. This further generalisation is performed by the procedure *genl* (Algorithm 4) with as inputs a set of target subtrees, a set of target forks (both used to check against overgeneralisation) and the set of forks to be generalised. This procedure returns the most general forks that are allowed by the overgeneralisation check. It only considers generalisations at the bottommost level (i.e., introducing wildcards only for leaves that are at depth n, with n the height of the tree); this is a heuristic decision inspired by earlier experimental results.

To reduce redundancy, the partial order between generalisations is exploited. One

position at a time is generalised and checked for overgeneralisation before considering further generalisations.[1] The procedure *gentree* (Algorithm 5) returns all acceptable forks with one position generalised.

Consider for instance a fork $a(b(d), c(e, f))$. It has seven possible generalisations for the labels at the bottommost level. They are partially ordered by the covers relation. The most specific ones are $a(b(*), c(e, f))$, $a(b(d), c(*, f))$ and $a(b(d), c(e, *))$ (the ones tested by *gentree* when passed the initial fork), the more general ones are $a(b(*), c(*, f))$, $a(b(*), c(e, *))$ and $a(b(d), c(*, *))$ and $a(b(*), c(*, *))$ is the most general one. The fork and two of its generalisations are shown below.



The detailed gl-testable procedure is shown in Algorithm 3. The used heuristics were determined by experiment.

Example 2 *Applying Algorithm 3 on the term of Example 1 for* $k = 3$*, we obtain:*

- $FS = r_2(t) = \{a(b, c)\}$, $\mathcal{OF} = \{a(b(a), c)\}$, $\mathcal{TF} = \{b(a(b, x))\}$ *and* $\mathcal{S} = s_2(t) = \{a(b, x),\ b,\ x,\ c\}$.
- $\mathcal{TS} = \{a(b, x)\}$
- $\mathcal{P} = \{\{a(b(a), c)\}\}$

[1] A further optimisation would be to store unacceptable generalisations, and exploit the fact that any generalisations of these are automatically unacceptable as well. E.g., if a(b,c,*) is acceptable, then our algorithm considers its generalisation a(*,c,*); however, if it already knows that a(*,c,d) is unacceptable, then this new generalisation need not be considered. This further optimisation is not implemented in our algorithm.

Algorithm 3 gl-testable

Input: A set T of positive examples and a positive integer k
Output: A tree automaton (V, Q, Δ, FS)

$\mathcal{F}_0 := \bigcup\{f_k(t) \mid t \in T\}$
$\mathcal{TF} := \{f \in \mathcal{F}_0 \mid f \text{ contains } x\}$
$\mathcal{OF} := \mathcal{F}_0 - \mathcal{TF}$
$\mathcal{F} = \mathcal{TF}$
$\mathcal{P} = partition(\mathcal{OF})$
$FS := \{r_{k-1}(t) \mid t \in T\}$
$\mathcal{S} := \bigcup\{s_{k-1}(t) \mid t \in T\}$
$\mathcal{TS} = \{s \in \mathcal{S} \mid s \text{ contains } x, \text{ height}(s) = k-1\}$
for each $\mathcal{C} \in \mathcal{P}$ **do**
  $c = pgen(\mathcal{C})$ % candidate generalization
  if c covers one of $\mathcal{TF}$ **or** one of $children(c)$ covers one of $\mathcal{TS}$ **then**
    $\mathcal{F} = \mathcal{F} \cup genl(\mathcal{TS}, \mathcal{TF}, \mathcal{C})$
  else
    $\mathcal{F} = \mathcal{F} \cup genl(\mathcal{TS}, \mathcal{TF}, \{c\})$
  end if
end for
$Q := \mathcal{S} \cup FS \cup \{r_{k-1}(f) \mid f \in \mathcal{F}\} \cup \bigcup\{s_{k-1}(f) \mid f \in \mathcal{F}\}$
$\Delta := \{(v, t_1, \ldots, t_m) \to v(t_1, \ldots, t_m) \mid v(t_1, \ldots, t_m) \in \mathcal{S}\}$
$\Delta := \Delta \cup \{(v, t_1, \ldots, t_m) \to r_{k-1}(v(t_1, \ldots, t_m)) \mid v(t_1, \ldots, t_m) \in \mathcal{F}\}$

Algorithm 4 genl

Input: Sets $\mathcal{TS}$ of target subtrees, $\mathcal{TS}$ of target forks, and T of trees
Output: A set of trees G (a generalisation of T)

$G := \emptyset$
while $T \neq \emptyset$ **do**
  select t from T and remove it
  $C := gentree(\mathcal{TS}, \mathcal{TF}, t)$
  if $C = \emptyset$ **then**
    $G := G \cup \{t\}$
  else
    $T := T - \{t \mid t \in T \text{ and } t \text{ is covered by some } c \in C\}$
    $T := T \cup C$
  end if
end while

Algorithm 5 gentree

Input: Sets $\mathcal{TS}$ of target subtrees, $\mathcal{TF}$ of target forks, and a tree t
Output: A set of trees C

$C_0 := \{t' \mid t' \text{ is derived from } t \text{ by replacing one bottommost label} \neq * \text{ by } *\}$
$C := C_0 - \{c \in C_0 \mid c \text{ covers one of } \mathcal{TF}, \text{ or one of } children(c) \text{ covers one of } \mathcal{TS}$

- $\mathcal{F} = \{a(b(*), c),\ b(a(b, x))\}$
- $Q = \{a(b, c),\ b(a),\ b(*),\ a(b, x),\ b,\ x,\ c\}$
- *transitions:*
  - $a(b, x) \in \mathcal{S} : (a, b, x) \rightarrow a(b, x)$
  - $b \in \mathcal{S} : (b) \rightarrow b$
  - $x \in \mathcal{S} : (x) \rightarrow x$
  - $c \in \mathcal{S} : (c) \rightarrow c$
  - $a(b(*), c) \in \mathcal{F} : (a, b(*), c) \rightarrow a(b, c)$
  - $b(a(b, x)) \in \mathcal{F} : (b, a(b, x)) \rightarrow b(a)$

Note again that the use of wildcards in the representation of the sets $\mathcal{F}$, Q and Δ in the example is really just an abbreviation; e.g., when Q contains $b(*)$, this really means it contains the states $b(a)$, $b(b)$, $b(c)$ and $b(x)$.

