Theory Comput. Systems **40**, 163–185 (2007)
DOI: 10.1007/s00224-005-1233-3

Theory of Computing Systems
© 2005 Springer Science+Business Media, Inc.

Query Learning of Regular Tree Languages: How to Avoid Dead States

Frank Drewes and Johanna Högberg

Department of Computing Science, Umeå University,
S-901 87 Umeå, Sweden
{drewes,johanna}@cs.umu.se

Abstract. We generalize an inference algorithm by Angluin, that learns a regular string language from a "minimally adequate teacher", to regular tree languages. The (deterministic bottom-up) finite tree automaton constructed by the learning algorithm is the minimal partial one recognizing the unknown language. This improves a similar algorithm proposed by Sakakibara by avoiding dead states both in the resulting automaton and the learning phase, which also leads to a considerable improvement with respect to efficiency.

1. Introduction

Language learning addresses the problem of algorithmically deriving an explicit grammatical description of a language U which is only implicitly available in the form of examples or similar information. Given such information, the algorithmic goal is to derive a formal description of U, e.g., an automaton or a grammar. A popular model for language learning is *query learning*: The learning algorithm, called the *learner*, may actively query an oracle, the *teacher*, in order to gather information about U [Ang2] (see also [Ang3] for a recent survey focusing on results about the number of queries needed).

One type of query learning, often called MAT-learning, was proposed by Angluin in [Ang1]. It assumes the existence of a *minimally adequate teacher* (MAT) capable of answering two types of queries: membership and equivalence queries. A membership query asks whether a certain string is an element of U. An equivalence query asks whether a proposed automaton correctly describes U. If not, the teacher returns a counterexample, i.e., an element of the symmetric difference of both languages. For the case where U is regular, it is shown in [Ang1] that the minimal deterministic finite automaton recognizing U can be constructed in polynomial time in the MAT model. Several other

researchers adopted the model to learn languages by constructing, e.g., nondeterministic finite-state automata [Yok] and restricted types of context-free grammars [BR], [Ish2], [SY], [FR]. (For a general overview of results about learning context-free languages see [Lee].) Fernau [Fer] showed that the so-called deterministic even linear matrix grammars can be constructed in polynomial time in the MAT model.

In this article we use the same model, but for learning regular *tree* languages. These languages provide a well-known generalization of regular string languages to which nearly all the classical results carry over (see, e.g., [GS1] and [GS2]). In particular, there is a convenient type of language acceptor for this class of languages, namely the deterministic bottom-up finite-state tree automaton [TW], called fta in the following.

An important property of regular tree languages is that their yields are exactly the context-free string languages. Here, the yield of a tree is the string of leaves of this tree, read from left to right. Intuitively, the tree language consists of the derivation trees of strings in the string language. In fact, for this it suffices to consider so-called skeletal trees, in which internal nodes are unlabelled [LJ]. Such a tree reveals only the syntactic structure of the string but not the concrete rules generating it.

These facts make it interesting to study learning algorithms for regular tree languages as is done in, e.g., [FK], [Sak1], [Sak2], and [COC]. Using the MAT model, Sakakibara [Sak1] extended the algorithm proposed by Angluin to skeletal regular tree languages. In this way, context-free string languages can be learned in polynomial time, provided that the teacher is able to check (skeletal) trees and fta's (instead of strings and ordinary finite-state automata), supplying the learner with skeletal trees as counter-examples. In the following we drop the restriction to skeletal trees because regular tree languages are not only useful as a syntactic basis for context-free string grammars, but also for, e.g., graph and picture generating devices (see, e.g., [Eng], [Dre1], and [Dre2]).

We propose an alternative extension of the algorithm by Angluin, which generalizes and improves the one in [Sak1]. Our algorithm constructs the minimal partial fta accepting the regular tree language learned. It combines the technique by Angluin with contradiction backtracking [Sha] in order to avoid dealing with large or unnecessary examples. Here, an example is considered to be unnecessary if it provides the same information as an earlier one, or corresponds to a "dead state". As a consequence, we are not only able to show that the algorithm runs in polynomial time, but also that the maximum rank of symbols and the maximum size of counter-examples returned by the teacher appear only as linear factors in the expression that bounds its running time. The remainder of this Introduction discusses our approach in more detail.

We first describe the approach used in [Sak1]. Inspired by the Myhill–Nerode theorem and the minimization of finite automata, the central idea is to determine equivalence classes of trees, using them as the states of a total fta. For this, finite sets S and C of trees and contexts are maintained, where a context is a tree c with a unique placeholder for which another tree s can be substituted, which is as usual denoted by $c[\![s]\!]$. Intuitively, the trees in S are representatives of the equivalence classes found, and the contexts in C witness that certain trees in S belong to different equivalence classes. More precisely, given some tree s, let $obs_C(s)$ be a bit string of all "observations", telling for each $c \in C$ whether $c[\![s]\!] \in U$. Then $s, s' \in S$ are inequivalent if $obs_C(s) \neq obs_C(s')$. Even though the converse might not be true, the algorithm will tentatively assume that s and s' are equivalent if $obs_C(s) = obs_C(s')$. Based on this assumption (and certain consistency

and completeness conditions that we omit here), an fta can be synthesized. Its state set is $\{obs_C(s) \mid s \in S\}$, where a state $obs_C(s)$ is accepting if $s \in U$. The bottom-up transition function δ is obtained by defining, for every tree $s = f[s_1, \dots, s_k]$ with $s_1, \dots, s_k \in S$,

$$\delta(obs_C(s_1) \cdots obs_C(s_k), f) = obs_C(s).$$

If the synthesized automaton is passed to the teacher and she returns a counter-example, then this counter-example is, together with all its subtrees, added to S. This will create an inconsistency, which is resolved by adding a suitable context to C. In this way, S and C are iteratively enlarged until all equivalence classes have been found and separated from each other. The synthesized automaton will then be the minimal total fta recognizing U.

Assuming that a fixed ranked alphabet is considered, it is shown in [Sak1] that this algorithm is polynomial in the size of the sought automaton and the size of counter-examples received from the teacher. However, one may identify two problems with this approach:

1. Let m be the maximum size of counter-examples returned by the teacher and let r be the maximum rank of symbols in the alphabet. Then the expression m^r appears as a factor in all major terms of the polynomial that bounds the running time of the learner. This is mainly due to the fact that, if the teacher selects a counter-example that is larger than necessary, the tree and all its subtrees are added to the set S. Thus, states may be represented many times in S. Since S must be searched in all important parts of the algorithm, this has quite a negative impact on the running time. Due to this fact, the algorithm is polynomial only if r is assumed to be a constant (which is of course the case if the alphabet is considered to be fixed).

   The approach presented in this paper uses the counter-examples provided by the teacher in a different way in order to overcome this problem. Roughly speaking, we search a given counter-example t for a single small subtree s representing a new equivalence class. We then decompose t into $c[\![s]\!]$ and add s to S and c to C. As will be seen later, this results in sets S and C being, in a certain sense, minimal. In contrast to this, C may contain huge contexts (obtained from large counter-examples), but this does not matter as the algorithm never needs to inspect a context in C in detail. As a consequence, the upper bound on the running time no longer involves the factor m^r; only the two linear factors m and r occur.

   Our method to extract a suitable tree s from a counter-example (described in detail in Section 3) is essentially the contradiction backtracking technique by Shapiro [Sha],[1] which has also been used to learn regular string languages (see, e.g., [Ish1] and [Yok]).
2. As pointed out in [Sak1], for many practically relevant languages the constructed fta will typically contain many transitions that involve a dead state, i.e., one from which no path leads to an accepting state. Being a minimal total fta, the automaton can at most contain one dead state, but in the extreme all but a linear number of transitions may lead to that state. In other words, the transition table of the minimal partial fta (obtained from the total one by removing the dead state

[1] We thank an anonymous referee who observed this relationship and provided us with the reference.

together with all transitions involving it) may be much smaller than the minimal total one. In particular, if we do not assume any more that a fixed alphabet is considered, thus allowing r to vary, the minimal total fta may be exponentially larger than the minimal partial one. Thus, it would be valuable to be able to construct the minimal partial fta by an algorithm whose running time depends on the size of *that* automaton rather than on the size of the total one.

We show that this is indeed possible. Our algorithm maintains a third set R of trees besides S and C. Instead of defining the transition function by simply setting $\delta(obs_C(s_1)\cdots obs_C(s_k), f) = obs_C(s)$ for all $s = f[s_1, \ldots, s_k]$ with $s_1, \ldots, s_k \in S$, which by construction yields a total fta, we collect representatives of transitions in R and set $\delta(obs_C(s_1)\cdots obs_C(s_k), f) = obs_C(s)$ for all $s = f[s_1, \ldots, s_k]$ in R. By a careful choice of the trees in R, we guarantee that no tree in R represents a dead state. In fact, it will also turn out that R is minimal in the sense that its trees represent pairwise distinct transitions.

We finally discuss the practical relevance of the questions, assumptions, and results described above. Clearly, the minimally adequate teacher is a formal idealization. A real teacher will probably not be able to check the correctness of a proposed fta directly, and in particular not with error probability zero. However, one may nevertheless think of meaningful applications in which the theoretical teacher is simulated by, e.g., a user who wants to convey her knowledge to the learner. To see this, it is helpful to notice that the tree language itself is usually not what the "end user" is interested in. Instead, a tree is an internal syntactic description of some object of interest. It may, as in [Sak1], be the skeletal description of a string (a program in some programming language or a sentence in a natural language), an expression yielding a graph in a context-free graph language [Eng], or a description of a picture in some kind of picture language [Dre1], [Dre2]. In all these cases the trees are viewed as expressions that denote the actual objects of interest. Hence, one may envision future applications that learn languages of such objects by internally constructing a suitable fta but on the outside dealing only with strings, graphs, pictures, or the like.

