The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning

Malte Isberner[1], Falk Howar[2], and Bernhard Steffen[1]

[1] TU Dortmund University, Dept. of Computer Science,
Otto-Hahn-Str. 14, 44227 Dortmund, Germany
{malte.isberner,steffen}@cs.tu-dortmund.de
[2] Carnegie Mellon University, Silicon Valley,
Moffett Field, CA 94043, USA
howar@cmu.edu

Abstract. In this paper we present TTT, a novel active automata learning algorithm formulated in the Minimally Adequate Teacher (MAT) framework. The distinguishing characteristic of TTT is its redundancy-free organization of observations, which can be exploited to achieve optimal (linear) space complexity. This is thanks to a thorough analysis of counterexamples, extracting and storing only the essential refining information. TTT is therefore particularly well-suited for application in a runtime verification context, where counterexamples (obtained, e.g., via monitoring) may be excessively long: as the execution time of a test sequence typically grows with its length, this would otherwise cause severe performance degradation. We illustrate the impact of TTT's consequent redundancy-free approach along a number of examples.

1 Introduction

The wealth of model-based techniques developed in Software Engineering – such as model checking [10] or model-based testing [7] – is starkly contrasted with a frequent lack of formal models. Sophisticated static analysis techniques for obtaining models from a source- or byte-code representation (e.g., [11]) have matured to close this gap to a large extent, yet they might fall short on more complex systems: be it that no robust decision procedure for the underlying theory (e.g., floating-point arithmetics) is available, or that the system performs calls to external, closed source libraries or remote services.

Dynamic techniques for model generation have the advantage of providing models reflecting actual execution behavior of a system. *Passive* approaches (e.g., [21]) construct finite-state models from previously recorded traces, while *active* techniques (e.g., [2]) achieve this by directly interacting with ("querying") the system. In this paper, we focus on the latter; in particular, we consider Angluin-style active automata learning [2], or simply active automata learning.

Active automata learning allows to obtain finite-state models approximating the runtime behavior of systems. These models are *inferred* by invoking sequences of operations (so-called *membership queries*) on the system, and observing the system's response. The technique relies on the following assumptions:

B. Bonakdarpour and S.A. Smolka (Eds.): RV 2014, LNCS 8734, pp. 307–322, 2014.

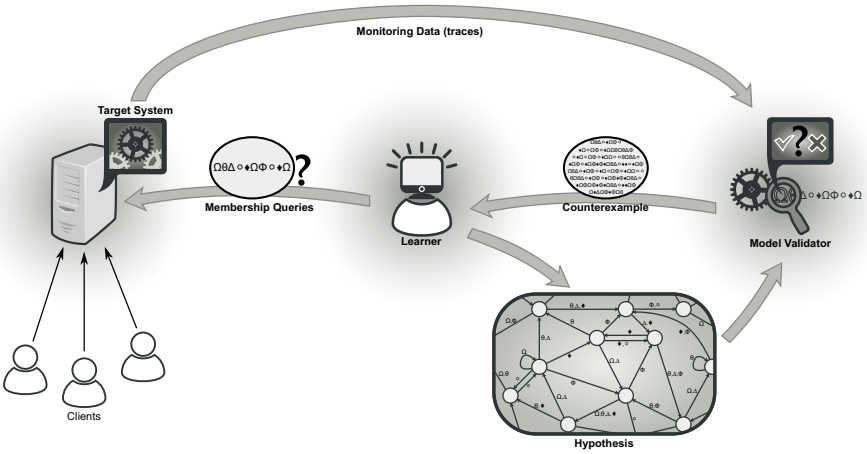

Fig. 1. Active automata learning setup with monitoring in the loop

- the set of operations that can be invoked has to be known a priori (e.g., from a public API),
- the reaction of the system to a membership query must be observable,
- the system has to behave deterministically, under the chosen output abstraction, and
- a way of *resetting* the system is required, i.e., subsequent membership queries have to be independent.

While some of these assumptions seem rather strong, the work of Cho *et al.* [8] on inferring botnet protocols has shown that active learning is a viable technique for obtaining useful models even in highly adverse scenarios.