4 Experimental Results

4.1 Test on the benchmark datasets

We evaluated our method on some semi-structured data sets commonly used in the IE research[2]: a collection of web pages containing people's contact addresses (the Internet Address Finder (IAF) database) and a collection of web pages about stock quotes (the Quote Server (QS) database). There are 10 example documents in each of these datasets. The number of fields to be extracted is respectively 94 (IAF-organization), 12 (IAF-alt.name), 24 (QS-date), and 25 (QS-vol). The motivation to choose these datasets is

[2]Available from `http://www.isi.edu/~muslea/RISE/`.

as follows. Firstly they are benchmark datasets that are commonly used for research in information extraction, so we can compare the results of our method directly with the results of other methods. Secondly, they are the most difficult benchmarks we are aware of that require the extraction of a whole node of the document tree. In fact, one of the authors in [40] has tried to build a handcrafted extractor given all available documents from the QS dataset and achieved only 88% accuracy.

We also test the k-testable, g-testable, and gl-testable algorithms on a small and simplified Shakespeare[3] data set, which is a significantly reduced version of Jon Bosak's Shakespeare XML dataset.[4] We use it to test the feasibility of our methods. In this dataset, the task is to extract the second scene of every act in a particular play. The motivation to test on this data set is that we believe that the extraction task is very difficult for string-based methods even on the simplified data. This is because each scene has a complex structure of varying length. We used the simplified Shakespeare dataset because our three algorithms above performed very poorly on the full dataset. (The latter is a consequence of the conversion to ranked trees: an "act" field, for instance, can have many children, many of which precede the second scene of the act. After the conversion, this second scene then ends up very deep in the subtree below the "act" tag, making it very difficult to identify using a k-testable automaton.) Thus, we expect the simplified data set to be a good example of a task that is difficult for string-based wrappers but manageable for tree-based ones.

The training and the testing processes follow the procedures outlined in Section 3.3. For evaluating our method, we use criteria that are commonly used in the information

[3] Available from `http://www.cs.kuleuven.ac.be/~raymond/ie/`.
[4] Available from `http://www.ibiblio.org/bosak/`.

retrieval research community. Precision P is the number of correctly extracted objects divided by the total number of extractions, while recall R is the number of correct extractions divided by the total number of objects present in the answer template. The F1 score is defined as $2PR/(P+R)$, the harmonic mean of P and R. The precision, recall and F1 scores are calculated for each extracted slot, without collecting them in a case frame beforehand.

Table 1 shows the results we obtained as well as those obtained by some current state-of-the-art string-based methods: an algorithm based on Hidden Markov Models (HMMs) [20], the Stalker wrapper induction algorithm [41] and BWI [19]. We also include the results of the k-testable algorithm (as we reported in [31]) and the g-testable algorithm (as we reported in [29]). The results of HMM, Stalker and BWI are adopted from [19]. All tests are performed with 10-fold cross validation following the splits used in [19], except in the small Shakespeare dataset where 2-fold cross validation was used. Each split has 5 documents for training and 5 for testing. We refer to the related work section for a brief description of the other methods.

Table 1 shows the results of the k-testable, g-testable and gl-testable algorithms for the optimal k value (more specifically, k was optimised during the first fold of the cross-validation, this optimal value was then used for all the other folds). As can be seen, our methods perform better in most of the test cases than the existing state-of-the-art string-based methods. The only exception is the field date in the QS dataset where BWI performs better. Compared to the results of k-testable, the gl-testable algorithm performs better in the IAF-alt.name, IAF-organization and small Shakespeare data. Compared to the results of g-testable, the gl-testable performs better in the IAF-alt.name and IAF-organization data but worse in the small Shakespeare data. We shall discuss these results

Table 1: Comparison of the results

	IAF - alt.name			*IAF - organization*			*QS - date*			*QS - volume*		
	Prec	*Rec*	*F1*	*Prec*	*Rec*	*F1*	*Prec*	*Rec*	*F1*	*Prec*	*Rec*	*F1*
HMM	1.7	90	3.4	16.8	89.7	28.4	36.3	100	53.3	18.4	96.2	30.9
Stalker	100	-	-	48.0	-	-	0	-	-	0	-	-
BWI	90.9	43.5	58.8	77.5	45.9	57.7	100	100	100	100	61.9	76.5
k-testable	100	73.9	85	100	57.9	73.3	100	60.5	75.4	100	73.6	84.8
g-testable	100	73.9	85	100	82.6	90.5	100	60.5	75.4	100	73.6	84.8
gl-testable	100	84.8	91.8	100	84.6	91.7	100	60.5	75.4	100	73.6	84.8

	Small Shakespeare		
	Prec	*Rec*	*F1*
k-testable	56.2	90	69.2
g-testable	66.7	80	72.7
gl-testable	66.7	80	72.7

Table 2: Parameters used for the experiments

	IAF - alt.name	*IAF - org.*	*QS - date*	*QS - volume*	*Shakespeare*
k-testable (k)	4	4	2	5	3
g-testable (k, l)	(5,2)	(5,2)	(3,2)	(6,5)	(4,2)
gl-testable (k)	4	4	2	6	4

below.

Table 2 shows the parameters k, (k, l) and k that were used by the k-testable, g-testable, and gl-testable algorithms respectively to produce the results in Table 1. A distinguishing context was used in the datasets IAF-alt.name and IAF-organization.

In these datasets some of the best results with the gl-testable algorithm (i.e. in QS-volume and small Shakespeare data) are obtained with a k value bigger than the value used in the k-testable algorithm. This means that our goal, performing more generalisation while using bigger contexts, is achieved. Some other best results (*i.e.*, in IAF-alt.name, IAF-organization and QS-date data) are obtained with using the same k value. These results indicate that: (1) The wildcards are useful for our IE tasks as they can improve the results of the k-testable algorithm. (2) The two step generalisation, done by the procedure *pgen* that generalises all forks in each partition and by the procedure *genl* that searches the generalisation of the bottommost labels more thoroughly, is useful for our IE tasks.

This is shown by the better results of the gl-testable algorithm compared to the results of the g-testable algorithm in the two IAF datasets that are obtained with a smaller value of k.