The approach presented in this article improves the one in [DH]. The biggest disadvantage of the method used there was that the avoidance of dead states was an added feature instead of being an inherent property of the approach. Consequently, it did not lead to a provable reduction in complexity even though it was expected to improve the running time in practice. As explained above, the present paper solves this problem.

The paper is structured as follows. In the next section we collect some basic notions regarding trees and tree automata. Section 3 is devoted to our basic algorithm and illustrates it by means of an example. The correctness and complexity of this algorithm is addressed in Section 4. Section 5 concludes the paper with a short discussion.

2. Trees and Tree Automata

The set of natural numbers (including 0) is denoted by $\mathbb{N}$. For $n \in \mathbb{N}$, $[n]$ denotes the set $\{1, \ldots, n\}$. The cardinality of a set S is denoted by $|S|$ and the symmetric difference of S and another set S' is $S \ominus S' = (S \setminus S') \cup (S' \setminus S)$. Given a function $f\colon A \to B$, we

denote the canonical extensions of f to subsets of A and to sequences over A by f as well. Hence, $f(S) = \{f(a) \mid a \in S\}$ for all $S \subseteq A$ and $f(a_1 \cdots a_k) = f(a_1) \cdots f(a_k)$ for $a_1, \ldots, a_k \in A$.

A *ranked alphabet* is a finite set of symbols $\Sigma = \bigcup_{k \in \mathbb{N}} \Sigma_{(k)}$ which is partitioned into pairwise disjoint subsets $\Sigma_{(k)}$. The symbols in $\Sigma_{(k)}$ are said to have rank k. We write a symbol a with rank k as $a^{(k)}$ if the rank cannot easily be determined from the context. The set T_Σ of all trees over Σ is defined inductively, as usual: It is the smallest set of strings over Σ such that $f t_1 \cdots t_k \in \mathrm{T}_\Sigma$ for every $f \in \Sigma_{(k)}$ and all $t_1, \ldots, t_k \in \mathrm{T}_\Sigma$. For the sake of better readability we write $f[t_1, \ldots, t_k]$ instead of $f t_1 \cdots t_k \in \mathrm{T}_\Sigma$ unless $k = 0$.

The size of a tree, denoted by $|t|$, is the number of (occurrences of) symbols in t. Thus, $|t| = 1 + \sum_{i=1}^{k} |t_i|$ for every tree $t = f[t_1, \ldots, t_k]$. Given a set T of trees, $\Sigma(T)$ denotes the set of all trees of the form $f[t_1, \ldots, t_k]$ such that $f \in \Sigma_{(k)}$ for some $k \in \mathbb{N}$ and $t_1, \ldots, t_k \in T$. A subset of T_Σ is called a *tree language*. We identify a tree language L with the corresponding predicate on trees, i.e.,

$$L(t) = \begin{cases} \mathit{true} & \text{if} \quad t \in L, \\ \mathit{false} & \text{otherwise.} \end{cases}$$

We frequently decompose a tree into a so-called context and a subtree. For this purpose, we reserve a special symbol $\square$ of rank 0 that does not occur in Σ. A tree $c \in \mathrm{T}_{\Sigma \cup \{\square\}}$ in which $\square$ occurs exactly once as a leaf is called a *context* (over Σ). The set of all contexts over Σ is denoted by C_Σ. Given a context $c \in \mathrm{C}_\Sigma$ and a tree s, we denote by $c[\![s]\!]$ the tree obtained by substituting s for the unique occurrence of $\square$ in c. If $L \subseteq \mathrm{T}_\Sigma$ is a tree language, then a tree s is a *forbidden subtree* (with respect to L) if $c[\![s]\!] \notin L$ for all contexts $c \in \mathrm{C}_\Sigma$.

We now recall the definition of bottom-up finite-state tree automata. We only work with deterministic automata, but they may be partial. Hence, a *bottom-up finite-state tree automaton* (fta, for short) is a tuple $A = (\Sigma, Q, \delta, F)$ where

- Σ is the ranked input alphabet,
- Q is the finite set of *states*,
- δ is a finite set of *transitions* of the form $(q_1 \cdots q_k, f, q)$ such that $f \in \Sigma_{(k)}$ and $q_1, \ldots, q_k, q \in Q$ for some $k \in \mathbb{N}$, and
- $F \subseteq Q$ is the set of *accepting* (or *final*) states.

Given $f \in \Sigma_{(k)}$ and $q_1, \ldots, q_k \in Q$, we require that there is at most one $q \in Q$ with $(q_1 \cdots q_k, f, q) \in \delta$. In other words, A is deterministic. Hence, δ may also be viewed as a partial *transition function* which, for $f \in \Sigma_{(k)}$ and $q_1, \ldots, q_k \in Q$, yields the resulting state $q = \delta(q_1 \cdots q_k, f)$ (or is undefined if no such q exists). We say that A is *total* if δ is a total function.

In the obvious way, δ extends to trees, yielding a partial function $\delta \colon \mathrm{T}_\Sigma \to Q$ as follows: for $t = f[t_1, \ldots, t_k] \in \mathrm{T}_\Sigma$, if $q_1 = \delta(t_1), \ldots, q_k = \delta(t_k)$ are defined, then $\delta(t) = \delta(q_1 \cdots q_k, f)$. Otherwise, $\delta(t)$ is undefined. The set of trees accepted by A is

$$L(A) = \{t \in \mathrm{T}_\Sigma \mid \delta(t) \text{ is defined and belongs to } F\}.$$

A tree language L is called a *regular tree language* if there exists an fta A such that $L = L(A)$. In this case, A is said to *recognize* L.

Example 2.1. As a running example, let $\Sigma = \Sigma_{(2)} \cup \Sigma_{(0)}$ where $\Sigma_{(2)} = \{a, b\}$ and $\Sigma_{(0)} = \{\varepsilon\}$, and consider the tree language

$$U = \{t \in \mathrm{T}_\Sigma \mid t = c[\![s]\!] \text{ for some } c \in \mathrm{C}_{\{a,\varepsilon\}} \text{ and } s \in \mathrm{T}_{\{b,\varepsilon\}}\}.$$

Thus, a tree is an element of U if it is composed of a context over a and ε, and a tree over b and ε.

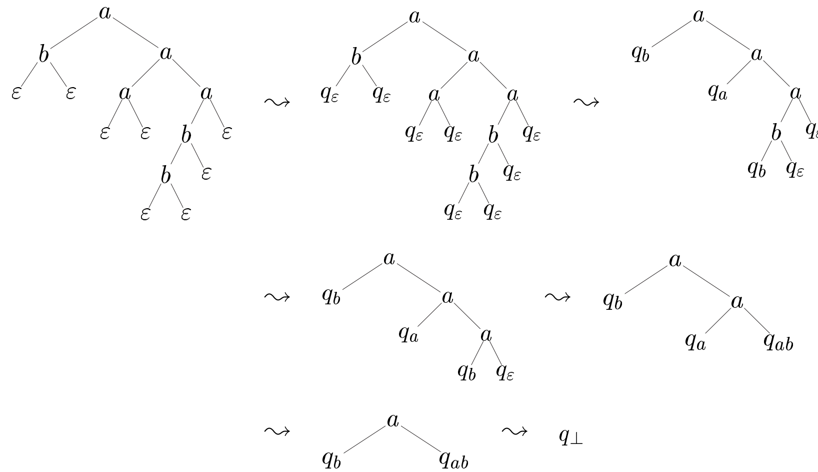
We discuss how to recognize U by means of a total fta. For this, we need a set Q of five states:

- a state q_ε such that $\delta(t) = q_\varepsilon$ if and only if $t = \varepsilon$,
- a state q_a such that $\delta(t) = q_a$ if and only if t contains a's but no b's,
- a state q_b such that $\delta(t) = q_b$ if and only if t contains b's but no a's,
- a state q_{ab} such that $\delta(t) = q_{ab}$ if and only if $t = c[\![s]\!]$ for some $c \in \mathrm{C}_{\{a,\varepsilon\}} \setminus \{\square\}$ and $s \in \mathrm{T}_{\{b,\varepsilon\}} \setminus \{\varepsilon\}$ (i.e., $t \in U$ and it contains both a's and b's), and
- a state $q_\perp$ such that $\delta(t) = q_\perp$ if and only if $t \notin U$.

Among these states, $q_\varepsilon, q_a, q_b, q_{ab}$ are the final ones; the transition function is given as follows:

$$\begin{aligned}
&\delta(\lambda, \varepsilon) = q_\varepsilon \\
&\delta(q_x q_y, a) = q_a && \text{for } x, y \in \{a, \varepsilon\}, \\
&\delta(q_x q_y, a) = q_{ab} && \text{for } x \in \{a, \varepsilon\} \text{ and } y \in \{b, ab\} \text{ or } x \in \{b, ab\} \\
& && \text{and } y \in \{a, \varepsilon\}, \\
&\delta(q_x q_y, b) = q_b && \text{for } x, y \in \{b, \varepsilon\}, \\
&\delta(q_x q_y, z) = q_\perp && \text{for all remaining cases.}
\end{aligned}$$

The (rejecting) computation on input $t = a[b[\varepsilon, \varepsilon], a[a[\varepsilon, \varepsilon], a[b[b[\varepsilon, \varepsilon], \varepsilon], \varepsilon]]]$ can be illustrated as follows, where the transitions "consume" the tree stepwise from the leaves towards the root:



Thus, $\delta(t) = q_\perp$, indicating that t does not belong to U.

