Proc. SAIS 2006, pp. 35-44

Extensions of a MAT Learner for Regular Tree Languages

Frank Drewes and Johanna Högberg
Department of Computing Science, Umeå University
{drewes,johanna}@cs.umu.se

Abstract. In an earlier paper, we proposed a learning algorithm for regular tree languages in the Minimal Adequate Teacher model and investigated its complexity from a theoretical perspective. Here, we focus on more practical issues. We discuss a concrete implementation made available on the web, which includes two extensions of the basic algorithm. In the paper, the usefulness of these extensions is studied in an experimental setting, by running the variants of the algorithm against target languages with different characteristics.

1 Introduction

This paper contributes to the field of algorithmic learning, where the aim is to derive a formal description (an automaton, say) of a language U which is only implicitly available in the form of examples or similar information. Often, this situation lacks natural termination criteria, which led to the notion of 'identification in the limit' [Gol67]. To obtain a finite time bound, one can resort to so-called Probably Accurate Learning. In this setting, real values ϵ and δ, representing the desired accuracy and confidence, respectively, are chosen from the interval $[0, \ldots, 1]$. The learning component (the learner) processes available examples until the probability that P_{err} exceeds ϵ is less than δ, where P_{err} is the probability that the learner will classify an element incorrectly [RN03].

Another approach is to consider a restricted class of languages and use active learning, where U is learned with the help of an oracle (the teacher). This makes it possible to avoid probabilistic arguments, resulting in algorithms that are guaranteed to discover a correct description of U in a finite number of steps. A popular model for such learning situations is the Minimal Adequate Teacher (MAT) model introduced by Angluin [Ang87]. Here, the teacher can answer two types of queries: a *membership query* asks whether a given object is a member of U. An *equivalence query* proposes an automaton A and asks whether the recognised language $L(A)$ is equal to U. If not, the teacher returns a *counterexample* – an object belonging to the symmetric difference of $L(A)$ and U.

Several learning algorithms based on the MAT model have been studied in the literature. In this paper, we continue our study of an efficient algorithm for learning regular tree languages. Thus, both the learner and the teacher deal with trees and (deterministic) bottom-up finite-state tree automata. The algorithm mentioned was proposed in [DH06], following ideas by Angluin and Sakakibara [Ang87, Sak90][1], with a focus on theoretical issues, namely a formal correctness proof and a detailed complexity analysis.

The present paper serves two purposes. Firstly, it shows how the ideas discussed in [DH06] can be turned into a concrete implementation without loosing efficiency. Secondly,

[1] For a more detailed discussion of related work and further references, see [DH06].

aiming at a reduction in the number of equivalence queries needed to learn a language, we propose two extensions and investigate their usefulness in an experimental setting.

In [DH06], the effort made by the teacher is disregarded; queries are assumed to be answered instantaneously. In a more realistic setting, one may expect that equivalence queries are considerably harder to answer than membership queries (although this need not be true for all applications). This motivates the investigation of extensions that reduce the number of equivalence queries needed, even at the expense of increasing the number of membership queries. Here, we propose and explore two such extensions. Intuitively, the learner described in [DH06] discards a counterexample t as soon as it has extracted one piece of information from it. It then continues by asking for a new counterexample. In a real setting, this behaviour can be wasteful because t may still be a counterexample. The learner can check this without the help of the teacher, and could thus reuse t as long as possible. This yields the first extension proposed in this paper: reuse each counterexample as long as possible; discard it only when it is recognised correctly. However, this behaviour may be wasteful as well. This is because, later on, and in the light of new facts, t may again become a counterexample.[2] Therefore, our second extension is: reuse each counterexample as long as possible, but also save all old counterexamples and return to them later.

