Schema-Guided Induction of Monadic Queries

Jérôme Champavère, Rémi Gilleron, Aurélien Lemay, and Joachim Niehren

University of Lille and INRIA Lille-Nord Europe, France

Abstract. The induction of monadic node selecting queries from partially annotated XML-trees is a key task in Web information extraction. We show how to integrate schema guidance into an RPNI-based learning algorithm, in which monadic queries are represented by pruning node selecting tree transducers. We present experimental results on schema guidance by the DTD of HTML.

1 Introduction

Various machine learning techniques have been applied for automating Web information extraction. These range from classification [11,12,16], conditional random fields [14], inductive logic programming [7], to tree automata induction [19,13,4,15].

We study information extraction from well-structured HTML documents generated by some database. The basic problem is to find monadic queries that select informative nodes in unranked trees. Surprisingly, no schema information has been taken into account so far, even not the document type definition (DTD) of HTML. Inferred DTDs obtained from some independent algorithm have not been exploited either [1]. Instead, all available techniques rely on some finite set of attributes or local properties of the environment of nodes in the trees. The reason for ignoring schema information may be that it cannot be integrated into most approaches. Tree automata based techniques for the inference of regular tree languages are the exception [4,15], as we show in this article, but it requires considerable effort. Automata for local tree languages are not sufficient [19,13].

In this article, we introduce schema guidance into the learning algorithm for monadic queries represented by pruning node selecting tree transducers (pNSTTs) presented in [4]. These are tree automata that recognize monadic queries represented as tree languages. The first idea is to learn only such queries that are consistent with the schema, in that they select nodes in trees satisfying the schema only. This is known as *domain bias* [9] in the more restrictive framework for inference of regular word languages. The second idea is that schema information is useful in pruning heuristics for interactive query learning.

Checking the consistency of queries with respect to schemas amounts to testing language inclusion $L(A) \subseteq L(D)$ for stepwise tree automata A which may be nondeterministic [8,5] and (deterministic) DTDs D over the same signature Σ. This can be done in time $O(|A| * |\Sigma| * |D|)$ due to a recent algorithm [6], motivated by the application presented here, which is quite evolved. It avoids

A. Clark, F. Coste, and L. Miclet (Eds.): ICGI 2008, LNAI 5278, pp. 15–28, 2008.

quadratic blowups in two places: in the translation of DTDs to bottom-up deterministic tree automata by introducing factorization, and by avoiding automata complementation all over (the completion of a binary tree automaton may be of quadratic size).

Interactions between the user and the system are essential for keeping the overall amount of annotations reasonably small. Pruning heuristics serve for interactive learning of pNSTT queries from partially annotated example trees. The only completeness assumption that can be maintained in interactive information extraction is that all selected ancestors of positively annotated nodes are annotated. pNSTTs restore complete annotations by pruning subtrees that do not contain positive annotations (and interpreting missing annotations above positive annotations as negative). Pruning means to replace subtrees by placeholder symbols. pNSTTs use a single symbol $\top$ that denotes the set of all possible trees. If the schema is defined by a deterministic tree automaton S (or a DTD), we can refined this idea and replace subtrees by their type, which is the unique state into which it is evaluated by S. This leads to a generalization of pNSTTs to schema-dependant S-pNSTTs. Algorithmically, the RPNI algorithm [17] has to check whether a tree automaton is an S-pNSTT, i.e., whether it actually represents a valid query. We present a polynomial time algorithm for this purpose.

We have implemented the complete interactive learning algorithm for S-pNSTTs from scratch including the two aspects of schema guidance by S. This is done in such a way that we can run the same algorithm with or without schema consistency, schema-guided pruning, or a state typing heuristics. No other hidden heuristics or preprocessing steps have been used. A preliminary experimental evaluation yields the following insights on schema guidance by the DTD of HTML. First, it might be of interest to observe that the number of state merges is decreased considerably by schema guidance, while the learning time remains stable. This means that the time gained by fewer merge failures is sufficient to account for the additional inclusion tests. We didn't expect this effect at the beginning. It shows that the approach is feasible. Second, the overall learning quality (precision and recall) do not change very much. Typing heuristics permit to avoid wrong generalizations and allow to improve performance of learning algorithms. The effect of schema guidance remains questionable, while schema guided pruning works well. We only use HTML documents, thus the DTD of HTML. Clearly XML queries with other XML schemas must be considered to answer the remaining questions. In the interactive setting, typing heuristics and schema-guided pruning allow to decrease the number of user interactions needed to achieve a consistent query.

2 Schemas, Tree Automata, and Inclusion Checking

Schemas and node selection queries for unranked trees can be defined in various XML standards or by tree automata [8]. We will use DTDs as for the definition of HTML and stepwise tree automata for encoding DTDs, and defining queries.

```
<!ELEMENT doc
(block+)> <!ELEMENT block (text ,(link , text?)?
                           | link , text?)>
<!ELEMENT text
(#PCDATA)> <!ELEMENT link
(#PCDATA)>
```

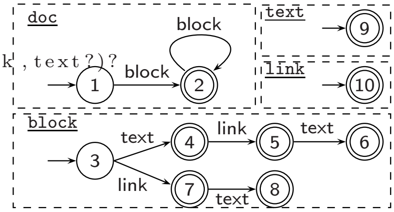

Fig. 1. An example DTD and the corresponding Glushkov automata

We recall how to translate DTDs into deterministic stepwise tree automata, and discuss inclusion checking.