Given an fta $A = (\Sigma, Q, \delta, F)$, it follows by an obvious induction from the definition of δ that any occurrence of a subtree s in a tree t may be replaced with any tree s' without affecting $\delta(t)$, provided that $\delta(s') = \delta(s)$.[2] This yields the following lemma.

Lemma 2.2. *Let $A = (\Sigma, Q, \delta, F)$ be an fta. For all contexts $c \in \mathrm{C}_\Sigma$ and all trees $s, s' \in \mathrm{T}_\Sigma$, if $\delta(s) = \delta(s')$ then $\delta(c[\![s]\!]) = \delta(c[\![s']\!])$.*

One consequence of the previous lemma is that dead states may be removed from A without affecting $L(A)$. Here, a *dead state* is a state $q \in Q$ such that $\delta(s) = q$ for some forbidden subtree $s \in \mathrm{T}_\Sigma$. Given such a state, Lemma 2.2 yields that $\delta(c[\![s']\!]) = \delta(c[\![s]\!])$ for all $s' \in \mathrm{T}_\Sigma$ with $\delta(s') = q$. In other words, all trees $s' \in \mathrm{T}_\Sigma$ with $\delta(s') = q$ are forbidden subtrees. Now, if we remove q from Q and restrict δ accordingly, δ becomes undefined only on trees outside $L(A)$ (namely on trees containing forbidden subtrees), which guarantees that the new automaton also recognizes $L(A)$. For instance, in Example 2.1 the trees s with $\delta(s) = q_\perp$ are forbidden subtrees. Thus, we may turn the automaton into a partial fta that recognizes the same language, by removing $q_\perp$. Only 17 out of 51 transitions would be left. In general, it is worthwhile noticing that the removal of dead states may drastically reduce the size of an automaton even if there is only one dead state. This is because a total fta has about $|\Sigma| \cdot |Q|^r$ transitions, where r is the maximum rank of symbols in Σ. In the extreme case, all but $|Q| - 1$ out of these transitions may involve the dead state.

It is well known that the Myhill–Nerode theorem carries over to regular tree languages. As a consequence, every regular tree language comes with a unique minimal fta recognizing it. This result is usually stated for total automata but holds for partial ones as well. Since our aim is to find a minimal partial fta for the language to be learned, we briefly discuss both variants.

Consider a tree language $L \subseteq \mathrm{T}_\Sigma$. Given two trees $s, s' \in \mathrm{T}_\Sigma$, let $s \sim_L s'$ if and only if, for every context $c \in \mathrm{C}_\Sigma$, $L(c[\![s]\!]) = L(c[\![s']\!])$. Thus, either both of $c[\![s]\!]$ and $c[\![s']\!]$ are in L or none of them is. In other words, $s \sim_L s'$ if s and s' can be substituted for each other in any context without affecting membership in L. Obviously, $\sim_L$ is an equivalence relation on T_Σ. The equivalence class containing $s \in \mathrm{T}_\Sigma$ is denoted by $[s]_L$. The *index* of L is the number of equivalence classes of $\sim_L$.

In general, the index of a tree language may of course be infinite. However, for an fta A, the index of $L(A)$ is always finite since $\delta(s) = \delta(s')$ implies $s \sim_{L(A)} s'$ by Lemma 2.2. Conversely, if a tree language L is of finite index, we can easily build a total fta $A = (\Sigma, Q, \delta, F)$ recognizing L. Its states are the equivalence classes of $\sim_L$ and δ is the set $\{\mathit{trans}(s) \mid s \in \mathrm{T}_\Sigma\}$, where for $s = f[s_1, \ldots, s_k]$,

$$\mathit{trans}(s) = ([s_1]_L \cdots [s_k]_L, f, [s]_L).$$

Similar to the string case, it is easy to check that δ is well defined because $[s]_L$ depends only on $[s_1]_L, \ldots, [s_k]_L$ rather than on the particular choice of $s_1, \ldots, s_k$. An equivalence class $[s]_L$ is a final state if $s \in L$. By construction, $\delta(s) = [s]_L$ for all $s \in \mathrm{T}_\Sigma$. Hence, by the definition of final states, it is indeed true that $L(A) = L$. Furthermore, it is not

[2] Whenever we write $\delta(s) = \delta(s')$, this is meant to express the fact that $\delta(s)$ and $\delta(s')$ are either both undefined or are defined and equal.

difficult to show that A is the uniquely determined minimal total fta recognizing L (up to a bijective renaming of states).

The automaton constructed above may of course contain a dead state (but not more than one since forbidden subtrees are pairwise equivalent). Hence, the construction can be improved to yield the uniquely determined minimal partial fta recognizing L, as follows. The state set Q is now the set of all equivalence classes $[s]_L$ such that $s \in \mathrm{T}_\Sigma$ is not a forbidden subtree. Similarly, $\delta = \{trans(s) \mid s \in \mathrm{T}_\Sigma$ and s is not a forbidden subtree$\}$.

Minimality and uniqueness are obvious, using the uniqueness of the minimal total fta. This is because a total fta can be obtained by adding a dead state (if required), together with the missing transitions. As the resulting total fta is unique, there cannot be two minimal partial ones (since that would give rise to distinct total ones, both of which would be minimal).

As an example, one may easily check that the index of the tree language L in Example 2.1 is (at least) 5, by verifying that the trees ε, $a[\varepsilon, \varepsilon]$, $b[\varepsilon, \varepsilon]$, $a[b[\varepsilon, \varepsilon], \varepsilon]$, and $a[b[\varepsilon, \varepsilon], b[\varepsilon, \varepsilon]]$ belong to different equivalence classes with respect to $\sim_L$. Since the fta constructed in that example has five states, we conclude that the index of L is indeed 5 and the fta discussed in the example is the unique minimal total fta recognizing L. To construct the minimal partial fta we simply omit the state $q_\perp$ and all transitions involving it, thus obtaining an fta with 17 transitions instead of 51.

3. Learning a Regular Tree Language

This section is devoted to the description of the learning algorithm. For the rest of the paper, we consider an arbitrary ranked alphabet Σ. The aim is to learn an unknown regular tree language $U \subseteq \mathrm{T}_\Sigma$, by constructing an fta recognizing U. Following the MAT model described in the Introduction, the learner has access to an oracle. This *teacher* is able to answer membership and equivalence queries:

Membership query. Given some tree $t \in \mathrm{T}_\Sigma$, the teacher checks whether or not $t \in U$ and returns $U(t)$.

Equivalence query. Given an fta A, if $L(A) \neq U$ the teacher returns any tree $\text{COUNTEREXAMPLE}(A) \in U \ominus L(A)$, a tree disproving the conjecture that A recognizes U. If $L(A) = U$, then the teacher returns the special symbol $\perp \notin \Sigma$, thus indicating successful completion of the learning process.

The main algorithmic idea behind the learner is inspired by the (theoretical) construction of the minimal partial fta $A = (\Sigma, Q, \delta, F)$ recognizing U. Recall from the discussion at the end of Section 2 that Q is the set of all $[s]_U$ such that $s \in \mathrm{T}_\Sigma$ is not a forbidden subtree. Thus, to describe the state set Q of A, we need $|Q|$ pairwise nonequivalent trees. Similarly, the transitions in δ are those of the form

$$trans(s) = ([s_1]_U \cdots [s_k]_U, f, [s]_U),$$

where $s = f[s_1, \ldots, s_k] \in \mathrm{T}_\Sigma$ is not a forbidden subtree. We can say that the tree s represents the given transition. In order to describe δ we need $|\delta|$ trees representing pairwise distinct transitions. Note that, except for the trivial case where two trees have

different root symbols, $s = f[s_1, \ldots, s_k]$ and $s' = f[s'_1, \ldots, s'_k]$ represent distinct transitions if and only if $[s_i]_U \neq [s'_i]_U$ for some $i \in [k]$.

In order to construct Q and δ, the learner will thus collect finite sets S and R of trees representing states, respectively transitions. However, the learner also needs some means to verify that two trees in S belong to different equivalence classes. Therefore, the learner will not only collect trees in S and R, but will also maintain a set C of contexts containing, for all distinct trees $s, s' \in S$, a context c such that $U(c[\![s]\!]) \neq U(c[\![s']\!])$. In fact, the contexts in C will also verify that the trees in R represent pairwise distinct transitions. This is because we shall build R in such a way that $R \subseteq \Sigma(S)$. Hence, in the situation discussed above, there will always be a context $c \in C$ such that $U(c[\![s_i]\!]) \neq U(c[\![s'_i]\!])$ for some $i \in [k]$ if $s = f[s_1, \ldots, s_k]$ and $s' = f[s'_1, \ldots, s'_k]$ are distinct trees in R.

We now prepare for the description of the actual learning algorithm. At any stage of the computation, the learner will maintain the mentioned sets S and R of trees, and a finite set of contexts $C \subseteq \mathrm{C}_\Sigma$. The first two sets will be related in a rather special way: It will always be true that $S \subseteq R \subseteq \Sigma(S)$. Following the approach of [Ang1], the algorithm builds a so-called observation table whose rows and columns are indexed by the elements of R, respectively C. The cell in row $s \in R$ and column $c \in C$ contains an *observation*—a truth value indicating whether $c[\![s]\!] \in U$. Note that the contents of each cell of the table (for a fixed language U) is uniquely determined by R and C and can be discovered by a membership query (although a concrete implementation should of course store this information in order to avoid asking the same questions over and over again). For this reason, we simply call the triple (S, R, C) an observation table, thus omitting an explicit representation of the cells. The precise definition reads as follows.