In the worst case, neither of the two extensions will reduce the number of equivalence queries, even if the teacher finds counterexamples that contain as much information as possible (see [DH06]). However, we want to find out how the algorithm is likely to behave in practice. Our hypothesis is that it will benefit from reusing, and even more from saving counterexamples, on average in many practical cases. Therefore, we choose an experimental setting. We explore how the three variants of the learner behave when counterexamples are picked at random, using as U several tree languages with different characteristics. Our results indicate that the extensions indeed have the desired effect. The reuse of counterexamples leads to a significant reduction of the number of equivalence queries; saving old counterexamples increases this effect even more, though not to the same extent.

To be able to conduct the experiments, a Java implementation of the learner has been created (together with a component realizing the teacher). The implementation can be downloaded from `http://www.cs.umu.se/~johanna/learning`.

In the next section, we explain the learner proposed in [DH06]. In Section 3, we discuss a concrete implementation, as well as the two extensions mentioned. Section 4 explains the setup of our experiments, while Section 5 presents their results.

2 An informal explanation of the learner

This section discusses informally the learner introduced in [DH06], where a more formal presentation can be found. We begin by recalling trees and tree automata.

Trees Trees are built over ranked alphabets, i.e., each symbol f has a rank $k \in \mathbb{N}$. To indicate that f has rank k, we may write $f^{(k)}$. The set T_Σ of all trees (or terms) over a ranked alphabet Σ is defined as usual. It is the smallest set such that $f[t_1, \ldots, t_k] \in \Sigma$,

[2] Note how this mirrors a real learning situation: as we learn more, we may realize that the interpretation of experiences made earlier was incorrect, although it happened to lead to the correct conclusion.

for all $f^{(k)} \in \Sigma$ and $t_1, \ldots, t_k \in \mathrm{T}_\Sigma$. If $k = 0$, we write f rather than $f[]$. A set T of trees is called a tree language. We identify T with the corresponding predicate on trees, i.e., $T(t) = \mathit{true}$ if $t \in T$, and $T(t) = \mathit{false}$, otherwise. For a tree language T and a ranked alphabet Σ, we let $\Sigma(T) = \{f[t_1, \ldots, t_k] \mid f^{(k)} \in \Sigma,\ t_1, \ldots, t_k \in T\}$.

We reserve the symbol $\Box^{(0)} \notin \Sigma$ for a special purpose. A tree $c \in \mathrm{T}_{\Sigma \cup \{\Box\}}$ in which $\Box$ occurs exactly once as a leaf is called a *context*. The set of all contexts over Σ is denoted by C_Σ. For $c \in \mathrm{C}_\Sigma$ and $s \in \mathrm{T}_\Sigma$, $c[\![s]\!]$ denotes the tree obtained by substituting s for $\Box$ in c.

Tree automata A (deterministic, but possibly partial) bottom-up finite-state tree automata (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* (which are viewed as symbols of rank 0), $\delta\colon \Sigma(Q) \to Q$ is the (possibly partial) *transition function*, and $F \subseteq Q$ is the set of *accepting* or *final* states.

The canonical extension of δ to trees yields the partial function $\delta\colon \mathrm{T}_\Sigma \to Q$ with $\delta(t) = \delta(f[\delta(t_1), \ldots, \delta(t_k)])$ for $t = f[t_1, \ldots, t_k] \in \mathrm{T}_\Sigma$. Naturally, the set of trees accepted by A is $L(A) = \{t \in \mathrm{T}_\Sigma \mid \delta(t) \text{ is defined and belongs to } F\}$. For $t \in \mathrm{T}_\Sigma$, we also write $A(t)$ instead of $L(A)(t)$. A tree language of the form $L(A)$ is called a *regular tree language*.

The learner To discuss the learner, let U be some regular tree language to be learned. Thus, the aim is to build an fta A such that $L(A) = U$. As explained in the introduction, the learner is assumed to have access to a teacher, an oracle that can answer membership queries ("Is the tree t an element of U?") and equivalence queries ("Does the fta A recognise U? – If not, return a counterexample, i.e., a tree t such that $A(t) \neq U(t)$.").