Let Σ be a finite set, $\mathbb{N}$ the set of natural numbers (starting from 1) and $\mathbb{B} = \{0, 1\}$ the set of Booleans. The set of unranked trees T_Σ is the least set that contains all tuples $a(t_1, \ldots, t_n)$ where $a \in \Sigma$, $n \geq 0$, and $t_1, \ldots, t_n \in T_\Sigma$. A node of a tree is a word $\pi \in \mathbb{N}^*$ (using classical Dewey encoding). We write ϵ for the empty word and $i \cdot \pi$ for the concatenation of letter i with word π. The set of nodes of an unranked tree is $\mathsf{nodes}(a(t_1, \ldots, t_n)) = \{\epsilon\} \cup \{i \cdot \pi \mid 1 \leq i \leq n,\ \pi \in \mathsf{nodes}(t_i)\}$. The label of a tree t at a node $\pi \in \mathsf{nodes}(t)$ is denoted by $t(\pi) \in \Sigma$, the root of t is distinguished by $\mathsf{root}(t)$. By $t_{|\pi}$, we denote the subtree of t rooted by $t(\pi)$.

A *schema* over Σ is a regular language of unranked trees $L \subseteq T_\Sigma$. We will use two kinds of schema definitions: XML DTDs and stepwise tree automata [5], possibly with factorization [6] in order to avoid a quadratic blowup when encoding DTDs.

A DTD D over Σ consists of a collection of one-unambiguous regular expressions $(e_a^D)_{a \in \Sigma}$ [3], and a (unique) accepting symbol $\mathsf{start}_D \in \Sigma$. The word language $L(e) \subseteq \Sigma^*$ defined by a regular expression e over Σ is defined as usual. For every DTD D we define tree languages $L_a(D) \subseteq T_\Sigma$ by the least solution of the following system of equations where $a, a_1, \ldots, a_n \in \Sigma$:

$$L_a(D) = \{a(t_1, \ldots, t_n) \mid a_1 \cdots a_n \in L(e_a^D), a_i = \mathsf{root}(t_i), t_i \in L_{a_i}(D), 1 \leq i \leq n\}$$

The above definition basically means that the word obtained by concatenating the labels of the children of each node labeled by a must be in $L(e_a^D)$. The language of the schema is $L(D) = L_{\mathsf{start}_D}(D)$. An example DTD is given in Fig. 1. The regular expressions of DTDs can be converted into finite automata recognizing the same language by Glushkov's construction [2]. These automata are deterministic, since the regular expressions in DTDs are one-unambiguous [3]. The size of the Glushkov automaton G_e of a regular expression e is at most $|\Sigma| * |e|$, which is the maximal number of transitions in deterministic automata over Σ with $|e|$ states.

A *stepwise tree automaton* A over Σ is a standard tree automaton over the ranked signature $\Sigma_@ = \Sigma \uplus \{@\}$, where all elements of Σ are constants and @ is a binary function symbol. The rules of A, denoted $\mathsf{rules}(A)$, are of the form $a \to q$, $q_1 @ q_2 \to q$, or $q_1 \xrightarrow{\epsilon} q_2$, where $a \in \Sigma$ and $q, q_1, q_2 \in \mathsf{states}(A)$, the set of states of A. Automaton A is (bottom-up) *deterministic*, if it has no ϵ-rules and

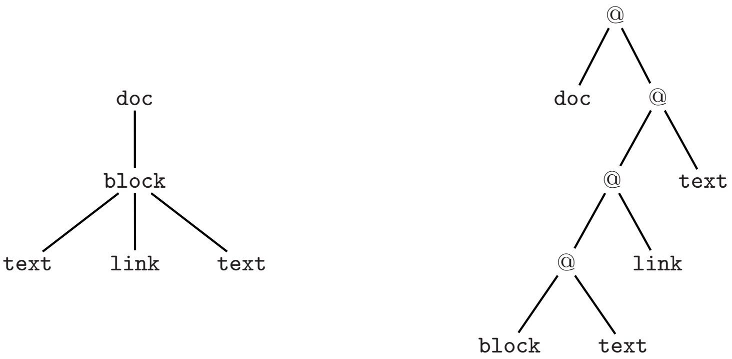

Fig. 2. Currying the unranked tree `doc`(`block`(`text`,`link`,`text`) into the binary tree `doc`@(`block`@`text`@`link`@`text`)

no two rules with the same left-hand side. We call an automaton *productive* if all of its states are accessible and co-accessible.

Let $T^{\text{bin}}_{\Sigma_@}$ be the set of binary trees over the signature $\Sigma_@$ and $L^{\text{bin}}(A) \subseteq T_{\Sigma_@}$ be the set of binary trees recognized by A. Every unranked tree $t \in T_\Sigma$ can be encoded *via* Currying into a binary tree in $T_{\Sigma_@}$, so that $\texttt{curry}(a) = a$ and if $n \geq 1$ then $\texttt{curry}(a(t_1, \ldots, t_n)) = \texttt{curry}(a(t_1, \ldots, t_{n-1}))@\texttt{curry}(t_n)$. An example is depicted in Fig. 2. Here, we write $t_1@t_2$ instead of $@(t_1, t_2)$. The language $L(A) \subseteq T_\Sigma$ of unranked trees recognized by a stepwise tree automaton A is the set:

$$L(A) = \{t \in T_\Sigma \mid \texttt{curry}(t) \in L^{\text{bin}}(A)\}$$

The direct transformation of DTDs into deterministic stepwise tree automata implies a quadratic blowup in the size of the DTD. To avoid such a blowup, we introduce *factorized tree automata* [6]. These are stepwise tree automata with ϵ-rules which represent stepwise tree automata in a compact manner. Nevertheless, we can still define an appropriate notion of determinism for factorized tree automata in order to deal with DTDs.