Definition 3.1 (Observation Table). Given a set $C \subseteq \mathrm{C}_\Sigma$ of contexts and a tree $s \in \mathrm{T}_\Sigma$, we denote by $obs_C(s)$ the function $f\colon C \to \{true, false\}$ such that $f(c) = U(c[\![s]\!])$ for all $c \in C$. An *observation table* is a triple $T = (S, R, C)$ of finite sets $S, R \subseteq \mathrm{T}_\Sigma$ and $C \subseteq \mathrm{C}_\Sigma$ such that

1. $S \subseteq R \subseteq \Sigma(S)$, and
2. $obs_C(s) \neq obs_C(s')$ for all distinct $s, s' \in S$.

Readers who are familiar with the papers of Angluin and Sakakibara may notice that the requirement $S \subseteq \Sigma(S)$ corresponds to the prefix-closedness, respectively subtree-closedness, in their works. Also, it may be helpful to have another look at the second requirement in the definition. In the terminology of [Ang1], this implies that our observation tables are automatically consistent. Thus, an explicit consistency requirement as in [Ang1] and [Sak1] is not needed.

The reader should notice the connection between $obs_C(s)$ and $\sim_U$. Clearly, $obs_{\mathrm{C}_\Sigma}(s) = obs_{\mathrm{C}_\Sigma}(s')$ if and only if $[s]_U = [s']_U$. For smaller sets $C \subseteq \mathrm{C}_\Sigma$, it follows at least that $obs_C(s) \neq obs_C(s')$ implies $[s]_U \neq [s']_U$. One task of the learner is to find suitable contexts to separate all equivalence classes of $\sim_U$ from each other.

The observation table maintained by the learner will always have two additional properties; these are defined next.

Definition 3.2 (Complete and Small Observation Table). An observation table (S, R, C) is *complete* if

$$obs_C(R) \subseteq obs_C(S).$$

It is *small* if

1. R does not contain any forbidden subtree, and
2. $|C| \leq |S|$.

As in the construction of the minimal partial fta recognizing U, a tree can be turned into a transition based on the information provided by a set $C \subseteq \mathrm{C}_\Sigma$ of contexts. For a tree $s = f[s_1, \ldots, s_k]$, we define the corresponding transition by

$$trans_C(s) = (obs_C(s_1 \cdots s_k), f, obs_C(s)).$$

In other words, the construction takes observations for equivalence classes even if the former may not yet reveal the whole truth about the latter. Given a complete observation table $T = (S, R, C)$, we use this construction to define an fta $\text{SYNTHESIZE}(T) = (\Sigma, Q_T, \delta_T, F_T)$, as follows:

- $Q_T = obs_C(S)$,
- $\delta_T = trans_C(R)$, and
- $F_T = \{obs_C(s) \mid s \in S \cap U\}$.

It is important to notice that, by the two requirements of Definition 3.1, $|Q_T| = |S|$ and $|\delta_T| = |R|$. As we shall see, this guarantees termination of the learning algorithm and allows us to derive an upper bound on its running time.

Note that $\text{SYNTHESIZE}(T)$ can be easily constructed. Assuming that the implementation of the observation table explicitly stores the boolean entries of the row $obs_C(s)$ for every $s \in R$, all information needed to build Q_T and δ_T can be retrieved from the table. To construct F_T, for every $s \in S$ one membership query is needed to find out whether $s \in U$ (unless this information has already been requested in some earlier step and is stored in memory).

Lemma 3.3. *For every complete observation table T, $\text{SYNTHESIZE}(T)$ is a well-defined fta.*

Proof. It suffices to show that δ_T is well defined. If $s = f[s_1, \ldots, s_k]$ is a tree in R then, by the completeness of T, $obs_C(s) \in obs_C(S)$ and, since $s \in \Sigma(S)$, also $obs_C(s_1 \cdots s_k) \in obs_C(S)^k$. Hence, $trans_C(s) \in Q_T^k \times \Sigma_{(k)} \times Q_T$. Further, if $s' = f[s'_1, \ldots, s'_k] \in R$ is distinct from s then $obs_C(s_1 \cdots s_k) \neq obs_C(s'_1 \cdots s'_k)$, by the second requirement in Definition 3.1 and the fact that $s_1, \ldots, s_k, s'_1, \ldots, s'_k \in S$. This proves that δ_T is a partial function. □

Before presenting the learning algorithm itself, we study the relation between $\text{SYNTHESIZE}(T)$ and T. We first state and verify the rather obvious property of δ_T that $\delta_T(s) = obs_C(s)$ for all $s \in R$.

Lemma 3.4. *For every complete observation table $T = (S, R, C)$ and every tree $s \in R$, it holds that $\delta_T(s) = obs_C(s)$.*

Proof. We proceed by structural induction on $s = f[s_1, \ldots, s_k]$. Since $R \subseteq \Sigma(S)$ we have $s_1, \ldots, s_k \in S \subseteq R$ and thus, by the induction hypothesis, $\delta_T(s_1 \cdots s_k) = obs_C(s_1 \cdots s_k)$. By the definition of δ_T, it contains the transition $trans_C(s) = (obs_C(s_1 \cdots s_k), f, obs_C(s))$, which readily implies $\delta_T(s) = obs_C(s)$. □

As an almost immediate consequence, we get the following lemma.

Lemma 3.5. *For every complete observation table $T = (S, R, C)$ and every tree $s \in S$,* SYNTHESIZE(T) *accepts s if and only if $s \in U$.*

Proof. By Lemma 3.4, SYNTHESIZE(T) accepts s if and only if there is some $s' \in S \cap U$ with $obs_C(s') = obs_C(s)$. However, by the second requirement of Definition 3.1, $obs_C(s') = obs_C(s)$ is equivalent to $s' = s$. □

An observation table $T = (S, R, C)$ can be easily turned into a complete observation table, denoted by COMPLETE(T). One simply chooses, for each $o \in obs_C(R) \backslash obs_C(S)$, some tree $s \in R$ with $obs_C(s) = o$ and adds it to S. In other words, COMPLETE$(T) = (S \cup S', R, C)$ where S' is some minimal subset of R with $obs_C(S') = obs_C(R) \setminus obs_C(S)$.

Lemma 3.6. *For every observation table T,* COMPLETE(T) *is a complete observation table. If T is small, then so is* COMPLETE(T).

Proof. Obvious. □

We also need the following easy lemma.

Lemma 3.7. *For every complete observation table $T = (S, R, C)$ and every tree $s = f[s_1, \ldots, s_k] \in \Sigma(S)$, if $\delta_T(obs_C(s_1 \cdots s_k), f)$ is defined, then $s \in R$.*

Proof. If $\delta_T(obs_C(s_1 \ldots s_k), f)$ is defined, then there is some $s' = f[s'_1, \ldots, s'_k] \in R \subseteq \Sigma(S)$ such that $obs_C(s_1 \cdots s_k) = obs_C(s'_1 \cdots s'_k)$. By the second requirement of Definition 3.1, $s_1 = s'_1, \ldots, s_k = s'_k$ and thus $s = s'$. □

We now turn to the description of the learning algorithm. The learner starts with an initial observation table $T = (\emptyset, \emptyset, \emptyset)$ and enters directly into the main loop. Here it synthesizes an fta A and asks for a counter-example, receiving in response a tree in $U \ominus L(A)$. From this counter-example, the learner selects a piece of information and uses it to extend the observation table before it continues by repeating the whole process. This results in the following overall structure:

$T = (S, R, C)$:= $(\emptyset, \emptyset, \emptyset)$;
`loop`

```
    A  := SYNTHESIZE(T);
    t  := COUNTEREXAMPLE(A);        (equivalence query)
    if t = ⊥ then return A
    else T  := EXTEND(T, t)
end loop
```

Line four of the routine is the only place where our algorithm uses an equivalence query. Exactly how the learner uses the returned counter-example to gain greater knowledge about the target language deserves a detailed explanation. A counter-example t is by definition a tree which A fails to classify correctly as a member/nonmember of the target language U. There are two possible reasons for this inaccuracy: either $t \in U$ but A lacks a transition (so that $\delta_T(t)$ is undefined) or $\delta_T(t) = obs_C(t')$ for some tree $t' \in S$ such that $t' \not\sim_U t$. Now, the idea behind the procedure EXTEND is to determine one reason (although there can be more than one) why δ_T fails to classify t correctly. This is done by simulating the run of A on t: The procedure first decomposes t into $c[\![s]\!]$ where $s = f[s_1, \ldots s_k]$ is an element of $\Sigma(S)\backslash S$. Thus, intuitively, s is a minimal subtree of t not contained in S (minimal in the sense that the direct subtrees of s belong to S). Such a subtree s must exist in every counter-example because, by Lemma 3.5, no counter-example can be an element of S. If $s \notin R$ then we know by Lemmas 3.7 and 3.4 that $\delta_T(s) = \delta_T(\delta_T(s_1) \cdots \delta_T(s_k), f)$ is undefined. In this case $\delta_T(t)$ is undefined as well, and thus t is rejected. Hence, $t \in U$ because t is a counter-example, showing that s is not a forbidden subtree and should be added to R in order to provide δ_T with the missing transition.