In order to understand how the learner works, suppose $A = (\Sigma, Q, \delta, F)$ is an fta with $L(A) = U$. The states of A divide the subtrees of trees in U into finitely many equivalence classes: trees t, t' are equivalent if $\delta(t) = \delta(t')$. The major task of the learner is to detect a unique representative for each of these classes. It exploits the fact that $\delta(t) = \delta(t')$ implies $\delta(c[\![t]\!]) = \delta(c[\![t']\!])$ for every $c \in \mathrm{C}_\Sigma$. Thus, if t and t' are equivalent, then A accepts $c[\![t]\!]$ if and only if it accepts $c[\![t']\!]$, regardless of c. This means that t and t' must be considered inequivalent in every fta recognising U if $U(c[\![t]\!]) \neq U(c[\![t']\!])$ for some $c \in \mathrm{C}_\Sigma$. Conversely, if $U(c[\![t]\!]) = U(c[\![t']\!])$ for all $c \in \mathrm{C}_\Sigma$, then t and t' may safely be considered to be equivalent.[3]

From the given counterexamples, the learner extracts trees representing equivalence classes and, at the same time, contexts witnessing their pairwise inequivalence. These two sets of trees are called S and C, resp. Thus, at all times during the execution of the algorithm, and for each pair of distinct elements s, s' of S, there exists $c \in C$ with $U(c[\![s]\!]) \neq U(c[\![s']\!])$. In addition to the sets S and C, a set R of trees is collected, whose elements represent transitions. The triple $T = (S, R, C)$ is called an *observation table*.[4] The sets S and R are related by a nice invariant: we always have $S \subseteq R \subseteq \Sigma(S)$.

To see how the information collected in T can be used to build an fta, let $C = \{c_1, \ldots, c_n\}$, where the ordering on the elements of C is arbitrary but fixed. Given a tree $s \in R$, we let $\mathit{state}_C(s)$ denote the sequence $U(c_1[\![s]\!]) \cdots U(c_n[\![s]\!])$. This is the state we associate with s. We can easily make sure that T is *complete* in the sense that every state

[3] The formal version of this argument leads to the Myhill-Nerode theorem for regular tree languages.

[4] In the implementation, T is a real table with rows and columns indexed by R and C, resp. The cell in row t and column c contains the truth value $U(t[\![c]\!])$ retrieved by means of a membership query.

$state_C(s)$ with $s \in R$ is in $state_C(S) = \{state_C(s') \mid s' \in S\}$. For this, we use a procedure COMPLETE that adds each $s \in R$ with $state_C(s) \notin state_C(S)$ to S. Now, the state set of the automaton is given by $state_C(S)$. Every $s = f[s_1, \ldots, s_k] \in R$ gives rise to the transition $\delta(f[state_C(s_1), \ldots, state_C(s_k)]) = state_C(s)$. Naturally, the accepting states are the states $state_C(s)$ for which $s \in U$. In summary, the automaton synthesised from the (completed) observation table T is given by $A_T = (Q_T, \Sigma, \delta_T, F_T)$, where

- $Q_T = state_C(S)$ (as $state_C$ is injective on S, this yields $|S|$ pairwise distinct states),
- $\delta_T(f[state_C(s_1), \ldots, state_C(s_k)]) = state_C(s)$ for all $s = f[s_1, \ldots, s_k]$ in R, and
- $F_T = \{state_C(s) \mid s \in S \cap U\}$.