Definition 1. *A factorized tree automaton F over Σ is a stepwise tree automaton over Σ with ϵ-rules, and a partition into two sorts* $\mathsf{states}(F) = \mathsf{states}_1(F) \uplus \mathsf{states}_2(F)$ *such that if $q_1@q_2 \to q$ in* $\mathsf{rules}(F)$ *then $q_1 \in$* $\mathsf{states}_1(F)$ *and $q_2 \in$* $\mathsf{states}_2(F)$*. F is (bottom-up)* deterministic *if its ϵ-free part is (bottom-up) deterministic and all $q \in$* $\mathsf{states}(F)$ *have at most one outgoing ϵ-edge, the target of which must be of the other sort.*

It should be noticed that the idea of factorization is equally provided by the tree automata for unranked trees proposed in [18].

The collection of Glushkov automata $(G_a)_{a \in \Sigma}$ of a DTD D can now be translated in linear time to a factorized tree automaton F with $\mathsf{states}_1(F) = \uplus_{a \in \Sigma}\mathsf{states}(G_a)$ and $\mathsf{states}_2(F) = \Sigma$. The rules of F are defined as follows:

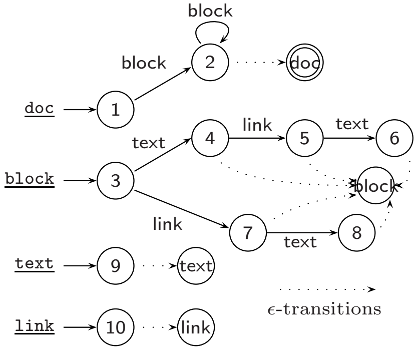

Fig. 3. The deterministic factorized tree automaton for the DTD in Fig. 1

$q_1 \xrightarrow{a} q_2 \in \mathsf{rules}(G_a)$ iff $q_1@a \to q_2 \in \mathsf{rules}(F)$, and $q \in \mathsf{final}(G_a)$ iff $q \xrightarrow{\epsilon} a \in \mathsf{rules}(F)$. This correspondence is equally useful for drawing factorized tree automata as in Fig. 3. The single final state of F is the accepting label of D, i.e., $\mathsf{final}(F) = \{\mathsf{start}_D\}$. Indeed, $L(A) = L(F)$. Furthermore, F is deterministic as a factorized tree automaton. Its ϵ-free part is deterministic, since all Glushkov automata are deterministic. The only states having outgoing ϵ-edges are the final states of Glushkov automata, which are of sort 1. They have at most one outgoing ϵ-edge, since every q belongs to at most one Glushkov automaton G_a; the target of this edge is of other sort 2. In principle, these ϵ-edges can be eliminated in order to obtain a deterministic tree automaton, but this could lead to a quadratic size increase.

Theorem 1 ([6]). *Language inclusion $L(A) \subseteq L(F)$ between stepwise tree automata A with ϵ-rules and deterministic factorized tree automata F can be tested in time $O(|A| * |F|)$.*

The most important point here is that one does not have to compute the automaton for the complement of F, which could grow up quadratically to $O(|F|^2)$ since completion is necessary before swapping final states. The second point is that factorization avoids a quadratic blowup when translating DTDs into stepwise tree automata. As a corollary, we can check language inclusion between stepwise tree automata A and DTDs D over signature Σ in time $O(|A| * |\Sigma| * |D|)$. Third, the inclusion test is incremental with respect to adding ϵ-edges to A. This can be used to check inclusion incrementally in a learning algorithm because, after state merging in A, we simply add back and forth ϵ-edges rather than physically identifying the merged states.

3 Schema-Guided pNSTTs for Monadic Queries

We generalize the notion of pNSTTs to S-pNSTTs such that pruning is guided by a schema S, and present a polynomial time algorithm testing whether a deterministic tree automaton is an S-pNSTT.

Node Selecting Tree Transducers. A *monadic query* Q in unranked trees over Σ is a total function mapping trees $t \in T_\Sigma$ to sets of nodes $Q(t) \subseteq \mathsf{nodes}(t)$. We call a monadic query Q *consistent with a schema* $L \subseteq T_\Sigma$ if it selects nodes only in trees satisfying the schema, i.e., if $Q(t) = \emptyset$ for all $t \in T_\Sigma \setminus L$.

A (Boolean) *annotated tree* over Σ is a tree over $\Gamma = \Sigma \times \mathbb{B}$. Every annotated tree $s \in T_\Gamma$ can be decomposed in a unique manner into two trees $t \in T_\Sigma$ and $\beta \in T_\mathbb{B}$ with the same sets of nodes $\mathsf{nodes}(s) = \mathsf{nodes}(t) = \mathsf{nodes}(\beta)$ such that for all nodes π therein, $s(\pi) = (t(\pi), \beta(\pi))$. In this case, we write $s = t * \beta$.

A language $L \subseteq T_\Gamma$ of annotated trees defines a relation $r_L \subseteq T_\Sigma \times T_\mathbb{B}$ between trees of the same structure, which is $r_L = \{(t, \beta) \mid t * \beta \in L\}$. We call L *functional* if this relation r_L is a partial function. In other words, for every tree $t \in T_\Sigma$ there exists at most one tree $\beta \in T_\mathbb{B}$ such that $t * \beta \in L$. A *node selecting tree transducer (NSTT)* for Σ is an automaton over Γ which recognizes a functional language of annotated trees.