On the other hand, if $s \in R$, then the learner checks the unique tree s' in S such that $obs_C(s) = obs_C(s')$. There are two cases. If $U(c[\![s]\!]) \neq U(c[\![s']\!])$ we know that $s \not\sim_U s'$ although A cannot distinguish between them. As a remedy, we add s to S and c to C. The new context distinguishes between s and s' and guarantees that henceforth $obs_C(s) \neq obs_C(s')$. Finally, if $U(c[\![s]\!]) = U(c[\![s']\!])$ then $c[\![s']\!]$ is also a counter-example since we have $\delta_T(s') = obs_C(s') = obs_C(s) = \delta_T(s)$ (using Lemma 3.4 twice) and thus, by Lemma 2.2, $c[\![s]\!] \in L(A)$ if and only if $c[\![s']\!] \in L(A)$. EXTEND then calls itself recursively with the parameter $c[\![s']\!]$. Intuitively, the procedure goes up in the tree until it finds a place where the behaviour of the synthesized fta contradicts the information given by the teacher. As mentioned in the Introduction, this is basically the contradiction backtracking technique invented by Shapiro [Sha].

In each recursive call of EXTEND, the number of occurrences of subtrees which are not elements of S is reduced by at least one. Consequently, EXTEND(T, t) terminates after fewer than $|t|$ recursive calls. An outline of the procedure EXTEND is listed below:

```
procedure EXTEND(T, t) where T = (S, R, C)
    decompose t into t = c[[s]] where s = f[s_1, ..., s_k] ∈ Σ(S)\S;
    if s ∈ R then
        let s' ∈ S be such that obs_C(s') = obs_C(s);
        if U(c[[s']]) = U(c[[s]]) then                (membership query)
            return EXTEND(T, c[[s']])
        else return COMPLETE(S ∪ {s}, R, C ∪ {c})
    else return COMPLETE(S, R ∪ {s}, C)
```

Notice the membership query in line five. (Assuming that an explicit observation table is used, indeed only one membership query is needed since $U(c[\![s']\!]) = obs_C(s')(c)$ is already known.) In addition, a concrete implementation which explicitly maintains the entries of the observation table must use membership queries when executing one of the two last lines. If COMPLETE($S \cup \{s\}$, R, $C \cup \{c\}$) is returned, the new column given by the new context c must be filled with the respective boolean values. Hence, in this case $|R|$ membership queries are needed. In contrast, when returning COMPLETE(S, $R \cup \{s\}$, C), the new row represented by s must be established, thus leading to $|C|$ membership queries. Of course, one can delay updating the table until SYNTHESIZE is called again.

We postpone a more detailed discussion of these issues including formal correctness and complexity statements to the next section. We now continue Example 2.1 and use the rest of this section to illustrate how the learner acts in order to learn that language. Hence, $\Sigma = \Sigma_{(2)} \cup \Sigma_{(0)}$ where $\Sigma_{(2)} = \{a, b\}$ and $\Sigma_{(0)} = \{\varepsilon\}$, and U is the set of all trees of the form $c[\![s]\!]$, where $c \in \mathrm{C}_{\{a,\varepsilon\}}$ and $s \in \mathrm{T}_{\{b,\varepsilon\}}$. In the tables of this example we use "+" and "−" instead of "*true*" and "*false*" in order to indicate membership, respectively nonmembership. Furthermore, we denote the sequence of tables generated during the execution of the learner by $T_1, T_2, T_3, \ldots$, and the automata SYNTHESIZE(T_i) by A_i. In each table, rows are indexed by elements of R and columns by elements of C. The upper section of each table displays the rows given by the trees in S while the lower part lists those given by trees in $R \backslash S$.

The learner starts its computation with the initial observation table $T_1 = (\emptyset, \emptyset, \emptyset)$. Since $A_1 = $ SYNTHESIZE(T_1) only recognizes the empty language the teacher will return as a counter-example some tree in U, say $t_1 = a[a[\varepsilon, \varepsilon], b[\varepsilon, \varepsilon]]$. EXTEND then selects a subtree from the counter-example in $\Sigma(S) \backslash S$, which in this specific situation can only be ε since $S = \emptyset$. Hence ε is added to R and since S is empty, also to S by COMPLETE . The observation table becomes

$T_2 =$

ε	

An fta A_2 synthesized from T_2 has only a single state $q = obs_{\emptyset}(\varepsilon)$ and a single transition $(\lambda, \varepsilon, q)$ and thus rejects t_1 which is in U, so the teacher may use the same counter-example again. This time EXTEND chooses $s = a[\varepsilon, \varepsilon]$ to be added to R, but as $obs_{\emptyset}(\varepsilon) = obs_{\emptyset}(a[\varepsilon, \varepsilon])$, s will not be added to S by COMPLETE . The observation table at this point is

$T_3 =$

ε	
$a[\varepsilon, \varepsilon]$	

The learner has not yet discovered that ε and $a[\varepsilon, \varepsilon]$ are inequivalent but knows already that they are not forbidden subtrees (and are, in fact, elements of U). As A_3 fails to accept $b[\varepsilon, \varepsilon]$ this may be the next counter-example given. If so then R grows to three elements while S remains unchanged. We omit T_4 for brevity, but remark that $L(A_4) = \mathrm{T}_{\Sigma}$. A fourth counter-example, $t = b[a[a[\varepsilon, \varepsilon], \varepsilon], a[\varepsilon, \varepsilon]] \in L(A_4) \backslash U$, is now provided by the teacher as a response to the fta A_4 . The procedure EXTEND selects a subtree of t in $\Sigma(S_4)$, say the leftmost $a[\varepsilon, \varepsilon]$, tries to replace it with the tree in S_4 whose

observed behaviour is the same (i.e., with ε), and checks whether the resulting tree is also a counter-example. Thus, t is turned into the counter-example $t' = b[a[\varepsilon, \varepsilon], a[\varepsilon, \varepsilon]]$ and EXTEND is called recursively. Again, some $a[\varepsilon, \varepsilon]$ can be replaced with ε (say the leftmost one), leading to a recursive call with the counter-example $t'' = b[\varepsilon, a[\varepsilon, \varepsilon]]$. Here, the remaining occurrence of the subtree $a[\varepsilon, \varepsilon]$ cannot be replaced with ε because $b[\varepsilon, \varepsilon] \in U$. Hence, EXTEND returns $T_5 = (S_4 \cup \{a[\varepsilon, \varepsilon]\}, R, C_4 \cup \{b[\varepsilon, \square]\})$, i.e., the following (complete) table:

$T_5 =$

	$b[\varepsilon, \square]$
ε	$+$
$a[\varepsilon, \varepsilon]$	$-$
$b[\varepsilon, \varepsilon]$	$+$

However, the fta A_5 still maps $b[\varepsilon, \varepsilon]$ to the same state as ε, so the teacher gives yet another counter-example, $a[b[\varepsilon, \varepsilon], b[\varepsilon, \varepsilon]] \in L(A_5) \setminus U$. EXTEND decomposes this into $a[b[\varepsilon, \varepsilon], \square][\![b[\varepsilon, \varepsilon]]\!]$, causing $b[\varepsilon, \varepsilon]$ to be added to S and $a[b[\varepsilon, \varepsilon], \square]$ to C:

$T_6 =$

	$b[\varepsilon, \square]$	$a[b[\varepsilon, \varepsilon], \square]$
ε	$+$	$+$
$a[\varepsilon, \varepsilon]$	$-$	$+$
$b[\varepsilon, \varepsilon]$	$+$	$-$

There is still a state missing though. The learner will first discover that one of the trees $a[\varepsilon, b[\varepsilon, \varepsilon]]$ and $a[b[\varepsilon, \varepsilon], \varepsilon]$ has to be added to R. Afterwards, COMPLETE will move the tree further into S, where it will give rise to the missing state. What remains is a series of synthesized fta's and counter-examples that will fill in the missing rules, yielding in step 21 the final observation table

$T_{21} =$

	$b[\varepsilon, \square]$	$a[b[\varepsilon, \varepsilon], \square]$
ε	$+$	$+$
$a[\varepsilon, \varepsilon]$	$-$	$+$
$b[\varepsilon, \varepsilon]$	$+$	$-$
$a[b[\varepsilon, \varepsilon], \varepsilon]$	$-$	$-$
$a[\varepsilon, a[\varepsilon, \varepsilon]]$	$-$	$+$
$a[a[\varepsilon, \varepsilon], \varepsilon]$	$-$	$+$
$a[a[\varepsilon, \varepsilon], a[\varepsilon, \varepsilon]]$	$-$	$+$
$a[\varepsilon, b[\varepsilon, \varepsilon]]$	$-$	$-$
$a[a[\varepsilon, \varepsilon], b[\varepsilon, \varepsilon]]$	$-$	$-$
$a[b[\varepsilon, \varepsilon], a[\varepsilon, \varepsilon]]$	$-$	$-$
$a[\varepsilon, a[b[\varepsilon, \varepsilon], \varepsilon]]$	$-$	$-$
$a[a[b[\varepsilon, \varepsilon], \varepsilon], \varepsilon]$	$-$	$-$
$a[a[b[\varepsilon, \varepsilon], \varepsilon], a[\varepsilon, \varepsilon]]$	$-$	$-$
$a[a[\varepsilon, \varepsilon], a[b[\varepsilon, \varepsilon], \varepsilon]]$	$-$	$-$
$b[b[\varepsilon, \varepsilon], \varepsilon]$	$+$	$-$
$b[\varepsilon, b[\varepsilon, \varepsilon]]$	$+$	$-$
$b[b[\varepsilon, \varepsilon], b[\varepsilon, \varepsilon]]$	$+$	$-$

If we compare the fta A_{21} with the total fta described in Example 2.1 we see that the states $\langle ++\rangle, \langle -+\rangle, \langle +-\rangle, \langle --\rangle$ are renamings of the states q_ε, q_a, q_b, and q_{ab}, respectively. However, since A_{21} is the minimal *partial* fta recognizing U there is no state corresponding to $q_\perp$. Note that, indeed, only 17 instead of 51 transitions have been computed, as mentioned in Example 2.1.