A practical problem is that Angluin-style automata learning relies on *counterexamples* for model refinement, which are to be provided by an external source. Without such counterexamples, the inferred automata usually remain very small. Bertolino *et al.* [5] have thus suggested to combine active learning with *monitoring*, continuously validating inferred hypotheses against actual system traces. The setup is sketched in Figure 1: a learner infers an initial hypothesis through queries. The system is further instrumented to report its executions during regular operation to a *model validator* component in real-time. This component checks whether the monitored traces conform to the model. In case of a violation, a trace forming a counterexample is reported to the learner, which then refines its hypothesis.

The problem with this approach is that counterexamples obtained through monitoring can be very long. In the following section, by means of a simple example we sketch why virtually all existing active learning algorithms are not prepared to deal with such long counterexamples.

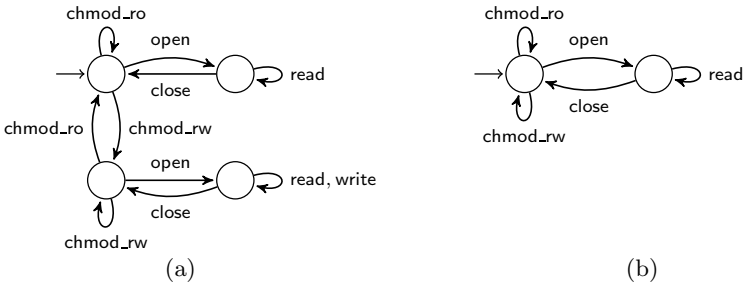

Fig. 2. (a) Example target system, (b) inferred approximation

1.1 Practical Motivation

Consider the behavioral fragment of the system depicted in Figure 2a, representing a stripped-down version of a resource access protocol. A resource can be opened, read from, and closed. Reading from and closing the resource is possible only if it has been opened before. A resource can only be opened if it is closed. Additionally, it is possible to write to a resource. This requires the access mode of this resource to be set to read/write previously (chmod_rw). It is not possible to change the access mode of an open resource.

Active automata learning aims at inferring such a behavioral model by executing test cases and observing the outcome (i.e., whether a sequence of operations is legal or not). The chmod_rw action does not have an *immediate* effect: to notice a difference, the resource needs to be opened *and then* written to. The model inferred by a learning algorithm might therefore be an incorrect *approximation*, as depicted in Figure 2b. If writing is a rare operation compared to reading, this incompleteness will go unnoticed for a long time. Only if the access mode is set to read/write *and* the resource is opened *and* written to, our hypothesis will fail to explain the observed behavior.

If we validate the inferred model by monitoring the actual system, the causal relationship between these three events might not at all be easily identifiable from a counterexample trace. Consider the following trace, not supported by the hypothesis in Fig. 2b:

$$\overbrace{\mathsf{open\ read\ close\ open\ read\ close\ chmod\_rw}}^{prefix}\ \underbrace{\mathsf{open\ read\ close\ open\ read\ close\ open\ write}}_{suffix}$$

When presented with such a counterexample, a learning algorithm needs to incorporate the contained information into its internal data structures. Most algorithms implement a variant of one of the following strategies: they identify that the chmod_rw transition in Figure 2b is wrongly routed, as a subsequent execution of the *suffix* part yields a different outcome than executing the suffix only (i.e., from the initial state). Rivest&Schapire's algorithm [26], for example, will add the suffix part to its internal data structures. Other algorithms, such as

Maler&Pnueli's [22] or NL* [6], will even add *all* suffixes of the counterexample to their data structures.

Alternatively, a learning algorithm might recognize that the *prefix* part corresponds to a state not yet reflected in the hypothesis, as a subsequent execution of the single action write is successful. Kearns&Vazirani's algorithm [20] will thus identify the new state using the prefix part. The original L* algorithm will even use *all* prefixes of the counterexample to ensure identification of a new state.

While this increases the space required by the internal data structures, the much graver issue is that the stored information is used for membership queries during subsequent refinements. The *redundant* open read close... sequences thus have to be executed again and again, even if no valuable information can be gained from them. Furthermore, as these redundancies appear both *before* and *after* the actual point of interest (chmod_rw), neither entirely prefix- nor entirely suffix-based approaches will avoid this problem.