The overall structure of the learner is very simple; it repeatedly asks for counterexamples and uses them in order to enlarge T until the teacher indicates that the proposed fta is correct (by returning the special symbol '$\perp$' instead of a counterexample):

```
T = (S, R, C) := (∅, ∅, ∅);
loop
    t := COUNTEREXAMPLE(A_T);        (construct A_T, ask equivalence query)
    if t = ⊥ then return A
    else T := EXTEND(T, t)
```

The actual extraction of new information from t takes place when the call EXTEND(T, t) is executed. Since t is a counterexample, we have $U(t) \neq A_T(t)$. To find the cause for this discrepancy, the learner proceeds with a step-by-step simulation of a run of A_T on t. It repeatedly chooses a subtree $s = f[s_1, \ldots, s_k] \in \Sigma(S) \setminus S$, thus decomposing t into $c[\![s]\!]$. Now, there are three cases. If $s \notin R$ (case 1), the run of A_T aborts prematurely as soon as the computation arrives at the node labelled f, because $\delta_T(f[state_C(s_1), \ldots, state_C(s_k)])$ is undefined. In other words, s must be added to R.

The second and third cases deal with the situation where $s \in R$. Let s' be the element of S such that $state_C(s') = state_C(s)$. Thus, $A_T(c[\![s]\!]) = A_T(c[\![s']\!])$. Now, the learner checks (using membership queries) whether $U(c[\![s]\!]) = U(c[\![s']\!])$. If so (case 2a), the tree $c[\![s']\!]$ is a counterexample, too, and EXTEND repeats the whole process using $c[\![s']\!]$ instead of t. Otherwise (case 2b), we have $U(c[\![s]\!]) \neq U(c[\![s']\!])$. This means that s' is not equivalent to s, as is witnessed by the context c. Therefore, EXTEND adds s' to S and c to C. In pseudocode notation, this looks as follows:

```
procedure EXTEND(T, t) where T = (S, R, C)
    loop
        decompose t into t = c[[s]] where s = f[s_1, ..., s_k] ∈ Σ(S) \ S;
        if s ∉ R then return COMPLETE(S, R ∪ {s}, C)        (case 1)
        else
            let s' ∈ S be such that state_C(s') = state_C(s);
            if U(c[[s']]) = U(c[[s]]) then t := c[[s']]      (case 2a)
            else return COMPLETE(S ∪ {s}, R, C ∪ {c})        (case 2b)
```

Termination of EXTEND is guaranteed since, in case 2a, $c[\![s']\!]$ contains fewer subtrees in $T_\Sigma \setminus S$ than does t. A detailed correctness proof of the learner is found in [DH06], where it is shown that the fta returned by the learner is the minimal partial fta recognising U.

3 Implementation and extensions of the learner

An implementation of the learner could, of course, directly follow the description given in the previous section. However, the resulting implementation of EXTEND would be rather inefficient because the decomposition of the counterexample t into $c[\![s]\!]$, which would typically involve a recursive search for a subtree in $\Sigma(S)\setminus S$, is done in each iteration. Moreover, recursive tests for equality of trees should preferably be avoided as such tests occur frequently in order to check whether $s \in \Sigma(S) \setminus S$ and whether $s \in R$. In [DH06], an efficient variant of EXTEND is sketched to the extent necessary for estimating its (theoretical) complexity. The idea is to combine the loop of EXTEND and the repeated decomposition of a counterexample into a single bottom-up process. Another important issue is that the elements of R are stored in a shared structure, a directed acyclic graph (dag) in which each tree is represented by a unique node. Thus, the equality test for trees in R boils down to a comparison of references. Here, we describe roughly how this idea has been realized in our Java implementation of the learner, thus showing how the theoretical considerations of [DH06] can be turned into a concrete implementation.

The observation table is an instance of a class in which EXTEND is a method. Similarly, the dag is an instance of a class providing suitable lookup and update methods. The idea behind the modified version of EXTEND is to check cases 1, 2a, and 2b during the recursive decomposition of the counterexample t. For this, EXTEND is called with the arguments c (initially $\Box$) and $s = f[s_1, \ldots, s_k]$ (initially t). EXTEND starts by calling itself recursively with the arguments $c[\![f[\Box, s_2, \ldots, s_k]]\!]$ and s_1. This recursive call may return `null`, which indicates that the desired information was found and the table has been updated. Otherwise, the recursive call returns the tree $s'_1 \in S$ (as a reference to a node in the dag) such that $state_C(s'_1) = \delta_T(s_1)$. Further recursive calls examine $c[\![f[s'_1, \Box, s_3, \ldots, s_k]]\!]$ and s_2, yielding s'_2 (or `null`), and so on.