4. Correctness and Complexity

In this section we prove that the learner is correct, i.e., it returns the minimal partial fta recognizing U, and examine its time complexity. For the complexity considerations, we assume that the teacher can answer any query in constant time. In other words, we do not take into account the complexity of the teacher (which, of course, cannot be known unless we make further assumptions regarding the teacher). We assume that the algorithm is implemented on a Random Access Machine (RAM). Such a machine has by definition a potentially infinite number of registers, each capable of storing an arbitrarily large integer. Its program is a sequence of instructions like assignment, addition, and conditional jumps. Moreover, what is most important to our algorithm, it supports indirect addressing, i.e., the contents of a register may be used to address another register in order to retrieve or store a value. Thus, we can use pointers in order to deal with trees in an efficient way. For more information on RAMs the reader is referred to, e.g., the textbook by Papadimitriou [Pap].

The correctness of the learner is obvious once termination has been proved, simply because the learner will only return an automaton accepted by the teacher. To prove termination, we basically have to show that the learner never collects useless information. At the same time, this will enable us to estimate the time complexity of the learner. To start, we give two lemmas showing that the recursion of EXTEND works correctly and a small complete observation table remains small when the learner enlarges it by the information extracted from a counter-example.

The assumptions of the first lemma correspond to the situation in EXTEND where $s \in R$, with the two subcases that EXTEND either calls itself recursively or returns COMPLETE$(S \cup \{s\}, R, C \cup \{c\})$.

Lemma 4.1. *Let $T = (S, R, C)$ be a small complete observation table and let $A =$ SYNTHESIZE(T). Let $c \in \mathrm{C}_\Sigma$ and $s \in R$ be such that $c[\![s]\!] \in L(A) \ominus U$, and let s' be the unique tree in S with $obs_C(s') = obs_C(s)$. If $U(c[\![s']\!]) = U(c[\![s]\!])$ then $c[\![s']\!] \in L(A) \ominus U$. Otherwise, $(S \cup \{s\}, R, C \cup \{c\})$ is a small observation table.*

Proof. By Lemma 3.4, $\delta_T(s) = obs_C(s) = obs_C(s') = \delta_T(s')$. Consequently, by Lemma 2.2, $\delta_T(c[\![s']\!]) = \delta_T(c[\![s]\!])$, which in particular implies that $c[\![s']\!] \in L(A)$ if and only if $c[\![s]\!] \in L(A)$. Thus, the assumption $c[\![s]\!] \in L(A) \ominus U$ yields $c[\![s']\!] \in L(A) \ominus U$ if $U(c[\![s']\!]) = U(c[\![s]\!])$. This proves the first statement of the lemma.

Suppose now that $U(c[\![s']\!]) \neq U(c[\![s]\!])$. We must prove that $T' = (S \cup \{s\}, R, C \cup \{c\})$ is a small observation table. Obviously, the first requirement of Definition 3.1 is satisfied. As for the first requirement, note that $obs_C(t) \neq obs_C(t')$ implies $obs_{C\cup\{c\}}(t) \neq obs_{C\cup\{c\}}(t')$ for all $t, t' \in \mathrm{T}_\Sigma$. Thus, it suffices to prove that $obs_{C\cup\{c\}}(s'') \neq obs_{C\cup\{c\}}(s)$

for all $s'' \in S$. For $s'' = s'$ this is clear as $U(c[\![s']\!]) \neq U(c[\![s]\!])$. For $s'' \neq s'$ we already have $obs_C(s'') \neq obs_C(s') = obs_C(s)$ as (S, R, C) is an observation table, and hence $obs_{C\cup\{c\}}(s'') \neq obs_{C\cup\{c\}}(s)$.

Thus, T' is an observation table. Furthermore, T' obviously satisfies the smallness conditions (due to the assumption that T satisfies them), which completes the proof. □

The second lemma captures the case where EXTEND discovers that $s \notin R$ and returns COMPLETE$(S, R \cup \{s\}, C\}$.

Lemma 4.2. *Let $T = (S, R, C)$ be a small complete observation table and let $A =$ SYNTHESIZE(T). If $c \in \mathrm{C}_\Sigma$ and $s \in \Sigma(S)\backslash R$ are such that $c[\![s]\!] \in L(A) \ominus U$, then $(S, R \cup \{s\}, C)$ is a small observation table.*

Proof. Since $s \in \Sigma(S)$, it is clear that $(S, R \cup \{s\}, C)$ is an observation table. To convince ourselves that it is also small, we have to argue that s is not a forbidden subtree. Let $s = f[s_1, \ldots, s_k]$. By Lemma 3.4 and since $s_1, \ldots, s_k \in S \subseteq R$, we know that $\delta_T(s) = \delta_T(obs_C(s_1 \cdots s_k), f)$, and since the latter is undefined, so is $\delta_T(c[\![s]\!])$. This shows that $c[\![s]\!] \notin L(A)$, which means that $c[\![s]\!] \in U$, thus proving the claim that s is not a forbidden subtree. □

We are now ready to show that the learner works correctly and returns the minimal partial fta recognizing U.

Theorem 4.3. *The learner terminates and returns the minimal partial fta recognizing U (up to renaming of states).*

Proof. Let $A = (\Sigma, Q, \delta, F)$ be the minimal partial fta recognizing U, where $Q = \{[s]_U \mid s \in \mathrm{T}_\Sigma \text{ is not a forbidden subtree}\}$ (see Section 2). The learner will call the procedure EXTEND in every iteration of the main loop (unless there is no counter-example, in which case the learner halts). Each of these calls terminates (as argued in the paragraph preceding EXTEND) and will result in a new element being added to at least one of the sets S and R. Initially, $T = (\emptyset, \emptyset, \emptyset)$ is a small complete observation table. By induction, using Lemmas 3.6, 4.1, and 4.2, this means that $T = (S, R, C)$ is always a small complete observation table.

The fact that T is a small observation table implies in particular that $|S| \leq |Q|$ and $|R| \leq |\delta|$. To see this, first note that the smallness of T implies $\{[s]_U \mid s \in S\} \subseteq Q$ and $trans(R) \subseteq \delta$ as it disallows R (and hence S) to contain forbidden subtrees. Now recall the remark after Definition 3.1 saying that $obs_C(s) \neq obs_C(s')$ implies $[s]_U \neq [s']_U$. Hence, $|S| \leq |Q|$ is a consequence of the second requirement of Definition 3.1. Similarly, $|R| \leq |\delta|$ since the second requirement yields $trans(s_1) \neq trans(s_2)$ for all distinct $s_1, s_2 \in \Sigma(S) \supseteq R$.

The two upper bounds assure that the learner will eventually terminate and return a partial fta $A_T = (\Sigma, Q_T, \delta_T, F_T)$. This fta recognizes U since the teacher has accepted it. Moreover, by the inequalities above, $|Q_T| \leq |Q|$ (and $|\delta_T| \leq |\delta|$). Due to the uniqueness of the minimal partial fta recognizing U, this means that A and A_T are identical up to the names of states. □

We now study the time complexity of the learner. The running time of the algorithm will be measured in the size of the sought fta and the maximum size m of the counter-examples returned by the teacher. The former is furthermore decomposed into three parameters, namely the number of states, number of transitions, and maximum rank of symbols. As in the Introduction, we denote the latter by r. More precisely, if Σ is the smallest ranked alphabet satisfying $U \subseteq T_\Sigma$ then r denotes the largest natural number such that $\Sigma_{(r)} \neq \emptyset$ (and $r = 0$ if $\Sigma = \emptyset$).

Note that we need m as a parameter even though the procedure EXTEND avoids adding large counter-examples and their subtrees to S and R. This is because unnecessarily large counter-examples clearly influence the running time of the procedure EXTEND itself in a negative way. Hence, the overall running time of the learner depends to a certain degree on m as well.

To gain efficiency (and avoid repeated membership queries) we assume in the following that the learner explicitly maintains, for every $s \in R$, a reference to a memory cell holding $obs_C(s)$, where $T = (S, R, C)$ is the current observation table. Here, $obs_C(s)$ is stored as a single integer, namely the one obtained by interpreting $obs_C(s)$ as a binary number consisting of $|C|$ bits. The least significant bit is the one corresponding to the element added to C most recently. Note that if an element c is added to C, the integer representing $obs_C(s)$ can be turned into the one representing $obs_{C\cup\{c\}}(s)$ in constant time as it requires a multiplication by 2, a membership query, and, in case of a positive answer, an additional increment by 1. Similarly, if an element s is added to R, computing $obs_C(s)$ takes $O(|S|)$ steps as $|C|$ membership questions are required and $|C| \leq |S|$ by the second smallness condition.

We show first that it can be checked efficiently whether a given transition is contained in δ_T. Note that this can be used in EXTEND to test whether $s \in R$ because $s \in R$ if and only if $trans_C(s) \in \delta_T$ (see Lemma 3.7).

Lemma 4.4. *Let $T = (S, R, C)$ be an observation table constructed during a run of the learner. It can be checked in time $O(r \cdot |S|)$ whether $trans_C(s) \in \delta_T$ for a given tree $s \in \Sigma(S)$.*

Proof. This is based on a suitable representation of δ_T, which we describe first. For every tree $s' \in R$, let us identify $obs_C(s')$ with its integer representation in memory (as explained above). Now, let $f \in \Sigma_{(k)}$ and denote by δ_f the set of all $i_1 \cdots i_{k+1} \in \mathbb{N}^{k+1}$ such that $(i_1 \cdots i_k, f, i_{k+1}) \in \delta_T$. We represent δ_f as the rooted tree (in the graph-theoretic sense) whose edges are labelled with numbers in $obs_C(S)$, such that there is a path labelled $i_1 \cdots i_{k+1}$ from the root to a leaf if and only if $i_1 \cdots i_{k+1} \in \delta_f$.