Despite the improvements of both gl-testable and g-testable algorithms in IAF-alt.name, IAF-organization and small Shakespeare datasets, they were not able to improve the results of the k-testable algorithm in the two QS datasets. For the QS-volume data, the reason is not clear to us. One explanation is that the result might be optimal for these learners given a certain set of training examples. For the QS-date data, the reason is that the inferred automaton is not general enough, as can be seen in Figure 3. In that figure, the recall of the most general automaton inferred ($k = 2$) is not very high and the maximum precision is already reached with $k = 2$. Thus we cannot improve the F1 score by increasing the k.

The gl-testable algorithm performs worse than the g-testable algorithm in the small Shakespeare data, although better in IAF-alt.name and IAF-organization datasets. The reason is that, besides optimising the k parameter, we also optimise the generalisation level of the g-testable algorithm by cross validation to suit a specific dataset. In other words, the g-testable algorithm has an extra parameter to optimise and is more heuristic in nature than the gl-testable algorithm.

Figure 3 shows how the F1 score of the gl-testable algorithm changes with k. The solid line is the F1 score, the dotted line precision and the dashed line recall. In this figure, we can clearly see the trade off between precision and recall. Actually, the behavior of the three tree-based algorithms that we test is quite similar. With small value of k, the precision of these tree-based methods tend to be low because the automaton inferred is relatively general. As the value of k increases, the precision rises until a certain value of k

where the maximum precision is reached. The recall of these tree-based methods behaves the other way around. At the low value of k, the recall of these tree-based methods tends to be high. However, as the value of k becomes higher, the recall decreases gradually. As the harmonic mean of the precision and recall, typically the F1 score curve starts with a low value at $k = 2$, increases, reaching a maximum and then starts to decrease.

Figure 4 shows the average training time of the k-testable and the gl-testable algorithms for different values of k. Overall, both algorithms show somewhat similar training time on our datasets. By using a suitable data structure for looking up forks and subtrees, the k-testable algorithm can be made to run in $O(kn)$ time, where n is the total size of the examples. A similar upper bound holds for the time complexity of the g-testable algorithm. The theoretical running time of the gl-testable algorithm is in the worst case exponential in the size of the subtrees, in the worst case, due to the finer search in the generalisation lattice. In our experiments, the gl-testable algorithm is still feasible to run if the k value used is less than 8. The preprocessing consists of parsing, conversion to the binary tree representation (both processes take time linear in the size of the document) and the manual insertion of the label x. Our prototype implementation was implemented in Prolog and tested on a Pentium 1.7 Ghz PC. The figure shows that the actual training time (after preprocessing) needed to infer the automaton is more or less linear in k. One exception is the training time of the gl-testable algorithm for IAF-alt.name which looks non-linear. The reason is that with $k < 6$ most candidate generalizations (the result of *pgen* function) do not suffer from overgeneralisation. Thus only one candidate generalization is input to the *genl*. This is also the reason why in this task the gl-testable algorithm is slightly faster than the k-testable algorithm for $k < 6$. At $k = 6$ several candidate generalizations suffer from overgeneralisation.

Actually the theoretical training time of the k-testable and the g-testable algorithms is better than that of BWI [19], one of the string-based methods that are used for comparison. The training time of BWI increases exponentially with the increase of the look-ahead parameter. As reported in [19] the IAF-alt.name, IAF-organization and QS-volume datasets need a long lookahead. They used lookaheads of 8 because these tasks need very long boundary detectors. We cannot compare the actual training time of BWI to ours in these datasets, as it was not reported.

Figure 5 shows the average extraction time per document of the gl-testable algorithms for different values of k. The theoretical time complexity of the extraction procedure is $O(n^2)$ where n is the number of nodes in the document. Indeed, the time of a single run is linear in the number of nodes (using suitable data structures), while the automaton has to run for each replacement of a node by the target symbol x. Just for comparison, the extraction time of the tree automaton inferred by the k-testable algorithm (not shown here) is about two times faster than the extraction time of the generalised automaton inferred by the gl-testable algorithm. The reason is that the latter automaton needs additional time to match the wildcards.

Figure 6 shows the number of states inferred by the k-testable and gl-testable algorithms for different values of k. As we can see from the figure, the number of states inferred by the gl-testable algorithm is always smaller than the number of states inferred by the k-testable algorithm for $k > 2$. For $k = 2$, the number of states inferred is equal as the gl-testable algorithm performs no generalisation in this case.