Eventually, for each s_i the corresponding tree $s'_i \in S$ stored in the dag has been found. Now, the question is whether $f[s'_1, \ldots, s'_k] \in R$, i.e., whether this tree is already represented in the dag. Since $s'_1, \ldots, s'_k$ are dag references, this can be checked efficiently by a lookup method that returns either the sought tree $\overline{s}$ (in the form of a dag reference) or `null`. Now, there are four possibilities corresponding to the three cases of the original version of EXTEND, and the additional possibility that $f[s'_1, \ldots, s'_k]$ is not only in R but even in S:

1. The lookup method returned `null`. In this case, $f[s'_1, \ldots, s'_k]$ is added to R and inserted into the dag (by creating a new f-node with references to the subtrees $s'_1, \ldots, s'_k$), and EXTEND returns `null`.
2. The lookup method returned $\overline{s} \neq$ `null`, where $\overline{s} \notin S$. Hence, there is a unique tree $s' \in S$ (again given in the form of a dag reference) such that $state_C(s') = state_C(\overline{s})$.
   2a. If $U(c[\![s']\!]) = U(c[\![\overline{s}]\!])$, then EXTEND returns s'.
   2b. Otherwise, $\overline{s}$ is added to S, c is added to C, and EXTEND returns `null`.
3. The lookup method returned $\overline{s} \neq$ `null`, where $\overline{s} \in S$. In this case, EXTEND simply has to return $\overline{s}$.

The corresponding pseudocode is given below.

```
method EXTEND(c, s), where s = f[s_1, ..., s_k]
    for i = 1, ..., k do
        [ s'_i := EXTEND(c[[f[s'_1, ..., s'_{i-1}, □, s_{i+1}, ..., s_k]]], s_i);
        [ if s'_i = null then return null;
    s̄ := dag.LOOKUP(f, s'_1 ··· s'_k);
    if s̄ = null then                                          (case 1)
        [ s' := dag.INSERT(f, s'_1 ··· s'_k);
        [ ADDTOTABLE(null,s',null);
        [ return null;
    elsif s̄ ∉ S
        [ let s' ∈ S be such that state_C(s') = state_C(s̄);
        [ if U(c[[s']]) = U(c[[s̄]]) then return s'             (case 2a)
        [ else                                                 (case 2b)
        [     [ ADDTOTABLE(s',null,c);
        [     [ return null;
    else return s̄;                                            (case 3)
```

The learner, as introduced in [DH06] and discussed up to this point, retrieves exactly one piece of information (i.e. a tree in S or R, yielding a state or a transition) from each counterexample given, before discarding it. As argued in [DH06], there are tree languages that make it impossible to provide more than one piece of information per counterexample. Thus, in general, we cannot even expect to benefit from adopting another strategy if the teacher is assumed to select "rich" counterexamples. However, from a practical point of view, it is nevertheless interesting to explore how much is typically gained by exploiting the received counterexamples in a more exhaustive way. This becomes particularly important in settings where the effort spent by the teacher in order to construct counterexamples is significantly larger than the effort spent in order to answer membership queries.