A completely annotated example for a query Q is a tree $t * \beta$ where $\beta(\pi) = 1$ for all selected nodes $\pi \in Q(t)$ and $\beta(\pi) = 0$ otherwise. An algorithm for testing functionality and an RPNI algorithm for learning NSTTs from completely annotated examples have been presented in [4].

Schema-Guided Pruning. In Web information extraction, however, only partially annotated examples for the target query Q are available. These are triples (t, e_+, e_-) such that $e_+ \subseteq Q(t)$ and $e_- \subseteq \mathsf{nodes}(t) \setminus Q(t)$. The only completeness assumption for partially annotated examples that can be maintained is that selected nodes on paths from the root to some annotated selected node in e_+ are annotated, too. If $\pi \in e_+$ and $\pi' \in Q(t)$ is a prefix of π then $\pi' \in e_+$.

Pruning is a method by which to deal with partially annotated examples. The idea is to cut down all subtrees from t that do not contain nodes in e_+. These subtrees are replaced by special symbols, which indicate the type of the subtree. Then all other nodes are annotated by using the given partial annotation and the completeness assumption, i.e., a node is annotated by 1 if it is in e_+, and by 0 otherwise. The pNSTTs from [4] permit only a single type satisfied by all trees. When having schema information available, we can refine this approach by using the states of the schema as type information. This leads to the following formal definitions.

Let S be a tree automaton over Σ which defines the schema, and let us consider $\mathsf{states}(S)$ as symbols of arity 0. An *S-pruned annotated tree* is a tree $s \in T_{\Gamma \cup \mathsf{states}(S)}$. We call a tree $s \in T_{\Gamma \cup \mathsf{states}(S)}$ an *S-consistent pruning* of an annotated tree $t * \beta \in T_\Gamma$ if:

(i) $s(\pi) = t * \beta(\pi)$ if $s(\pi) \in \Gamma$, or
(ii) $t_{|\pi} \in L_{s(\pi)}(S)$ if $s(\pi) \in \mathsf{states}(S)$.

Now, let us consider a language $L \subseteq T_{\Gamma \cup \mathsf{states}(S)}$. L is *S-cut-functional* if for all s_1 and s_2 in $T_{\Gamma \cup \mathsf{states}(S)}$ such that there exist $t * \beta_1$ and $t * \beta_2$ in L with s_1, resp s_2, an S-consistent pruning of $t * \beta_1$, resp. $t * \beta_2$, then for all nodes $\pi \in \mathsf{nodes}(\beta_1) \cap \mathsf{nodes}(\beta_2)$, it holds that $\beta_1(\pi) = \beta_2(\pi)$. In other words, two S-consistent prunings of an annotated tree can not define contradictory annotations. We can now define S-guided pruning node selecting tree transducers.

Definition 2. *Given a schema S, an S-*pNSTT *over Σ is a tree automaton over signature $(\Sigma \times \mathbb{B}) \cup \mathsf{states}(S)$ whose language is S-cut-functional.*

If $S_\top$ is the tree automaton with a unique state $\top$ that recognizes all unranked trees, then $S_\top$-pNSTTs coincide with pNSTTs presented before in [4]. Such pNSTTs can be learned by the variant pRNPI of RPNI, which tests for cut-functionality after all deterministic merges. In order to generalize pRNPI with schema-guided pruning, we need an algorithm testing S-cut-functionality for languages recognized by tree automata over the signature $\Gamma \cup \mathsf{states}(S)$.

Let S be a deterministic factorized tree automaton over Σ and let A be a deterministic stepwise tree automaton over $\Gamma \cup \mathsf{states}(S)$. Deciding whether A is an S-pNSTT, i.e., verifying that the language of A is S-cut-functional, can be done by a ground Datalog program of polynomial size, and thus in polynomial time (see, e.g., [10]). Such a program can be inferred from A and S as described in the following.

The predicate $\mathtt{schema}(p,q)$ holds for $p \in \mathsf{states}(A)$ and $q \in \mathsf{states}(S)$ if there exist an annotated tree $t * \beta$ and an S-consistent pruning s of $t * \beta$ such that A evaluates s to p and S evaluates t to q. Note that only the second rule is concerned by pruning (state q of S is a constant symbol for A).

$$\frac{(a,b) \to p \in \mathsf{rules}(A) \qquad a \to q \in \mathsf{rules}(S)}{\mathtt{schema}(p,q).} \qquad \frac{q \in \mathsf{states}(S) \qquad q \to p \in \mathsf{rules}(A)}{\mathtt{schema}(p,q).}$$

$$\frac{p_1@p_2 \to p \in \mathsf{rules}(A) \qquad q_1@q_2 \to q \in \mathsf{rules}(S)}{\mathtt{schema}(p,q) :\!- \ \mathtt{schema}(p_1,q_1), \mathtt{schema}(p_2,q_2).} \qquad \frac{q \xrightarrow{\epsilon} q' \in \mathsf{rules}(S)}{\mathtt{schema}(p,q') :\!- \ \mathtt{schema}(p,q).}$$

The predicate $\mathtt{sim}(p,p')$ (for similar pruning) holds for two states p, p' of $\mathsf{states}(A)$ if there exist two S-consistent-pruning s, s' of the same annotated tree $t * \beta$ which are evaluated to p and p' by A. Here, only the second rule is directly concerned by pruning.