Figure 3: The graphs of F1 score versus k

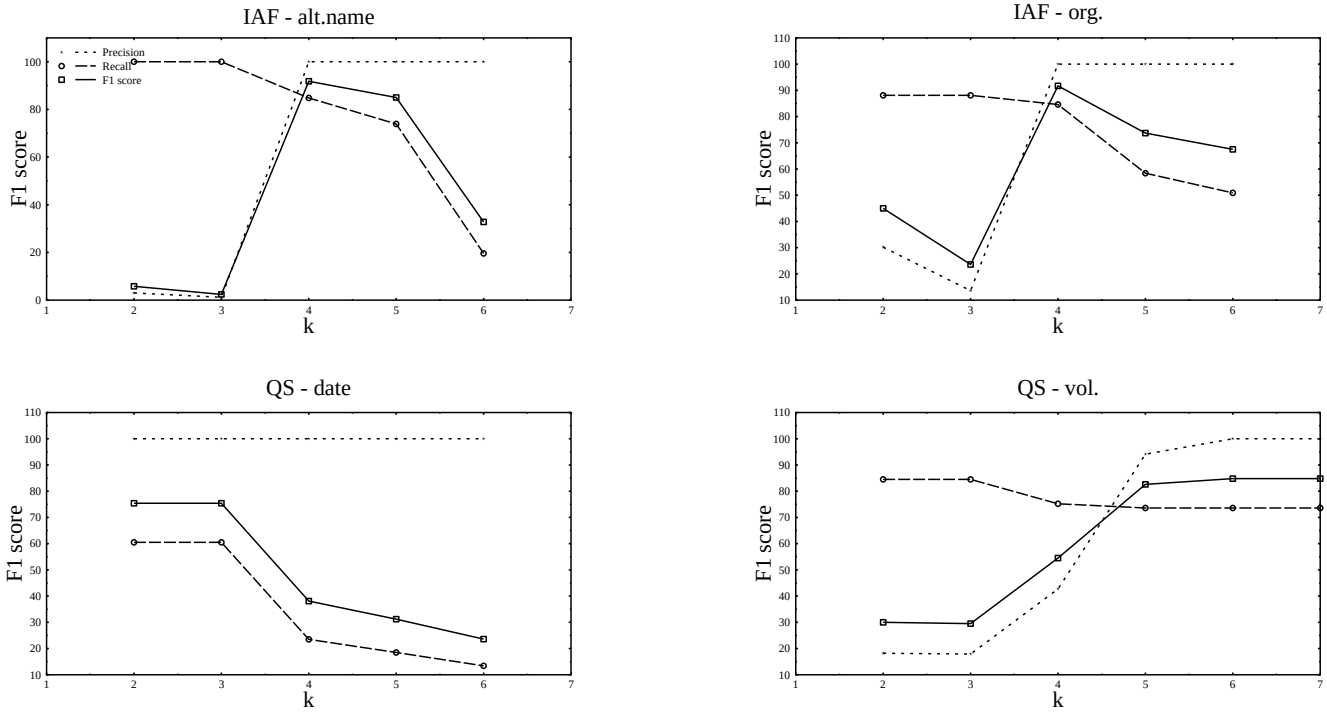


Figure 4: The graphs of training time versus k.

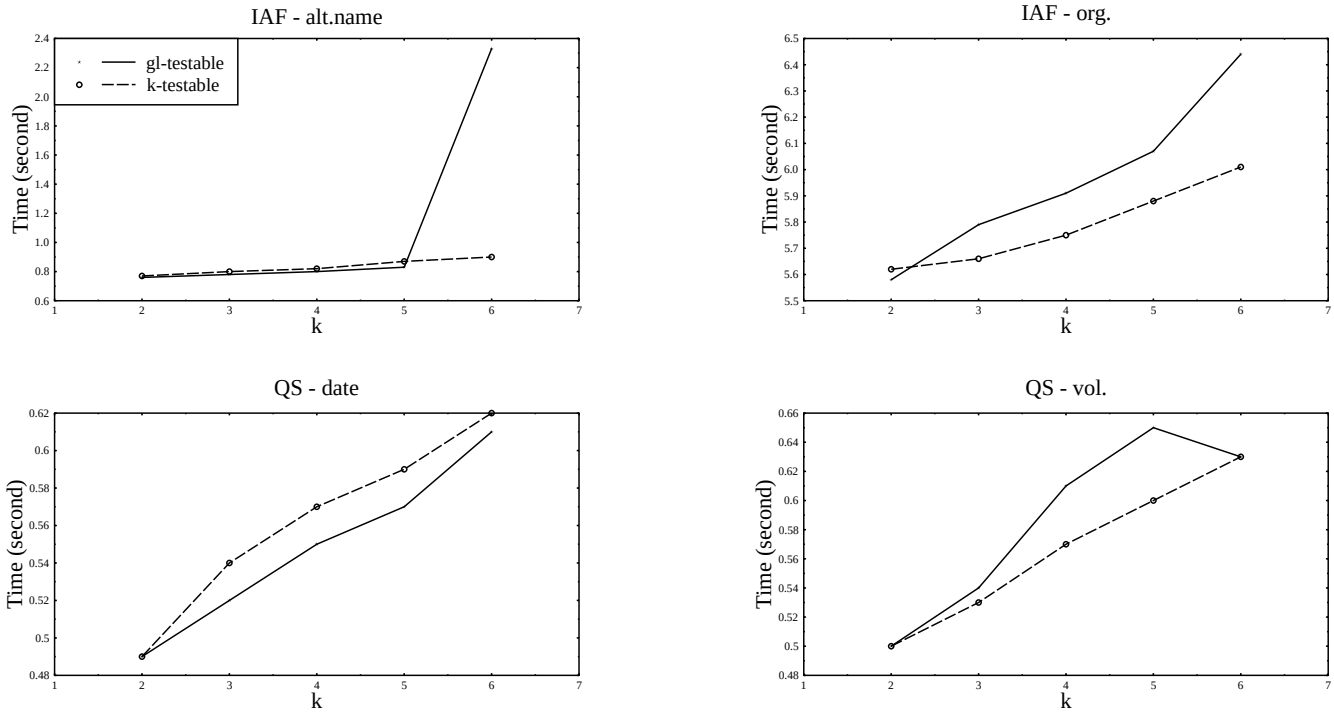


Figure 5: The graph of extraction time versus k, for the gl-testable algorithm.