In this paper, we present the TTT algorithm, which eliminates all future performance impacts caused by redundancies in counterexamples. In particular, after the refinement step is completed, the internal data structures maintained by TTT after processing the above counterexample would be completely indistinguishable from those resulting from processing the stripped-down counterexample chmod_rw open write.

Outline. After establishing notation in the next section, the main contribution is presented in Section 3: the description of the novel TTT algorithm, particularly highlighting the above-described approach. Section 4 reports on the results of a first experimental evaluation of TTT. Section 5 gives an overview on related work in the field of active automata learning, before Section 6 concludes the paper with an outlook on future work.

2 Preliminaries

2.1 Alphabets, Words, Languages

Let Σ be a finite set of *symbols* (we call such a set a (finite) *alphabet*). By Σ^* we denote the set of all finite *words* (i.e., finite sequences) over symbols in Σ, including the empty word ε. We define $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. The *length* of a word $w \in \Sigma^*$ is denoted by $|w|$. For words $w, w' \in \Sigma^*$, $w \cdot w'$ is the *concatenation* of w and w'. Unless we want to emphasize the concatenation operation, we will usually omit the concatenation operator $\cdot$ and just write ww'.

2.2 Deterministic Finite Automata

As DFA are one of the fundamental concepts in computer science, we will only give a very brief recount for the sake of establishing notation.

Definition 1 (DFA). *Let Σ be a finite alphabet. A* deterministic finite automaton (DFA) *$\mathcal{A}$ over Σ is a 5-tuple $\mathcal{A} = \langle Q^{\mathcal{A}}, \Sigma, q_0^{\mathcal{A}}, \delta^{\mathcal{A}}, F^{\mathcal{A}} \rangle$, where*

- $Q^{\mathcal{A}}$ *is a finite set of* states,
- $q_0^{\mathcal{A}} \in Q^{\mathcal{A}}$ *is the* initial state,
- $\delta^{\mathcal{A}} \colon Q^{\mathcal{A}} \times \Sigma \to Q^{\mathcal{A}}$ *is the* transition function, *and*
- $F^{\mathcal{A}} \subseteq Q^{\mathcal{A}}$ *is the set of* final *(or* accepting*)* states.

For $a \in \Sigma$ and $q \in Q^{\mathcal{A}}$, we call $q' = \delta^{\mathcal{A}}(q, a)$ the a-successor of q. Slightly abusing notation, we extend the transition function to words by defining $\delta^{\mathcal{A}}(q, \varepsilon) = q$ and $\delta^{\mathcal{A}}(q, wa) = \delta^{\mathcal{A}}(\delta^{\mathcal{A}}(q, w), a)$ for $q \in Q, a \in \Sigma, w \in \Sigma^*$.

The following shorthand notations will greatly ease presentation. For $q \in Q^{\mathcal{A}}$, we define the *output function* $\lambda_q^{\mathcal{A}} \colon \Sigma^* \to \{\top, \bot\}$ of q as $\lambda_q^{\mathcal{A}}(v) = \top$ iff $\delta^{\mathcal{A}}(q, v) \in F$ for all $v \in \Sigma^*$. We denote by $\lambda^{\mathcal{A}}$ the output function of $q_0^{\mathcal{A}}$. For the (extended) transition function, we use a notation borrowed from [20]: for $u \in \Sigma^*$, $\mathcal{A}[u] = \delta^{\mathcal{A}}(q_0^{\mathcal{A}}, u)$ is the state reached by u.

We conclude this section with an important property of DFA.

Definition 2 (Canonicity). *Let $\mathcal{A}$ be a DFA. $\mathcal{A}$ is* canonical *iff:*

1. $\forall q \in Q^{\mathcal{A}} : \exists u \in \Sigma^* : \mathcal{A}[u] = q$ *(all states are* reachable*)*
2. $\forall q \neq q' \in Q^{\mathcal{A}} : \exists v \in \Sigma^* : \lambda_q^{\mathcal{A}}(v) \neq \lambda_{q'}^{\mathcal{A}}(v)$ *(all states are pairwisely* separable*, and we call v a separator).*

It is well known that canonical (i.e., minimal) DFA are unique up to isomorphism [24].