$$\frac{p \in \mathsf{states}(A)}{\mathtt{sim}(p,p).} \qquad \frac{q \in \mathsf{states}(S) \qquad q \to p' \in \mathsf{rules}(A)}{\mathtt{sim}(p,p') :\!- \ \mathtt{schema}(p,q).}$$

$$\frac{p_1@p_2 \to p \in \mathsf{rules}(A) \qquad p'_1@p'_2 \to p' \in \mathsf{rules}(A)}{\mathtt{sim}(p,p') :\!- \ \mathtt{sim}(p_1,p'_1), \mathtt{sim}(p_2,p'_2).}$$

The predicate $\mathtt{dast}(p,p')$ (different annotations on same tree) holds for two states p, p' of $\mathsf{states}(A)$ if there exist an S-consistent pruning s of $t * \beta$ and an S-consistent pruning s' of $t * \beta'$ with a position π verifying $s(\pi) \in \Gamma$, $s'(\pi) \in \Gamma$ and $s(\pi) \neq s'(\pi)$, such that A evaluates s to p and s' to p'. This predicate allows to detect failure for testing S-cut-functionality.

$$\frac{(a,b) \to p \in \mathsf{rules}(A) \qquad (a,\neg b) \to p' \in \mathsf{rules}(A)}{\mathtt{dast}(p,p').}$$

$$\frac{p_1@p_2 \to p \in \mathsf{rules}(A) \qquad p'_1@p'_2 \to p' \in \mathsf{rules}(A)}{\begin{array}{c}\mathtt{dast}(p,p') :- \mathtt{dast}(p_1,p'_1), \mathtt{dast}(p_2,p'_2).\\ \mathtt{dast}(p,p') :- \mathtt{dast}(p_1,p'_1), \mathtt{sim}(p_2,p'_2).\\ \mathtt{dast}(p,p') :- \mathtt{sim}(p_1,p'_1), \mathtt{dast}(p_2,p'_2).\end{array}}$$

The next proposition indicates how to determine whether an automaton is S-cut-functional by using the inferred Datalog program.

Proposition 1. *A deterministic tree automaton A over $\Gamma \cup \mathsf{states}(S)$ is S-cut-functional with respect to a productive deterministic factorized tree automaton S over Σ if and only if there are no two states p, $p' \in \mathsf{states}(A)$ such that $\mathtt{dast}(p,p')$ holds, and either p, $p' \in \mathsf{final}(A)$ or $\mathtt{sim}(p,p')$ holds.*

Note that when defining the schema by automaton $S_\top$, predicate `schema` becomes trivial, and the test for $S_\top$-cut-functionality coincides with the cut-functionality test for pNSTTs presented in [4].

4 Schema-Guided Learning

We present the learning algorithm $\mathtt{RPNI}^{S,\mathsf{type}}_{\mathsf{prune},\mathsf{cons}}$ in Fig. 4 which is a variant of RPNI [17] that learns S-pNSTTs in a schema-guided manner. It is parameterized by a deterministic factorized tree automaton S over Σ which defines the schema. It inputs a finite set of completely annotated examples $E \subseteq T_{\Sigma\times\mathbb{B}}$ and a single partially annotated example $\langle t, e_+, e_-\rangle$ in $T_\Sigma \times \mathsf{nodes}(t)^2$. The algorithm could easily be extended to a set of partially annotated examples, but a single one is enough in most interactive learning scenarios.

The schema S intervenes in the definition of the pruning algorithm prune_S, in definitions of queries by deterministic S-pNSTTs over Σ (distinguished by S-cut-functionality), and in consistency checking of queries with respect to the schema $L(S)$, which amounts to check for language inclusion (in polynomial time since S is deterministic). We will consider several variants of the algorithm: whether S-guided pruning is done; whether S-consistency is checked for queries, and whether typing heuristics are used.

Learning without schema means to choose $S = S_\top$, the automaton with a single state that accepts all trees. S-consistency checking for queries can be switched on by choosing parameter $\mathsf{cons} =$ yes. Learning without pruning amounts to set prune_S to the identity function on annotated trees, i.e., $\mathsf{prune}_S(s) = s$ for all $s \in T_{\Sigma\times\mathbb{B}}$. With pruning, the function prune_S replaces subtrees, in which no nodes are selected, by their state with respect to S. This can be defined as follows where $t \in T_\Sigma$, $\beta \in T_\mathbb{B}$, $s, s_1, \ldots, s_n \in T_{\Sigma\times\mathbb{B}}$, $a \in \Sigma$, and $b \in \mathbb{B}$:

$$\begin{array}{ll}\mathsf{prune}_S(t * \beta) = \mathsf{eval}_S(t) & \text{if } \beta \in T_{\{0\}}\\ \mathsf{prune}_S((a,b)(s_1,\ldots,s_n)) = (a,b)(\mathsf{prune}_S(s_1),\ldots,\mathsf{prune}_S(s_n)) & \text{otherwise}\end{array}$$