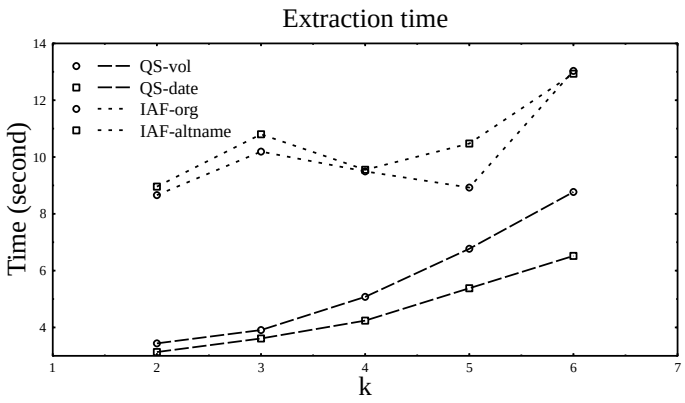
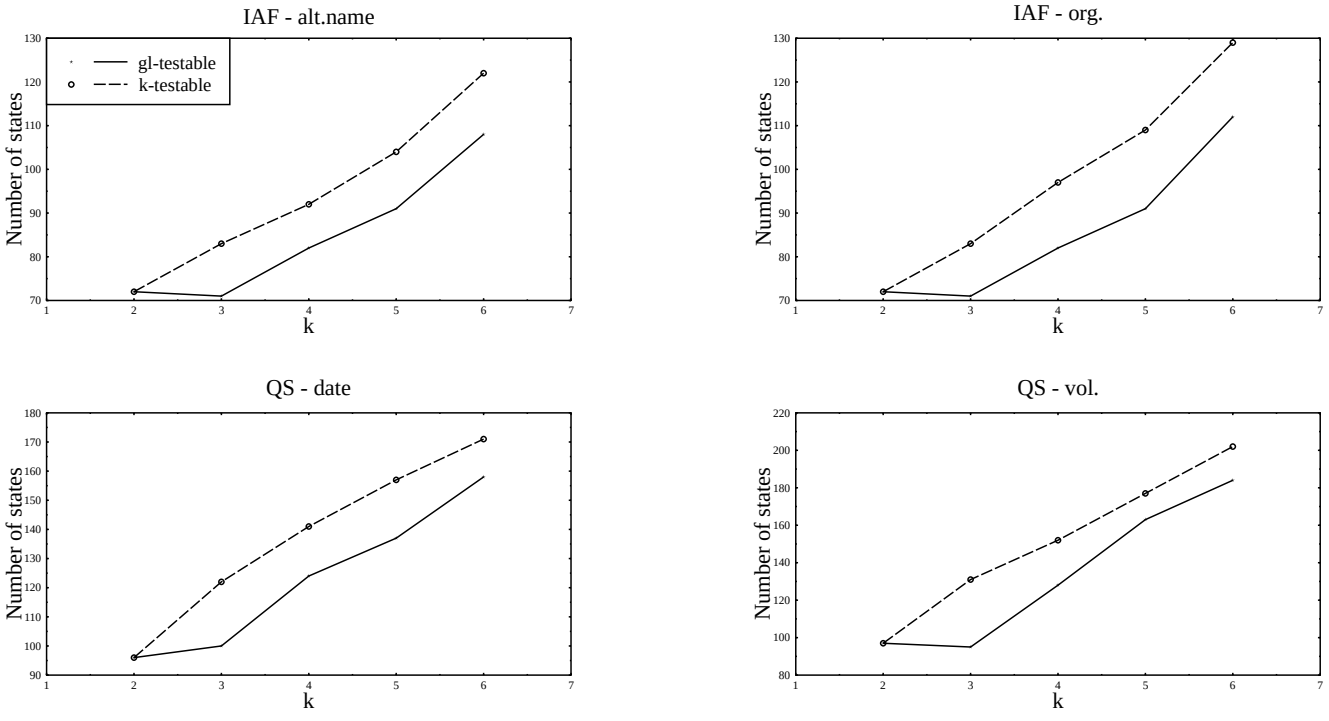


Figure 6: The graphs of the number of states versus k.



4.2 Test on larger datasets

To evaluate the quality of our approach, we test the gl-testable algorithm also on larger datasets. For the experiments below we use the Bigbook and the Okra datasets that are also available online. In the Bigbook dataset we train the automaton to extract the 'name' and 'address' fields, and in the Okra dataset we train the automaton to extract the 'name' and 'email' fields. The Bigbook and Okra datasets contains 235 and 252 files respectively. The number of the name and address fields to be extracted from the Bigbook dataset is 4299 while the number of the name and email fields in the Okra datasets is 3334.

All experiments in this section are done with 10-fold cross-validation and the experimental setting is as follows. We divide both datasets in two parts. The first part, consisting of ten files (documents), is used to test the generalization ability of the gl-testable algorithm. From these ten documents we obtain 184 and 120 examples from the Bigbook and Okra datasets respectively. First we determine for each extraction task the optimal k by cross-validation in one random fold of training and test set. The best k found is 5 for the tasks in the Bigbook dataset and 4 for the tasks in the Okra dataset. Then, from the same set of 184 (120) examples, we take randomly ten examples, then twenty examples, then thirty, and so on, and give them as training examples while the rest is for testing. This process is performed ten times for every extraction task. We stop when the induced automaton has an average F1 score of 98% or better on the test examples. Table 3 shows the number of examples needed for good generalization. We can see that the number of examples needed to learn a good wrapper for these datasets is quite small.

In the ten documents of the Bigbook dataset in Table 3, the training time for each document ranges from 0.11 to 0.12 seconds and the average testing time for each document

Table 3: The number of examples needed for good generalization

	k	*#documents*	*#fields*	*#examples*	*Prec*	*Recall*	*F1*
Bigbook-name	5	10	184	40	100	98.7	99.3
Bigbook-address	5	10	184	40	100	99.1	99.5
Okra-name	4	10	120	10	100	100	100
Okra-email	4	10	120	10	100	100	100

is 11.24 seconds (the average number of nodes in a document ≈ 513). In the ten documents of the Okra dataset in Table 3, the training time for each document ranges from 0.26 to 0.54 seconds and the testing time for each document varies from 0.08 to 38.67 seconds (the average number of nodes in a document ranges from 66 to 1299 respectively).

The second part of the experiment is to test the quality of the obtained automata. The data in the second part consists of 225 and 242 documents from the Bigbook and Okra datasets respectively. Using the learned tree automata wrappers, we perform extraction on the remaining documents in the data set. Note that none of these documents was used during the learning. Table 4 shows the results. The results for the Okra-name and Okra-email datasets are very good, considering that it used only 10 examples for learning. However, the results for the Bigbook-name and Bigbook-address datasets are not as good. The reason is that the tree automaton is sensitive to the small variability in the document tree, even after generalization. In the bigbook data, there is an index in every page that enable the user to 'jump to' the first company name beginning with a certain alphabet. This index sometimes contains full links but sometimes only partial links. The variability in this bottom part of the document is creating new states that were not seen before by the tree automaton, causing failure of the extraction task.

Hsu and Chang [26] list the performance of other systems on these datasets. In the Bigbook dataset, Stalker [41] achieves 97% recall (or accuracy in their definition) with 8 examples, WIEN [32] achieves 100% recall using an average of 15 documents containing

Table 4: The test on the rest of the larger datasets

	k	*#documents*	*#fields*	*Prec*	*Recall*	*F1*
Bigbook-name	5	225	4115	100	70.5	82.7
Bigbook-address	5	225	4115	100	71.7	83.5
Okra-name	4	242	3214	100	97	98.5
Okra-email	4	242	3214	100	97	98.5

approximately 274 examples, and SoftMealy [26] achieves 100% recall given 6 examples. In the Okra dataset, Stalker achieves 97% recall (or accuracy in their definition) with only 1 example, WIEN achieves 100% recall using an average of 3.5 documents containing approximately 46 examples, and SoftMealy achieves 100% recall given 1 example. These results cannot be compared rigorously with ours because the above wrapper systems extract 6 fields from the Okra and 4 fields from the Bigbook dataset, while our system was only tested on 2 fields from each dataset. Still, the comparison makes clear that the gl-testable algorithm needs more examples to learn from than the string-based methods we compare with. This is not unexpected: our tree-based methods search a larger hypothesis space, looking also for patterns further away from the field to be extracted, whereas the other methods look for patterns that narrowly enclose this field. As our methods consider more possible hypotheses, they need more examples to eliminate the incorrect ones.