3 The TTT Algorithm

In this section, we will present our main contribution: a new algorithm for actively inferring DFA. We start by giving a brief recount of active automata learning, defining the problem statement and sketching common assumptions and techniques. After this, we will introduce our running example that will accompany the explanation of the key steps, given in Section 3.4. We will also use this opportunity to provide the reader with a high-level idea of the interplay between TTT's data structures. Finally, we conclude the section with remarks on complexity.

3.1 Active Automata Learning in the MAT Model

In active automata learning, the goal is to infer an unknown target DFA $\mathcal{A}$ over a given alphabet Σ. For the remainder of the paper, we fix both the alphabet and the target DFA $\mathcal{A}$, which w.l.o.g. we assume to be canonical. The entity confronted with this task is called a *learner*, and to accomplish this task it may pose queries to a *teacher* (also called *Minimally Adequate Teacher*, MAT) [2]. Two kinds of queries are allowed: the *Membership Query* (MQ) of a word $w \in \Sigma^*$ corresponds to a function evaluation of $\lambda^{\mathcal{A}}(w)$. Whenever the learner has conjectured a hypothesis DFA $\mathcal{H}$, it may subject this to an *Equivalence Query* (EQ). Such a query either signals success (the hypothesis is correct) or yields a

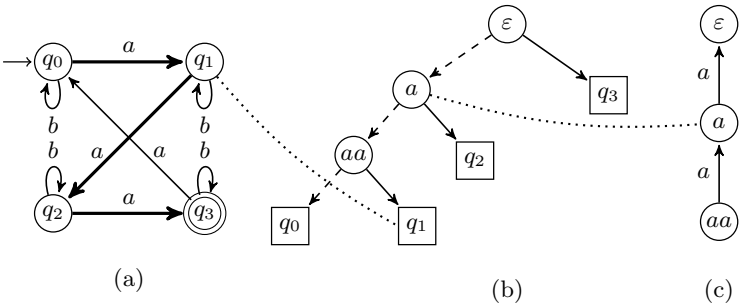

Fig. 3. Running example: (a) target DFA $\mathcal{A}$ and final hypothesis $\mathcal{H}_2$, (b) final discrimination tree $\mathcal{T}_2''$, (c) discriminator trie for final hypothesis

counterexample. A counterexample is a word $w \in \Sigma^*$ for which $\lambda^{\mathcal{A}}(w) \neq \lambda^{\mathcal{H}}(w)$. If presented with a counterexample, the learner needs to *refine* its hypothesis by asking additional membership queries, to conjecture a subsequent hypothesis $\mathcal{H}'$. These steps of hypothesis construction/refinement and equivalence checking are iterated until an equivalence query signals success. Note that in this setting, the learner is not in control over the appearance of counterexamples.

Many learning algorithms work by maintaining a finite, prefix-closed set $\mathcal{S}p \subset \Sigma^*$ identifying states in the target DFA $\mathcal{A}$. Each element of $\mathcal{S}p$ corresponds to a state of the hypothesis $\mathcal{H}$, and vice versa. For $q \in Q^{\mathcal{H}}$, we call its corresponding element $u \in \mathcal{S}p$ the *access sequence* of q, denoted by $\lfloor q \rfloor_{\mathcal{H}}$, and we have $\mathcal{H}[u] = q$. We extend this notation to arbitrary words, allowing to *transform* them into access sequences: for $w \in \Sigma^*$, we define $\lfloor w \rfloor_{\mathcal{H}} = \lfloor \mathcal{H}[w] \rfloor_{\mathcal{H}}$.