Let $s = f[s_1, \ldots, s_k]$ be the tree in the statement of the lemma. Since $obs_C(s_1), \ldots, obs_C(s_k)$ are available in memory, the transition $trans_C(s)$ can be computed by looking up $obs_C(s_1), \ldots, obs_C(s_k)$ within $O(r)$ steps and using $|C| \leq |S|$ membership queries[3] to determine $obs_C(s)$. Now, let $trans_C(s) = (i_1 \cdots i_k, f, i_{k+1})$. Every node of the described representation of δ_T has out-degree at most $|S|$. Hence, it can be checked by means of $(k+1)|S|$ comparisons whether $i_1 \cdots i_{k+1} \in \delta_f$ (i.e., whether $(i_1 \cdots i_k, f, i_{k+1}) \in \delta_T$). Since $k \leq r$, this yields the claimed bound. □

[3] Recall that $|C| \leq |S|$ since T is small.

We now estimate how much time is spent in each call EXTEND(T, t) until one of the nonrecursive return statements is reached (not including the time required to execute these final return statements).

Lemma 4.5. EXTEND *can be implemented in such a way that, for each call* EXTEND(T, t) *performed by the learner, one of the two nonrecursive return statements is reached in time* $O(r \cdot |t| \cdot |S|)$ *where* $T = (S, R, C)$.

Proof. In a bottom-up manner the subtrees of t are marked if they belong to S. We argue at the end of this proof that the test of whether a subtree s belongs to S can be implemented to run in time $O(r \cdot |S|)$. As soon as a subtree is reached that does not belong to S, it must necessarily be an element of $\Sigma(S)\backslash S$. This determines a decomposition $t = c[\![s]\!]$ as required in the description of EXTEND.

Next, we have to check whether $s \in R$, which is equivalent to testing whether $trans_C(s) \in \delta_T$ and thus by Lemma 4.4 takes $O(r \cdot |S|)$ steps. If $s \notin R$ then the execution of EXTEND has reached the nonrecursive return statement in the last line, so we suppose that $s \in R$.

Using a loop that ranges over all trees $s' \in S$ we now find the one for which $obs_C(s') = obs_C(s)$. By the assumption that all $obs_C(s)$, $s \in R$, are kept in memory, a comparison of $obs_C(s')$ with $obs_C(s)$ takes a constant number of steps (since $s, s' \in R$). Hence, s' is found in $O(|S|)$ steps. Afterwards, we check by means of a single membership query whether $U(c[\![s]\!]) = U(c[\![s']\!])$. If so, we replace s with s' (which takes a single pointer operation, as we shall see below) and continue the bottom-up process (which corresponds to the recursive call). In this way, each node of t is processed at most once.

It remains to be discussed how much time is required by the test of whether or not $s \in S$, where $s = f[s_1, \ldots, s_k]$ is a subtree of t whose own subtrees $s_1, \ldots, s_k$ are known to be in S as they have already been processed. For an efficient implementation, we use a convenient representation of (the trees in) S. We represent S by a directed acyclic graph G consisting of $|S|$ nodes, i.e., an acyclic pointer structure in which equal subtrees are represented by the same node. (Note that G has indeed exactly $|S|$ nodes because $S \subseteq \Sigma(S)$, i.e., every subtree of a tree in S is itself in S.) During the bottom-up inspection and modification of t, each subtree which is an element of S is turned into a pointer to the respective node of G. In particular, the replacement of the subtree s by $s' \in S$ above is implemented by establishing a pointer rather than by copying.

Now, for the test of whether $s \in S$, we know that $s_1, \ldots, s_k$ are actually pointers that refer to nodes in G. The membership test can thus be performed in $O(r \cdot |S|)$ steps by simply checking, for each node of G, whether it is labelled with f and whether its $k \leq r$ children are identical to the nodes $s_1, \ldots, s_k$ refer to.

In summary, each node of t is processed in time $O(r \cdot |S|)$, which means that EXTEND(T, t) runs in time $O(r \cdot |t| \cdot |S|)$. □

If the learner is implemented strictly along the lines of the previous section, SYNTHESIZE(T) must be computed in each execution of its main loop. However, in most cases EXTEND adds only a single tree to R (and possibly to S) without changing the set of contexts. Hence, a more efficient implementation can avoid recomputing

SYNTHESIZE(T) in all but a few loop executions by simply updating the old fta. This is expressed formally in the lemma below.

Lemma 4.6. *Suppose the learner calls* EXTEND(T, t), *which returns an observation table T' of the form $T' =$* COMPLETE($S, R \cup \{s\}, C$). *Then the internal data structures of the algorithm can be updated and the fta* SYNTHESIZE(T') *can be computed from* SYNTHESIZE(T) *in time $O(r \cdot |S|)$. This includes the time needed to execute* COMPLETE.

Proof. Let $s = f[s_1, \ldots, s_k]$. We may assume that $trans_C(s)$ has already been computed (see the proofs of Lemmas 4.4 and 4.5). We now also store $obs_C(s)$ in memory in order to reflect the fact that the observation table has been changed. After that, it suffices to add $trans_C(s)$ to the set of transitions of SYNTHESIZE(T) and to check by means of $|S|$ comparisons whether $obs_C(s) \in obs_C(S)$. If not, we add s to S, i.e., update the graph described in the proof of Lemma 4.5, within $O(r)$ steps. (For this, we simply create a new node labelled f and establish $k \leq r$ edges from that node to $s_1, \ldots, s_k$.) Furthermore, in this case we include $obs_C(s)$ in the set of states—and in the set of final states if $s \in U$. Checking the latter requires one additional membership query (see also the paragraph preceding Lemma 3.3). We note here that the representation of δ_T described in the proof of Lemma 4.4 can be updated simultaneously by adding the new transition in $O(r \cdot |S|)$ steps (similar to the lookup procedure in the proof of Lemma 4.4). □

In the case where the observation table is extended by a new context, we simply recompute SYNTHESIZE(T). The following lemma shows how much time this takes.

Lemma 4.7. *Suppose the learner calls* EXTEND(T, t), *which returns an observation table T' of the form $T' =$* COMPLETE($S \cup \{s\}, R, C \cup \{c\}$). *Then the internal data structures of the algorithm can be updated and the fta* SYNTHESIZE(T') *can be computed in time $O(r \cdot |S| \cdot |R|)$. This includes the time needed to execute* COMPLETE.

Proof. We first update the internal representation of the observation table entries by turning every $obs_C(s), s \in R$, into $obs_{C\cup\{c\}}(s)$. As pointed out earlier, this takes a constant number of steps for each $s \in R$, and thus $O(|R|)$ steps in total. Furthermore, for every $s \in R$ we check by means of $|S|$ comparisons whether $obs_C(s) \in obs_C(S)$, and add s to S within $O(r)$ steps if not. Thus, this task requires $O((r + |S|) \cdot |R|)$ steps in total.

Now, we compute SYNTHESIZE(T'). Clearly, constructing the transition table is the most time-consuming part (as $|R| \geq |S|$). For each $s = f[s_1, \ldots, s_k] \in R$, $trans(s)$ can be constructed in time $O(r)$ as we mainly have to look up the values $obs_{C\cup\{c\}}(s_1), \ldots, obs_{C\cup\{c\}}(s_k), obs_{C\cup\{c\}}(s)$, thus yielding $O(r \cdot |R|)$ steps in total.

While the first two steps run in time $O(r \cdot |R|)$, we must not forget to recompute the representation of $\delta_{T'}$ employed in Lemma 4.4. This can be done within the claimed time bound $O(r \cdot |S| \cdot |R|)$ since, as observed in the proof of Lemma 4.6, a single transition can be added in time $O(r \cdot |S|)$. □

We can now determine the running time of the learner as a whole.

Theorem 4.8. *Let $A = (\Sigma, Q, \delta, F)$ be the minimal partial fta recognizing U. The learner runs in time $O(r \cdot |Q| \cdot |\delta| \cdot (|Q| + m))$, where m is the maximum size of counter-examples returned by the teacher and r is the maximum rank of symbols in Σ.*

Proof. As argued in the proof of Theorem 4.3, every observation table $T = (S, R, C)$ constructed by the learner satisfies $|S| \leq |Q|$ and $|R| \leq |\delta|$ (see also the example discussed at the end of Section 3). In particular, the statement "`return` COMPLETE($S \cup \{s\}, R, C \cup \{c\}$)" in EXTEND is executed at most $|Q|$ times in total. By Lemma 4.7 the overall time spent on this, together with the required updatings, respectively recomputations, of SYNTHESIZE(T), is $O(r \cdot |Q|^2 \cdot |\delta|)$.

Similarly, the statement "`return` COMPLETE($S, R \cup \{s\}, C$)" in the last line of EXTEND is executed at most $|\delta|$ times in total. By Lemma 4.6, the time required in each of these calls is $O(r \cdot |Q|)$. Multiplying by $|\delta|$, we obtain the upper bound $O(r \cdot |Q| \cdot |\delta|)$.