5 Related work

A lot of methods have been used for IE problems. Many are described in [39, 51, 33]. As mentioned above, the work on IE can be classified into three main categories: IE from unstructured texts, IE from semi-structured texts and IE from structured texts. Within each of these categories, the work can be further divided into manually built systems and (semi-)automatic systems. Within the domain of extraction from structured documents,

the work on IE can be divided into:

- Manual systems. Examples of manually built systems can be found in [3, 24]. They apply knowledge engineering techniques for building wrappers. Manually building a wrapper for each data source becomes infeasible when confronted with the variety of Web sources. A separate area aims at the development of query languages for HTML/XML, e.g. [5, 57]. While these query languages are suitable for expressing complex extraction problem, their use remains time consuming and requires non-trivial skill. The advantage over other manual approaches is that they provide the user with a sophisticated user interface that simplifies the wrapper specification process as the user neither need the ability to program nor to know the HTML syntax. Most of the work in this area originated from the database community, other work originated from the document and logic programming communities. Some recent systems developed in this area are W4F [47], XWrap [36], and Lixto [4]. Related to this work is the large body of work on Web information integration. This addresses the problem of integrating heterogeneous data on the Web with the purpose of allowing users to pose queries to these integrated data. A typical information integration system consist of: wrappers that transform data from the original sources into a form that can be further processed by the system, a mediator consisting of a query planner and an execution engine, and a user interface for entering queries. Some examples of information integration systems are: Tsimmis [9], and Jedi [28].

- Semi-automatic systems. Our work is situated in the domain of semi-automatically built systems for IE from structured documents. Such systems make use of machine

learning and data mining techniques, as well as other algorithms. The process is known as *wrapper induction*. With the recent attention for the Web, this line of work has been more popular than the work on manually-built IE system. It can be noted, that some wrapper induction systems (not ours) are also able to work on semi-structured data and even on unstructured texts.

- Automatic systems. We classify an IE system as an automatically-built system if the wrapper is built only once and can be used for new extraction tasks directly, or if wrappers can be built for each new task using unsupervised training only. Some examples of the IE systems in this category are as follows. WHIRL [11] is a 'soft' logic system that incorporates a notion of textual similarity developed in the information retrieval community. WHIRL has been used to implement some heuristics that are useful for IE in [13]. Hemnani and Bressan [25] proposed a tree alignment algorithm that are based on two heuristics for extracting multiple record Web documents. IEPAD [8] is a system that automatically discovers extraction rules for identifying record boundaries from web pages.

Other work, *e.g.* [54, 42], on Web structure mining aims at finding structural similarity between web pages. It is known as schema discovery and DTD inference and is different in nature from ours. They aim at mining the frequent or common structure of web pages. The problem that we are facing goes beyond finding the common structure of web documents. We also try to find the pattern of the field to be extracted inside the common document structure.

In what follows, we restrict our attention to work on wrapper induction techniques that, similar to our work, use machine learning or data mining techniques for IE from

structured texts. For classical IE and the issues of IE from unstructured text we refer to *e.g.* [6, 15]. For IE work on semi-structured texts, we refer to recent reviews such as [39, 51, 33].

The term wrapper induction was first introduced in [34]. As mentioned in the introduction, much work on wrapper induction learns wrappers based on regular expressions. BWI [19] is basically a boosting approach in which the weak learner learns a simple regular expression with high precision but low recall. The HMM approach reported in Table 1 was proposed by Freitag and McCallum [20]. They learn a hidden Markov model, solving the problem of estimating probabilities from sparse data using a statistical technique called shrinkage. This model has been shown to achieve state-of-the-art performance on a range of IE tasks. The Stalker algorithm [41] induces extraction rules that are expressed as simple landmark grammars. The latter are a class of finite automata. Stalker performs hierarchical extraction guided by a manually built *embedded catalog* tree. This tree describes the structure of the fields to be extracted from the documents.

Freitag [16] describes several techniques based on naive-Bayes, two regular language inference algorithms, and their combinations for IE from unstructured texts. His results demonstrate that the combination of grammatical inference techniques with naive-Bayes improves the precision and accuracy of the extraction. WHISK [51] is a system that learns extraction rules with a top-down rule induction technique. The extraction rules of WHISK are based on a kind of regular expression patterns. To make the rules more powerful, WHISK has some built-in semantic classes and in addition allows for user-defined semantic classes. A semantic class is basically a set of terms that are considered to be equivalent. Chidlovskii et al. [10] describe an incremental grammar induction approach; their language is based on a subclass of deterministic finite automata that do

not contain cyclic patterns. Hsu and Dung's SoftMealy system [27] learns separators that identify the boundaries of the fields of interest. These separators are described by strings of fixed height in which each symbol is an element of a taxonomy of tokens (with fixed strings on the lowest level and concepts such as punctuation or word at higher levels). Hsu and Chang [26] propose two classes of SoftMealy extractors: single-pass, which is biased for tabular documents such as the QS dataset, and multi-pass, which is biased for tagged-list document such as the IAF dataset. Although their systems were tested on the same datasets as ours, their results cannot be compared directly because the experimental setting is different. Their evaluation gives only numbers for recall and uses a different set of examples.

The above mentioned methods learn string languages while our method learns a more expressive tree language. Compared to HMMs and BWI our method does not require the manual specification of the windows height for the prefix, suffix and the target fragments. Compared to Stalker and BWI our method does not require the manual specification of the special tokens or landmarks such as ">" or ";". Compared to Stalker our method works directly on document trees without the need for manually building the *embedded catalog* tree. Compared to SoftMealy extractors in [26] our method is generally applicable to any type of document formatting without requiring different classes of wrappers for different categories of documents.