Here $\mathsf{eval}_S(t) \in \mathsf{states}(S)$ is the state into which S evaluates t. This state exists for all annotated examples since these are supposed to satisfy schema S. It is unique since S is assumed deterministic.

```
$\mathtt{RPNI}^{S,\mathsf{type}}_{\mathsf{prune},\mathsf{cons}}$ ($E$, $\langle t, e_+, e_- \rangle$)
// sample of completely annotated examples $E \subseteq T_{\Sigma \times \mathbb{B}}$
// partially annotated example $\langle t, e_+, e_- \rangle \in T_\Sigma \times \mathsf{nodes}(t)^2$
// schema defined by a deterministic factorized tree automaton $S$ over $\Sigma$

// prune all example trees w.r.t. schema definition $S$//
let $E' = \{\mathsf{prune}_S(t' * \beta) \mid t' * \beta \in E\} \cup \{\mathsf{prune}_S(t * p_+)\}$
// compute the initial automaton
let $A$ be a deterministic $S$-pNSTT such that $L(A) = E'$
let states($A$)= $\{q_1, \ldots, q_n\}$ in some admissible order
// generalize $A$ by state merging //
for $i = 1$ to $n$ do
    for $j = 1$ to $i - 1$ with $\mathsf{type}$ $(q_j)$=$\mathsf{type}$ $(q_i)$ do
        let $A'$ = det-merge$(A, q_i, q_j)$
        if $A'$ is $S$ cut-functional // $S$-consistency of annotations on pruned trees
        and if $\mathsf{cons}$ =yes then $\{t \mid t * \beta \in L(A')\} \subseteq L(S)$ // query $S$-consistent
        and $A'$ consistent with sample $E$ and example $\langle t, e_+, e_- \rangle$
        then $A \leftarrow A'$
        else skip
Output : $A$
```

Fig. 4. Learning from completely and partially annotated example trees

State typing heuristics forbid to merge states of the automaton that have different types $\mathsf{type}(q_j) \neq \mathsf{type}(q_i)$. This is called typing bias in [9]. The definition of types depends on the application. In our algorithm, they are introduced by parameter type. When no typing heuristics are used, we set type to a constant function on states. Otherwise, we say that a state q of stepwise tree automaton A has type $\mathsf{type}(q) = a \in \Sigma$, if q is reachable from a in the graph representing the stepwise automaton. We impose that no two states may have the same type, so that the graphical representation of A can be decomposed into a disjoint union of independent connected components for all letters $a \in \Sigma$. For instance, typing heuristics for HTML forbid shared generalizations for different elements, which may be tables, rows, or lines, or the same elements with different attribute values.

Interactive Learning. All successful Web information extraction systems learn in an incremental manner [19,16,4]. This is essential for solving extraction tasks with a reasonably small number of user annotations. The algorithm $\mathtt{RPNI}^{S,\mathsf{type}}_{\mathsf{prune},\mathsf{cons}}$ can be used as core learning algorithm in an interactive environment such as Squirrel [4], in which it is repeatedly applied during user interaction.

Incremental learning algorithms help users to annotate a collection of Web pages. They always know a current hypothesis for the target query, which should solve the information extraction task at the end. At the beginning, this query can be chosen to be empty, i.e., $Q(t) = \emptyset$ for all $t \in T_\Sigma$. For every Web page, the user loops as follows in order to find complete annotations for all pages:

- apply the current query hypothesis to the current page,
- either accept the result and continue with the next page,
- or else, correct some of the errors by adding new partial annotations for the current page, learn a new query hypothesis by running algorithm $\mathtt{RPNI}^{\mathcal{S},\mathsf{type}}_{\mathsf{prune},\mathsf{cons}}$ with all complete annotations for earlier pages and all partial annotations for the current page, and repeat the procedure for the current page.

The quality of such interactive learning algorithms is usually measured in the number of pages that are to be annotated before the target query is found, and in the total number of corrections effected on these pages.

5 Experimental Results

We have implemented the learning algorithm $\mathtt{RPNI}^{\mathcal{S},\mathsf{type}}_{\mathsf{prune},\mathsf{cons}}$, and integrated it into the interactive environment Squirrel. Here we describe aspects of the implementation and results of experiments of schema-guided query induction for Web information extraction. The schema definition we use is the DTD of HTML.

Preprocessing the DTD of HTML. The complete DTD of HTML is huge. We have kept only the essential part with respect to information extraction. Indeed, HTML (XHTML1-transitional) DTD has 89 defined symbols. In practice, depending on the considered set of Web pages, the number of elements actually used ranges between 20 and 30. Reducing the size of the DTD can be done by filtering those elements and putting away the others and the unused rules of the schema automaton, i.e., rules that contain states that are not accessible or co-accessible. The automaton for the whole HTML DTD obtained by classical Glushkov's construction has 3951 rules. This reduction technique allows us to deal with an automaton whose number of rules ranges from 107 to 218 depending on the benchmark, and thus to speed up the inclusion tests.

Implementing the Learner. We have implemented $\mathtt{RPNI}^{\mathcal{S},\mathsf{type}}_{\mathsf{prune},\mathsf{cons}}$ in Objective CAML. Besides the usual efforts for implementing RPNI, a large part of the effort was spend on the inclusion test, which is done in an incremental manner. All parameters of the algorithm are provided, and can be freely instantiated (schema-guided pruning, schema-guided consistency, state typing heuristics). This allows us to measure the impact of these heuristics together or independently. No further heuristic has been introduced. This is quite important. It excludes all kinds of dirty tricks, so that the results can be obtained from the description presented here. As a drawback, it leaves some room for improving the performance.