In the following, we consider two such extensions of the plain learner. The first one, which we may call the *reusing learner*, continues to process the given counterexample as long as it happens to be a counterexample. Eventually, the counterexample is recognised correctly, and only then it is discarded. The second extension, called the *saving learner*, extends this strategy by not only reusing the most recent counterexample as long as possible, but also saving it. The rationale behind this is that, just as in real life, how much information the learner can retrieve from a counterexample depends on how much it already knows. In other words, as the learner discovers some new state (i.e., inserts a new tree in S), a former counterexample may become a counterexample, again. Therefore, the saving learner revisits the stored examples each time the examination of another counterexample has changed S. Note that the saving learner can easily check whether a former counterexample has become a counterexample again. For this purpose, it stores the answer the fta gave when the tree was considered last (which, in fact, tells us whether or not the tree belongs to U). Later, the learner runs its current fta on the tree and compares the results. If they differ, the tree must be revisited. The teacher is not involved in this decision.

For a (very simple) example illustrating this effect, consider $\Sigma = \{f^{(2)}, g^{(1)}, a^{(0)}, b^{(0)}\}$. We want to build expressions in which the first subtree of every occurrence of f is a tree

a
b
$g[a]$
$f[a,a]$

(a) Step 1

	$f[\Box, a]$
a	$+$
$f[a,a]$	$-$
b	$-$
$g[a]$	$+$

(b) Step 2

	$f[\Box, a]$
a	$+$
$f[a,a]$	$-$
b	$-$
$g[a]$	$+$
$g[f[a,a]]$	$-$
$f[a,f[a,a]]$	$+$

(c) Step 3

Table 1: Observation tables during a run of the saving learner

over g and a, whereas there is no such restriction regarding the second subtree (except that it, recursively, has the same property). Since the initial fta accepts the empty language, the teacher may return $f[g[a], g[b]]$ as a counterexample. Reusing this counterexample as long as possible yields Table 1(a), where $S = \{a\}$, $R \setminus S = \{b, g[a], f[a,a]\}$, and $C = \emptyset$. The resulting fta simply accepts T_Σ. The teacher may now give the counterexample $f[f[b,a],a]$, which leads to Table 1(b) (where '$+$' and '$-$' stand for *true* and *false*, resp). In particular, the learner has discovered the context $f[\Box, a]$ that distinguishes a from $f[a,a]$ and b. However, now the fta cannot handle the first counterexample any more, because it does not (erroneously) identify b with a any more. Instead, b is (correctly) put into the equivalence class represented by $f[a,a]$, thus making $\delta(g[b])$ undefined. Hence, it makes sense to revisit the first counterexample, yielding Table 1(c), which is the final one. Note the rationale behind revisiting old counterexamples (as opposed to, e.g., testing randomly generated examples every now and then): the addition of a new context in step 2 made it impossible to accept any tree containing b or $f[a,a]$. However, the fact that these trees belong to R proves that they appeared as subtrees of positive counterexamples (see [DH06]). Hence, we obviously have to gain something by revisiting the saved counterexamples. (In less trivial examples, revisiting negative counterexamples may reveal new information as well.)

Clearly, the plain learner will always use at least as many counterexamples than the reusing one, and that one will always use at least as many as the saving learner. However, while the reusing learner, compared with the plain learner, does not have any significant disadvantage, the saving learner is less memory efficient since old counterexamples are stored even if they perhaps never become useful, again.

4 Experimental setup and hypotheses

As mentioned in the introduction, the learners and a suitable teacher have been implemented and made available on the web. Because of the restricted available space, the implementation cannot be discussed in detail here; the interested reader is referred to the web site mentioned earlier. However, let us remark that the implementation of the teacher is based on the well-known fact that, given two ftas A, A' (corresponding to the fta proposed by the learner and the one describing the target language), one can effectively construct

an fta B such that $L(B)$ is the symmetric difference of $L(A)$ and $L(A')$, i.e., the set of all counterexamples.) Subject to pragmatic restrictions regarding the size of counterexamples, an equivalence query is answered by returning a random element of $L(B)$.