Despite the above advantages, our method also has some limitations. A first one is that our method only outputs a whole node. This seems to limit its applicability. For data-centric documents such as XML documents, this is not really the case since the data to be extracted is typically a whole node. However, it is true for HTML formatted documents. One way to broaden the applicability of our method is to perform a two step

extraction. A whole node of the tree can be extracted in a first step while a second step (using other techniques) can post-process the selected information to extract a part of it. A second limitation is that our method works only on structured documents. Indeed our method cannot be used for text-based IE, and is not intended for it. A third limitation is that our method is possibly slower (when extracting) than string-based methods because it has to parse the document tree and has to substitute each node with x when extracting information from the document. Despite these limitations, our results suggest that our method works better in the four structured domains than the more generally applicable string-based IE methods.

Some other approaches that exploit the structure of the documents have been proposed. WL^2 [12], a logic-based wrapper learner that uses multiple (string, tree, visual, and geometric) representations of the HTML documents, consists of one master builder and several specific builders that are created for specific page formats. The learning method is an inductive logic programming algorithm based on [45]. In fact, WL^2 is able to extract all four tasks in the IAF and QS datasets with 100% recall. The work of WL^2 suggests that indeed using task-specific document representation can yield a much better performance. Sakamoto et al. [50] propose a certain class of wrappers that use the tree structure of HTML documents and propose an algorithm for inducing such wrappers. They identify a field with a path from root to leaf, imposing conditions on each node in the path that relate to its label and its relative position among siblings with the same label (e.g., "2nd child with label `<B>`"). Their hypothesis language corresponds to a subset of tree automata.

Besides the k-testable algorithm proposed in this paper, we have also experimented with Sakakibara's reversible tree algorithm [48]. Preliminary results with this algorithm

suggested that it generalises insufficiently on our data sets. Hence, we did not further pursue its use.

6 Conclusion

The main contributions of this paper can be summarised as follows:

- We have motivated and presented a novel method that uses tree automata induction for information extraction from structured documents. Besides using the tree structure, this approach also has other advantages compared to string-based and other methods. Firstly, some IE systems preprocess documents to split them up in small fragments and use only a part of the document as training example. This is not needed here as the tree structure that we get for free takes care of this. Thus the entire document tree can be used as training example. Secondly, our method does not require the manual specification of a window length for the prefix, suffix and target fragments, and of the special tokens or landmarks such as ">" or ";", that are usually required by the string-based methods. Thus our method requires very little user intervention.

- We proposed several generalisations of the original k-testable algorithm. We have shown that these generalisations perform as good as or better than the original k-testable algorithm for structured IE from our datasets.

- We have demonstrated on two benchmark datasets that our method can perform better than string-based methods. These results suggest that it is worthwhile to consider our approach when confronted with difficult IE tasks with structured doc-

uments. On two other data sets, where the tree structure of the document is unimportant, our results are less good than those of some string-based methods. This confirms that it remains important to select a method with an appropriate bias, when inducing wrappers.

The use of automata learned from ranked trees requires the conversion of unranked trees to binary trees. A drawback is that the distance between target node and distinguishing context is increasing. This suggests that it is worthwhile to consider algorithms that can infer an automaton directly from unranked trees. We plan to explore this possibility in the near future; a start has been made in [30]. Such an algorithm may also allow us to address more general IE tasks on XML documents. Indeed, until now we have only performed experiments on standard benchmark IE tasks that can also be performed by the previous string-based approaches. However, there are tasks that seem clearly beyond the reach of string-based approaches (and our algorithms based on ranked trees). An example is the extraction of the second item from a list of items, where every item itself may have a complex substructure as in the *full* Shakespeare XML dataset.

Another direction to explore is to incorporate probabilistic information in the inference process to compensate for the lack of negative examples and to combine unstructured text extraction with structured document extraction.

Acknowledgements

We thank Nicholas Kushmerick for providing us with the datasets used for the BWI experiments. This work is supported by the FWO project query languages for data mining. Hendrik Blockeel is a post-doctoral fellow of the Fund for Scientific Research of

Flanders.

References

[1] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[2] D. Angluin and C. H. Smith. Inductive inference: Theory and methods. *ACM Computing Surveys*, 15(3):237–269, 1983.

[3] P. Atzeni and G. Mecca. Cut & paste. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 144–153. ACM Press, 1997.

[4] R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with lixto. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB 2001)*, pages 119–128, 2001.

[5] F. Bry and S. Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *Proceedings of the International Conference on Logic Programming*, 2002.

[6] C. Cardie. Empirical methods in information extraction. *AI Magazine*, 18(4):65–79, 1997.

[7] R. C. Carrasco, J. Oncina, and J. Calera-Rubio. Stochastic inference of regular tree languages. In *Proceedings of the 3rd International Colloquium on Grammatical Inference*, Lecture Notes on Articial Intelligence 1433, pages 187–198, 1998.

[8] C.-H. Chang and S.-C. Lui. IEPAD: Information extraction based on pattern discovery. In *Proceedings of the tenth International Conference on World Wide Web*, pages 681–688, 2001.

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

[10] B. Chidlovskii, J. Ragetli, and M. de Rijke. Wrapper generation via grammar induction. In *11th European Conference on Machine Learning, ECML'00*, pages 96–108, 2000.

[11] W. Cohen. Whirl: A word-based information representation language. *Artificial Intelligence*, 118:163–196, 2000.

[12] W. Cohen, M. Hurst, and L. S. Jensen. A flexible learning system for wrapping tables and lists in HTML documents. In *The Eleventh International World Wide Web Conference (WWW2002)*, 2002.

[13] W. W. Cohen. Recognizing structure in web pages using similarity queries. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on on Innovative Applications of Artificial Intelligence*, pages 59–66, 1999.

[14] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications.* Available on: http://www.grappa.univ-lille3.fr/tata, 1999.

[15] J. Cowie and W. Lehnert. Information extraction. *Communications of the ACM*, 39(1):80–91, 1996.

[16] D. Freitag. Using grammatical inference to improve precision in information extraction. In *ICML-97 Workshop on Automata Induction, Grammatical Inference, and Language Acquisition*, 1997.

[17] D. Freitag. Information extraction from HTML: Application of a general learning approach. In *Proceedings of the Fifteenth Conference on Artificial Intelligence AAAI-98*, pages 517–523, 1998.