Since each of the two nonrecursive return calls of EXTEND results in the addition of at least one new element to either S or R, at most $2 \cdot |\delta|$ calls of EXTEND are executed by the main procedure (again, the reader may wish to have a look at the example discussed at the end of Section 3). By Lemma 4.5 the recursion which each of these calls gives rise to, ends after $O(r \cdot m \cdot |Q|)$ steps, yielding $O(r \cdot m \cdot |Q| \cdot |\delta|)$ steps in total. Summing up, we get the time bound $O(r \cdot |Q|^2 \cdot |\delta| + r \cdot m \cdot |Q| \cdot |\delta|) = O(r \cdot |Q| \cdot |\delta| \cdot (|Q| + m))$. □

We finally discuss in an informal manner how many queries will be made. Assume again that $A = (\Sigma, Q, \delta, F)$ is the minimal partial fta recognizing U, and let M denote the sum of the sizes of counter-examples returned by the teacher. Implementing the algorithm without any optimization, at most $|Q| + |\delta| + 1$ equivalence queries will be made. This is because every execution of the body of the main loop requires one equivalence query while adding at least one tree to S or R. The membership queries of the learner belong to three different categories. Filling the at most $|Q| \cdot |\delta|$ entries of the observation table takes $|Q| \cdot |\delta|$ membership queries in the worst case. Further membership queries are asked by EXTEND in order to extract the required information from a given counter-example. Since the counter-example gets smaller with each recursive call, a maximum of M membership queries is needed here, if we sum up over all executions of EXTEND. Finally, at most $|Q|$ membership queries in total are needed in order to determine the final states of the automata SYNTHESIZE(T) constructed during the run of the algorithm. Altogether, in the worst case the algorithm requires $|Q| + |\delta| + 1$ equivalence and $M + |Q| \cdot (|\delta| + 1)$ membership queries. (See the next section for a partial improvement of the former.)

5. Conclusion

Based on an approach by Angluin [Ang1], we have presented a learning algorithm that constructs the minimal partial (deterministic bottom-up) finite-state tree automaton (fta) recognizing an initially unknown regular tree language U. To gather the necessary information about the target language, the MAT model from [Ang1] is used: The learner has access to a *minimally adequate teacher* who can answer membership and equivalence queries.

During the construction of the minimal partial fta recognizing U, the learner avoids dealing with so-called dead states. This solves an open problem mentioned by Sakakibara in [Sak1], where the first extension of the algorithm by Angluin to regular tree languages was proposed. A previous learner proposed by ourselves in [DH] attacked this problem as well but solved it only partially.

The benefit of learning a partial fta instead of a total one depends entirely on the target language. If the language has no forbidden subtrees then nothing is won, but there are languages where a partial fta is much smaller. As an example, consider the alphabet $\Sigma = \{a^{(k)}, b^{(0)}\}$ and let the language U contain only the fully balanced tree of height n. Clearly, the minimal total fta recognizing U has $n+2$ states $q_0, \ldots, q_n, q_\bot$, where q_n is final, and some $(n+2)^k$ transitions. However, although a minimal partial fta uses only one state less, the number of transitions drops to $n+1$, namely $\{(b, q_0)\} \cup \bigcup_{i=1}^{n} \{(q_{i-1} \cdots q_{i-1}, a, q_i)\}$.

For simplicity, and in order to estimate the running time of the pure learning algorithm, the complexity of membership and equivalence queries were disregarded, i.e., queries were assumed to take constant time. Of course, from a practical point of view this assumption is questionable (to say the least). Moreover, it may be suspected that it will normally be easier for the teacher to deal with membership than with equivalence queries. As explained at the end of the previous section, our algorithm makes $|Q|+|\delta|+1$ equivalence queries while learning a minimal partial fta $A = (\Sigma, Q, \delta, F)$. In contrast, the algorithms in [Sak1] and [DH] require only $|Q|+1$ of them (but, in the worst case, a number of membership queries which is exponential in the size of A). Hence, the number of equivalence queries needed by our algorithm may be considered to be a disadvantage compared with the earlier approaches. We conjecture that this cannot be avoided in general without, as in [DH], letting the learner also check transition rules that involve dead states, thus spoiling the gain in efficiency.

Nevertheless, an easy modification of the algorithm yields a certain improvement regarding the number of equivalence queries: Instead of requesting a new counter-example for each execution of EXTEND, one first checks whether the previous counter-example continues to be one (i.e., the new synthesized fta gives the same answer as the old one). In this case no new counter-example would have to be asked for. The algorithm would thus extract from each counter-example all information it can make use of, rather than discarding a counter-example as soon as one piece of information has been extracted. Clearly, in the worst case this does not yield an improvement as the teacher may always give a counter-example revealing precisely one missing row of the observation table.

For many languages this strategy will nevertheless result in an improvement if the teacher selects counter-examples at random or even finds counter-examples providing as much information about U as possible. Unfortunately, there are languages for which such counter-examples do not exist. For instance, let $\Sigma_{(0)} = \{b\}$ and $\Sigma_{(2)} = \{a_1, \ldots, a_n\}$, and define $U = \{a_i[b, b] \mid i \in [n]\}$. Then the teacher will be forced to give all trees $a_i[b, b]$ as counter-examples, one after the other, since no counter-example contains another counter-example as a subtree. This may also be considered as an indication that the approach presented here is optimal in a certain sense. Before receiving the corresponding counter-example, the learner cannot know whether $a_i[b, b]$ is a forbidden subtree unless this is checked by means of a membership query. However, testing $a_i[b, b]$ for all $i \in [k]$ by means of membership queries is not an option because it is too costly in general as all of them might be forbidden subtrees.

Obviously, the discussion above gives rise to interesting questions that could be addressed in future work. In particular, it could be interesting to investigate properties of the language U that allow the teacher to select counter-examples in such a way that only a comparatively small number of counter-examples is needed.

Acknowledgments

We thank Joost Engelfriet and the referees for useful comments that have helped improve both the results and their presentation.

References

[Ang1] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[Ang2] Dana Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988.

[Ang3] Dana Angluin. Queries revisited. *Theoretical Computer Science*, 313:175–194, 2004.

[BR] Piotr Berman and Robert Roos. Learning one-counter languages in polynomial time (extended abstract). In *Proc. 28th Annual Symposium on Foundations of Computer Science*, pages 61–67. IEEE Press, Piscataway, NJ, 1987.

[COC] Rafael C. Carrasco, Jose Oncina, and Jorge Calera. Stochastic inference of regular tree languages. In V. Honavar and G. Slutzki, editors, *Proc. 4th International Colloquium on Grammatical Inference* (*ICGI '98*), pages 185–197. Volume 1433 of Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 1998.

[DH] Frank Drewes and Johanna Högberg. Learning a regular tree language from a teacher. In Z. Ésik and Z. Fülöp, editors, *Proc. 7th International Conference on Developments in Language Theory* (*DLT '03*), pages 279–291. Volume 2710 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2003.

[Dre1] Frank Drewes. Tree-based picture generation. *Theoretical Computer Science*, 246:1–51, 2000.

[Dre2] Frank Drewes. *Grammatical Picture Generation — A Tree-Based Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin, to appear.

[Eng] Joost Engelfriet. Graph grammars and tree transducers. In S. Tison, editor, *Proc. CAAP* 94, pages 15–37. Volume 787 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1994.

[Fer] Henning Fernau. Even linear matrix languages: formal language properties and grammatical inference. *Theoretical Computer Science*, 289:425–456, 2002.

[FK] H. Fukuda and K. Kamata. Inference of tree automata from sample set of trees. *International Journal of Computer and Information Sciences*, 13(3):177–196, 1984.

[FR] Amr F. Fahmy and Robert Roos. Efficient learning of real time one-counter automata. In K. P. Jantke, T. Shinohara, and T. Zeugmann, editors, *Proc. 6th International Workshop on Algorithmic Learning Theory* (*ALT '95*), pages 25–40. Volume 997 of Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 1995.

[GS1] Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

[GS2] Ferenc Gécseg and Magnus Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages. Vol. III: Beyond Words*, chapter 1, pages 1–68. Springer-Verlag, Berlin, 1997.

[Ish1] Hiroki Ishizaka. Inductive inference of regular languages based on model inference. *International Journal of Computer Mathematics*, 27:67–83, 1989.

[Ish2] Hiroki Ishizaka. Polynomial time learnability of simple deterministic languages. *Machine Learning*, 5:151–164, 1990.

[Lee] Lillian Lee. Learning of context-free languages: a survey of the literature. Report TR-12-96, Center for Research in Computing Technology, Harvard University, 1996.

[LJ] Leon S. Levy and Aravind K. Joshi. Skeletal structural descriptions. *Information and Control*, 39:192–211, 1978.

[Pap] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, MA, 1994.

[Sak1] Yasubumi Sakakibara. Learning context-free grammars from structural data in polynomial time. *Theoretical Computer Science*, 76:223–242, 1990.

[Sak2] Yasubumi Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97:23–60, 1992.

[Sha] Ehud Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.

[SY] Hiromi Shirakawa and Takashi Yokomori. Polynomial-time MAT learning of c-deterministic context-free grammars. *Transactions of the Information Processing Society of Japan*, 34:380–390, 1993.

[TW] James W. Thatcher and Jesse B. Wright. Generalized finite automata theory with an application to a decision-problem of second-order logic. *Mathematical Systems Theory*, 2:57–81, 1968.

[Yok] Takashi Yokomori. Learning nondeterministic finite automata from queries and counterexamples. In K. Furukawa, D. Michie, and S. Muggleton, editors, *Machine Intelligence*, volume 13, chapter 7, pages 169–189. Clarendon Press, Oxford, 1994.

Received June 22, 2004, *and in revised form January* 27, 2005, *and in final form January* 28, 2005.
Online publication August 3, 2005.