The implementation has been used to study experimentally how the learners behave if they are applied to various target languages with different characteristics. In the following, we explain briefly some of these languages, why they have been chosen, and the expected behaviour of the learners. The actual results of the experiments are discussed in the next section. Regardless of the target language, one may expect that both the reusing and the saving learner ask more membership questions that the plain learner does. Intuitively, the reason for this is that `EXTEND` will ask membership queries (to find out whether $U(c[\![s']\!]) = U(c[\![\overline{s}]\!])$) at nodes where such questions were asked during prior runs as well. Even though this check is not useless (because the trees s' and $\overline{s}$ are not necessarily the same as in the earlier runs), one may expect that it will typically yield the same result.

SIMPLE ENGLISH The tree language SIMPLE ENGLISH is a small school book example of a simple English grammar inspired by [JM00]. As lexicon, we use a number of nonsensical words from Lewis Carroll's poem *Jabberwocky* [Car72]. The (minimal partial) fta recognising SIMPLE ENGLISH has 18 states and 54 transitions.

SIMPLE ENGLISH is included because of its resemblance to a natural language. Another reason for including it is the hypothesis that, in this case, the learner will not gain much by saving counterexamples. This is because, compared with the number of states, there are relatively few transitions. Even though revisiting old counterexamples may in certain situations reveal previously unknown states, what we typically expect to find are transitions (cf. the small example discussed towards the end of the previous section).

FORMULA The tree language FORMULA contains the parse trees of all sentences in a first-order predicate logic with terms of types *integer* and *real*, over a certain set of symbols (for details, see the web page mentioned above). Recognising FORMULA requires an fta with 14 states and 435 transitions.

This language is included because it represents an important category of tree languages occurring in many formal contexts (in contrast to SIMPLE ENGLISH, which stems from the linguistic area). The structural difference between the two examples is reflected by the fact that the transition table is not as sparse as the one for SIMPLE ENGLISH. For this reason, one may expect that saving counterexamples may be somewhat more beneficial for this target language. In fact, there are a number of situations where trees t, t' are allowed to occur in the same context although they belong to different syntactic categories. (For example, the operator *mod* accepts both integer and real operands in its first argument position.) Intuitively, this should give rise to situations such as the one discussed near the end of Section 3.

EXPRESSION The tree language EXPRESSION consists of all expressions over a certain set of operators. Expressions can be of four different types: *integer*, *real*, *string*, and *undefined*. There is a ternary operator *ifdef*, the idea being that it selects the second or third operand, depending on whether or not the first one is defined. As a consequence, the transition table of the corresponding fta is complete, i.e., the minimal partial fta coincides

Simple English						
	Eq	Mem	RQ	RT	SQ	ST
Plain	71.0	1480	-	-	-	-
Reuse	14.8	2230	17.0	40.2	-	-
Save	14.6	2230	17.0	40.4	0	0

Table 2: Statistics from 100 executions for the target language Simple English

Formula						
	Eq	Mem	RQ	RT	SQ	ST
Plain	447	11300	-	-	-	-
Reuse	240	14700	7.34	201	-	-
Save	227	14800	6.35	182	1.18	32.2

Table 3: Statistics from 100 executions for the target language Formula

Expression						
	Eq	Mem	RQ	RT	SQ	ST
Plain	106	3250	-	-	-	-
Reuse	18.6	12200	1.46	87.0	-	-
Save	13.0	13400	1.34	65.4	.280	27.0

Table 4: Statistics from 100 executions for the target language Expression

Random Monadic						
	Eq	Mem	RQ	RT	SQ	ST
Plain	51.5	831	-	-	-	-
Reuse	13.5	823	28.3	31.8	-	-
Save	7.81	878	13.8	15.7	18.8	20.0

Table 5: Statistics from 100 runs for the target language Random Monadic

with the minimal total one. (The fta has four states, corresponding to the types, and 103 transitions.) Hence, we expect the learner to take even greater advantage of saving counterexamples than is the case for Formula.