[18] D. Freitag. Machine learning for information extraction in informal domains. *Machine Learning*, 39(2/3):169–202, 2000.

[19] D. Freitag and N. Kushmerick. Boosted wrapper induction. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Innovative Applications of AI Conference*, pages 577–583. AAAI Press, 2000.

[20] D. Freitag and A. McCallum. Information extraction with HMMs and shrinkage. In *AAAI-99 Workshop on Machine Learning for Information Extraction*, 1999.

[21] P. Garcia. Learning k-testable tree sets from positive data. Technical report, Technical Report DSIC-ii-1993-46, DSIC, Universidad Politecnica de Valencia, 1993.

[22] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[23] G. Gottlob and K. Koch. Monadic datalog over trees and the expressive power of languages for web information extraction. In *21st ACM Symposium on Principles of Database Systems*, pages 17–28, 2002.

[24] J. Hammer, H. Garcia-Molina, J. Cho, A. Crespo, and R. Aranha. Extracting semistructured information from the Web. In *Proceedings of the Workshop on Management of Semistructured Data*, pages 18–25, 1997.

[25] A. Hemnani and S. Bressan. Information extraction - tree alignment approach to pattern discovery in web documents. In *Database and Expert Systems Applications, 13th International Conference, DEXA 2002*, pages 789–798, 2002.

[26] C.-N. Hsu and C.-C. Chang. Finite-state transducers for semi-structured text mining. In *Proceedings of IJCAI-99 Workshop on Text Mining: Foundations, Techniques and Applications*, 1999.

[27] C.-N. Hsu and M.-T. Dung. Generating finite-state transducers for semi-structured data extraction from the Web. *Information Systems*, 23(8):521–538, 1998.

[28] G. Huck, P. Fankhauser, K. Aberer, and E. J. Neuhold. Jedi: Extracting and synthesizing information from the web. In *Conference on Cooperative Information Systems*, pages 32–43, 1998.

[29] R. Kosala, M. Bruynooghe, H. Blockeel, and J. Van den Bussche. Information extraction by means of a generalized k-testable tree automata inference algorithm. In *Proceedings of the Fourth International Conference on Information Integration and Web-based Applications & Services (IIWAS)*, pages 105–109, 2002.

[30] R. Kosala, M. Bruynooghe, J. Van den Bussche, and H. Blockeel. Information extraction from web documents based on local unranked tree automaton inference. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, 2003. To appear.

[31] R. Kosala, J. Van den Bussche, M. Bruynooghe, and H. Blockeel. Information extraction in structured documents using tree automata induction. In *Proceedings of the 6th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, pages 299–310, 2002.

[32] N. Kushmerick. *Wrapper Induction for Information Extraction.* PhD thesis, University of Washington, 1997.

[33] N. Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118:15–68, 2000.

[34] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Proceedings of the International Joint Conference on Artificial Intelligence IJCAI-97*, pages 729–737, 1997.

[35] A. Levy, C. Knoblock, S. Minton, and W. Cohen. Trends and controversies: Information integration. *IEEE Intelligent Systems*, 13(5), 1998.

[36] L. Liu, C. Pu, and W. Han. Xwrap: An XML-enabled wrapper construction system for web information sources. In *Proceedings of the 16th International Conference on Data Engineering*, pages 611–621. IEEE Computer Society, 2000.

[37] MUC-6. *Proceedings of the Sixth Message Understanding Conference.* San Francisco, CA: Morgan Kaufmann, 1995.

[38] K. Murphy. Learning finite automata. Technical Report 96-04-017, Santa Fe Institute, 1996.

[39] I. Muslea. Extraction patterns for information extraction tasks: A survey. In *AAAI-99 Workshop on Machine Learning for Information Extraction*, 1999.

[40] I. Muslea, S. Minton, and C. Knoblock. A hierarchical approach to wrapper induction. In *Proceedings of the 3rd International Conference on Autonomous Agents*, 1999.

[41] I. Muslea, S. Minton, and C. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Journal of Autonomous Agents and Multi-Agent Systems*, 4:93–114, 2001.

[42] S. Nestorov, S. Abiteboul, and R. Motwani. Infering structure in semistructured data. *SIGMOD Record*, 26(4), 1997.

[43] C. Pair and A. Quere. Définition et etude des bilangages réguliers. *Information and Control*, 13(6):565–593, 1968.

[44] R. Parekh and V. Honavar. *Automata Induction, Grammar Inference, and Language Acquisition.* Handbook of Natural Language Processing. New York: Marcel Dekker, 1998.

[45] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[46] J. Rico-Juan, J. Calera-Rubio, and R. Carrasco. Probabilistic k-testable tree languages. In A. Oliveira, editor, *Proceedings of 5th International Colloquium, ICGI*

2000, Lisbon (Portugal), volume 1891 of *Lecture Notes in Computer Science*, pages 221–228. Springer, 2000.

[47] A. Sahuguet and F. Azavant. Looking at the web through XML glasses. In *Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems*, pages 148–159, 1999.

[48] Y. Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97(1):23–60, 1992.

[49] Y. Sakakibara. Recent advances of grammatical inference. *Theoretical Computer Science*, 185(1):15–45, 1997.

[50] H. Sakamoto, H. Arimura, and S. Arikawa. Knowledge discovery from semistructured texts. In S. Arikawa and A. Shinohara, editors, *Progress in Discovery Science - Final Report of the Japanese Discovery Science Project*, volume 2281 of *LNAI*, pages 586–599. Springer, 2002.

[51] S. Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272, 1999.

[52] M. Takahashi. Generalizations of regular sets and their application to a study of context-free languages. *Information and Control*, 27:1–36, 1975.

[53] L. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[54] K. Wang and H. Liu. Discovering association of structure from semistructured objects. *To appear in IEEE Transactions on Knowledge and Data Engineering*, 1999.

[55] G. Wiederhold. *Intelligent Information Integration.* Kluwer, 1996.

[56] XML. Extensible markup language (XML) 1.0 (second edition). W3C Recommendation 6 October 2000, 2000. `www.w3.org`.

[57] XQL. XQuery 1.0: An XML query language. W3C Working Draft 16 August 2002, 2002. `www.w3.org/TR/xquery`.