Benchmarks. We have performed preliminary experiments on three benchmarks: Google, Okra and Bigbook[1]. Google presents a set of 34 result Web pages for the well-known search engine where links are to be extracted. Okra (251 pages) and Bigbook (234 pages) are classical benchmarks for data extraction

[1] Those benchmarks can be found at `http://www.grappa.univ-lille3.fr/~carme/WebWiki/DataSets.html`

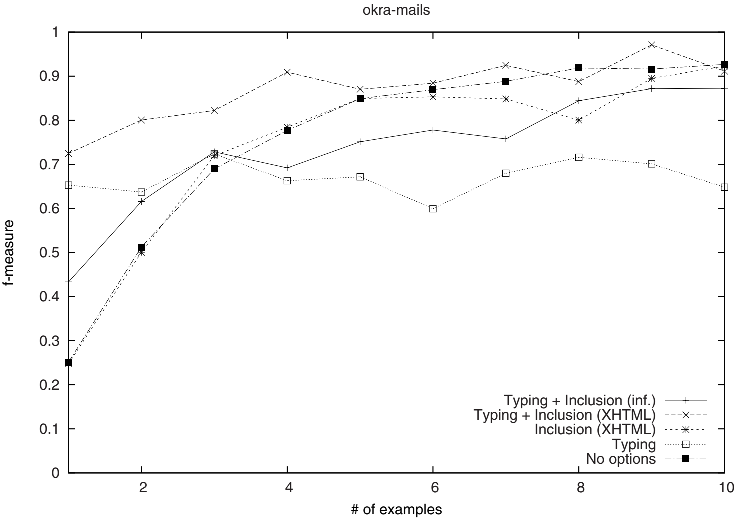

Fig. 5. Experimental results for Okra benchmark in non-interactive mode

on the Web. They both correspond to lists of people with several information on them. The task on Okra is to extract emails of persons, and names for Bigbook. We present here only Okra and Google because of space constraints. While a far simpler task, Bigbook results are comparable with those of Google in the way that the order of the different curves are similar, but with better overall results for every option set.

Non-interactive Learning. For the chosen benchmarks, a sample of 1 to 10 randomly chosen completely annotated Web pages is submitted to the learning algorithm. The resulting queries is tested on 30 other Web pages of the corpus using precision, recall and f-measure. The presented results are averages on 30 experiments.

Results are presented for different sets of options in Fig. 5 and 6. Results for pruning are not presented here because this option does not alter a lot the results in non-interactive learning. For inclusion with inferred DTDs, only the best result is presented (the one obtained when joint with typing).

From this, several conclusions can be raised. First, surprisingly, inclusion with inferred DTD leads to poor results. This might be related to the chosen DTD inference algorithm itself. Second, inclusion alone is not helpful, but joint with typing, it may be of serious help. In Okra benchmark, the learning algorithm with typing and inclusion within HTML DTD gives the best results, especially with few examples. On the other hand, results on Google (and Bigbook) do not really improve existing results.

Also, experiments on running time and number of merges performed have been done. Results vary but algorithm with schema consistency checking is usually around five time slower, which is still acceptable for an interactive use considering

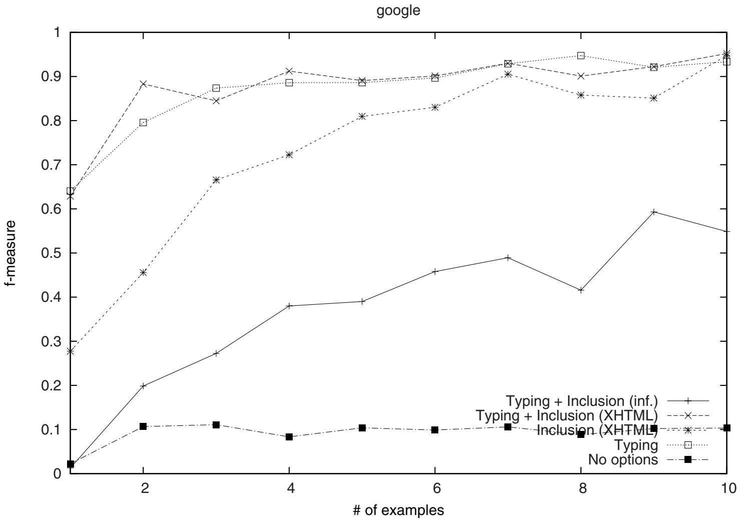

Fig. 6. Experimental results for Google benchmark in non-interactive mode

that the overall learning time rarely exceed one second. This proves the feasibility of the approach. On Okra, the schema consistency option allows to reduce the number of merges. This means that in this case, extra computation time can actually be compensated by a better guiding of merge operations. However, on other benchmarks, the number of merges is quite similar with or without schema consistency.

Interactive Learning. We now evaluate our algorithm in an interactive setting. It is tested automatically by a "user simulator" which performs the following task. We first begin with the empty query and a randomly chosen Web page. The query is tested on the Web page. If the result of the query is not correct, the user gives exactly one extra annotation (a correction): either a positive annotation on a node that the query forgot or a negative annotation on a node that is incorrectly annotated by the query. The annotation is chosen as being on the first node (in the document order) returned by the query that is incorrect. If the annotation returned by the query is correct, the user choses an other page and reiterate until it is satisfied, which is obtained when 30 consecutive Web pages are correctly annotated by this process.

During this protocol, we count the number of corrections the user had to do and the number of pages on which there has been an interaction (Web pages on which the query was already correct is not counted here). Also, once an annotated Web page has been accepted, we complete its annotation, i.e., we annotate negatively every non-annotated node. The results presented in Table 1 are averages on 30 experiments.

All the experiments have been performed using pruning.We present here the results obtained when adding typing, inclusion within HTML DTD, or both.

Table 1. Interactive learning. For each dataset, we present the number of necessary corrections/pages to learn the target query (T=typing heuristics; I=inclusion; P=schema-guided pruning). All experiments have been done with regular pruning, unless P is specified.