Random Monadic Our final example, the tree language Random Monadic, consists of at most twenty monadic trees over $\{f^{(1)}, a^{(0)}\}$ of height at most 40. The trees are chosen at random each time the language is used. Thus, the exact number of states and transitions needed to recognise Random Monadic varies, but the upper limits of 40 states and transitions will never be surpassed. Even though Random Monadic is a small language which can be learnt quickly by the plain learner, saving counterexamples is supposed to be quite beneficial since there is essentially no structural connection between the trees in Random Monadic, and, hence, none between the counterexamples either.

5 Results

Let us now turn to the results of our experiments. For each of the considered languages, statistics from 100 executions have been collected, using randomly selected counterexamples. Tables 2–5 show the resulting figures. In each table, the columns show the mean values of the numbers of equivalence queries (Eq) and membership queries (Mem) asked, states and transitions found by reusing a counterexample (RQ and RT, resp.), and states and transitions found by returning to a saved counterexample (SQ and ST, resp.). Tables including standard deviations can be found on the web. In the following, we denote by, e.g., $\mathrm{Eq}_{\mathrm{plain}}^{\mathrm{English}}$ the average number of equivalence queries used by the plain learner for the tree language Simple English.

Unsurprisingly, for all languages studied, reuse leads to a significant reduction of the average number of counterexamples required. As expected, the plain learner uses the least number of membership queries. However, the ratio $\mathrm{Eq}_{\mathrm{plain}}^{U}/\mathrm{Eq}_{\mathrm{reuse}}^{U}$ is always considerably

larger than the ratio $\mathrm{Mem}^U_{\mathrm{reuse}}/\mathrm{Mem}^U_{\mathrm{plain}}$.

By comparing, in Tables 3–5, the values $\mathrm{RQ}^U_{\mathrm{reuse}}$ and $\mathrm{RT}^U_{\mathrm{reuse}}$ with $\mathrm{RQ}^U_{\mathrm{save}}$ and $\mathrm{RT}^U_{\mathrm{save}}$, resp., we observe an interesting effect: To some extent, saving counterexamples diminishes the benefit of reusing them. This can be explained as follows. If a new state is discovered, the reusing learner is likely to find some of the corresponding transitions "by coincidence" while reusing one of the next counterexamples (rather than finding all of them in the present one). In contrast, the saving learner may discover the same transitions prior to requesting and examining the next counterexamples.

Interestingly, $\mathrm{Eq}^{\textsc{English}}_{\mathrm{reuse}}$ is much smaller than $\mathrm{Eq}^{\textsc{English}}_{\mathrm{plain}}$, whereas absolutely nothing is won by saving counterexamples in this case. As expected, we can observe that $\mathrm{Eq}^{\textsc{Formula}}_{\mathrm{save}}$ is strictly less than $\mathrm{Eq}^{\textsc{Formula}}_{\mathrm{reuse}}$. However, the reduction of approximately 5.4% is unexpectedly small. As the tables show, this situation changes if we look at Expressions and Random Monadic. In these cases, both reusing and saving counterexamples lead to a considerable reduction in the number of equivalence queries.

In conclusion, our results indicate that, if the learner from [DH06] is to be applied in practise, reusing counterexamples is a valuable technique. Depending on the structure of the learned language, saving counterexamples seems to be useful as well, but in this case the benefit is not as obvious. Future work should aim at formal results that are able to explain the findings of this paper.

Acknowledgment We thank the anonymous referees for constructive criticism.

References

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[Car72] Lewis Carroll. *Through the Looking-Glass, and What Alice Found There*. Macmillian & Co, London, 1872.

[DH06] Frank Drewes and Johanna Högberg. Query learning of regular tree languages: How to avoid dead states. *Theory of Computing Systems*, 2006. To appear.

[Gol67] E. Mark Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[JM00] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*. Prentice-Hall, New Jersey, 2000.

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 2003.

[Sak90] Yasubumi Sakakibara. Learning context-free grammars from structural data in polynomial time. *Theoretical Computer Science*, 76:223–242, 1990.