	T	I	T + I	T + I	T + P
		(HTML DTD)	(HTML DTD)	(Inferred DTD)	(HTML DTD)
Okra	failed	17.93/3.87	4.00/2.03	4.60/2.73	3.73/1.87
Bigbook	3.03/1.37	3.20/1.57	2.77/1.77	2.33/1.33	3.90/1.37
Google	4.53/2.33	9.60/3.43	8.00/4.00	28.60/12.03	6.90/3.53

Also, we present results with typing and inclusion within the inferred DTD. Last, we tried inference using typing and pruning using schema, without using inclusion. Indeed, while inclusion is not possible with regular pruning (i.e., without schema), it is possible to prune trees by replacing them by their DTD state, without using inclusion. This is useful to separate the effect of this kind of pruning and the one of inclusion checking.

Those results are to be compared with other existing systems. The Squirrel system [4] learns correctly the query for Okra with 1.6 pages and 3.5 (average) corrections, 1 page and 3 corrections for Bigbook and 1.9 pages and 4.8 corrections for Google. Squirrel is basically the same algorithm as the one presented here, with options typing and pruning, but with several other heuristics and various optimizations. In [19], the (k,l)-contextual learning algorithm can infer the query for Okra and Bigbook with respectively 2 and 2.3 corrections (number of pages is not specified).

In the interactive setting, we can observe that there is a benchmark where inclusion really helps. On Okra, we have not been able to obtain decent results without this option. On the other hand, on Google and Bigbook, inclusion either does not give important improvement or even may lower performance a bit. Surprisingly, results of pruning with schema without using inclusion are a bit better (although maybe not significantly) than with inclusion. This could tend to indicate that, at least on observed benchmarks, the main use of the schema is actually in the pruning part rather than in the inclusion test.

Acknowledgments. We are grateful to Grégoire Laurence for his help on experiments. The work was supported by the Mostrare team-project of the INRIA Lille-Nord Europe research center, the Laboratoire d'Informatique Fondamentale de Lille (LIFL) UMR 8022 CNRS.

References

1. Bex, G.J., Neven, F., Schwentick, T., Tuyls, K.: Inference of Concise DTDs from XML data. In: VLDB, pp. 115–126 (2006)
2. Brüggemann-Klein, A.: Regular Expressions to Finite Automata. Theoretical Computer Science 120(2), 197–213 (1993)

3. Brüggemann-Klein, A., Wood, D.: One-unambiguous Regular Languages. Information and Computation 142(2), 182–206 (1998)
4. Carme, J., Gilleron, R., Lemay, A., Niehren, J.: Interactive Learning of Node Selecting Tree Transducers. Machine Learning 66(1), 33–67 (2007)
5. Carme, J., Niehren, J., Tommasi, M.: Querying Unranked Trees with Stepwise Tree Automata. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 105–118. Springer, Heidelberg (2004)
6. Champavère, J., Gilleron, R., Lemay, A., Niehren, J.: Efficient Inclusion Checking for Deterministic Tree Automata and DTDs. In: LATA (to appear, 2008)
7. Cohen, W.W., Hurst, M., Jensen, L.S.: A Flexible Learning System for Wrapping Tables and Lists in HTML Documents. In: WWW, pp. 232–241 (2002)
8. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications (revised) (October 2007), `http://www.grappa.univ-lille3.fr/tata`
9. Coste, F., Fredouille, D., Kermovant, C., de la Higuera, C.: Introducing Domain and Typing Bias in Automata Inference. In: Paliouras, G., Sakakibara, Y. (eds.) ICGI 2004. LNCS (LNAI), vol. 3264, pp. 115–126. Springer, Heidelberg (2004)
10. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys 33(3), 374–425 (2001)
11. Finn, A., Kushmerick, N.: Multi-level Boundary Classification for Information Extraction. In: Boulicaut, J.-F., Esposito, F., Giannotti, F., Pedreschi, D. (eds.) ECML 2004. LNCS (LNAI), vol. 3201, pp. 111–122. Springer, Heidelberg (2004)
12. Gilleron, R., Marty, P., Tommasi, M., Torre, F.: Interactive Tuples Extraction from Semi-structured Data. In: WI, pp. 997–1004 (2006)
13. Kosala, R.: Information Extraction by Tree Automata Inference. PhD thesis, K. U. Leuven (July 2003)
14. Kristjansson, T.T., Culotta, A., Viola, P., McCallum, A.: Interactive Information Extraction with Constrained Conditional Random Fields. In: AAAI (2004)
15. Lemay, A., Niehren, J., Gilleron, R.: Learning n-ary Node Selecting Tree Transducers from Completely Annotated Examples. In: Sakakibara, Y., Kobayashi, S., Sato, K., Nishino, T., Tomita, E. (eds.) ICGI 2006. LNCS (LNAI), vol. 4201, pp. 253–267. Springer, Heidelberg (2006)
16. Lerman, K., Minton, S., Knoblock, C.: Wrapper Maintenance: a Machine Learning Approach. Journal of Artificial Intelligence Research 18, 149–181 (2003)
17. Oncina, J., Garcia, P.: Inferring Regular Languages in Polynomial Update Time. In: Pattern Recognition and Image Analysis, pp. 49–61 (1992)
18. Raeymaekers, S.: Information Extraction from Web Pages Based on Tree Automata Induction. PhD thesis, K. U. Leuven (January 2008)
19. Raeymaekers, S., Bruynooghe, M., Van den Bussche, J.: Learning (k,l)-contextual Tree Languages for Information Extraction. In: Gama, J., Camacho, R., Brazdil, P.B., Jorge, A.M., Torgo, L. (eds.) ECML 2005. LNCS (LNAI), vol. 3720, pp. 305–316. Springer, Heidelberg (2005)