It is desirable to ensure that distinct prefixes in $\mathcal{S}p$ also correspond to distinct states in the target DFA $\mathcal{A}$. To accomplish this, the learner maintains a finite set of *distinguishing suffixes* (or *discriminators*) $\mathcal{D} \subset \Sigma^*$. It then constructs $\mathcal{S}p$ in such a way that, for any distinct pair of prefixes $u \neq u' \in \mathcal{S}p$, there exists a discriminator $v \in \mathcal{D}$ such that $\lambda^{\mathcal{A}}(u \cdot v) \neq \lambda^{\mathcal{A}}(u' \cdot v)$ has been observed through membership queries. Due to determinism in $\mathcal{A}$, this implies $\mathcal{A}[u] \neq \mathcal{A}[u']$. We denote the set of states of $\mathcal{A}$ that the learner has identified (or *discovered*) through words in $\mathcal{S}p$ by $\mathcal{A}[\mathcal{S}p] = \{\mathcal{A}[u] \mid u \in \mathcal{S}p\}$.

3.2 Running Example

We will now introduce our running example. We will also use this opportunity to briefly sketch the ideas behind the TTT algorithm's organization in terms of data structures.

Consider the DFA $\mathcal{A}$ in Figure 3a, defined over the alphabet $\Sigma = \{a, b\}$. This DFA accepts words containing $4i + 3$ a's, $i \in \mathbb{N}$. The rest of Figure 3 shows the state of TTT's eponymous data structures for inferring this DFA as its final hypothesis.

First, some of the transitions in (a) are highlighted in bold. These transitions form a *spanning Tree*, and they correspond to the prefix-closed set $\mathcal{S}p$ maintained

by TTT; here, $\mathcal{S}p = \{\varepsilon, a, aa, aaa\}$. Conversely, since paths in a tree are uniquely defined, the spanning tree itself *defines* the access sequences of states, and can be used to compute $\lfloor\cdot\rfloor_{\mathcal{H}}$. States of the hypothesis correspond to leaves of the binary tree shown in (b), the *discrimination Tree* (DT). This discrimination tree maintains the information on which discriminators in $\mathcal{D}$ separate states: for every distinct pair of states, a separator can be obtained by looking at the label of the *lowest common ancestor* of the corresponding leaves. Thus, the labels of inner nodes act as discriminators (or separators). As they form a suffix-closed set, they can compactly be stored in a trie [12] – the *discriminator Trie*, shown in (c): each node in this trie represents a word, and this word can be constructed by following the path to the root.[1] The root itself thus corresponds to the empty word ε.

3.3 Discrimination Trees

Discrimination trees (DT) were first used in an active learning context by Kearns and Vazirani [20]. They replaced the *observation table* used in previous algorithms [2,26]: whereas an observation table requires to pose a membership query for every pair $(u, v) \in \mathcal{S}p \times \mathcal{D}$, a DT is redundancy-free in the sense that only MQs that contribute to the distinction of states have to be performed.

As can be seen in Figure 3b, a DT $\mathcal{T}$ is a rooted binary tree. Inner nodes are labeled by discriminators $v \in \mathcal{D}$, and leaves are labeled by hypothesis states $q \in Q^{\mathcal{H}}$. The two children of an inner node correspond to labels $\ell \in \{\top, \bot\}$: we call them the $\bot$-child (dashed line) and the $\top$-child (solid line), respectively.

The nature of a discrimination tree is best explained by considering the operation of *sifting* a word $u \in \Sigma^*$ into the tree. Starting at the root of $\mathcal{T}$, at every inner node labeled by $v \in \mathcal{D}$ we branch to the $\top$- or $\bot$-child depending on the value of $\lambda^{\mathcal{A}}(u \cdot v)$. This procedure is iterated until a leaf is reached, which forms the result of the sifting operation. Sifting thus requires a number of membership queries bounded by the height of the tree.

3.4 Key Steps

In this section, we will present the key steps of TTT. We will use the example presented in Section 3.2 to clarify the effects of the presented steps. A complete and thorough description of the algorithm is beyond the scope of this paper. For technical details, we refer to the source code, which we made publicly available under the GPL license at `https://github.com/LearnLib/learnlib-ttt`.

Hypothesis Construction. It has already been mentioned in Section 3.1 that states are identified by means of a *prefix-closed set* $\mathcal{S}p \subset \Sigma^*$. Furthermore, these states correspond to leaves in the discrimination tree. Using this information, a hypothesis can be constructed from a set $\mathcal{S}p$ and a discrimination tree $\mathcal{T}$ as follows:

[1] This is a slight modification to the usual interpretation of a trie, which considers paths *from* the root.

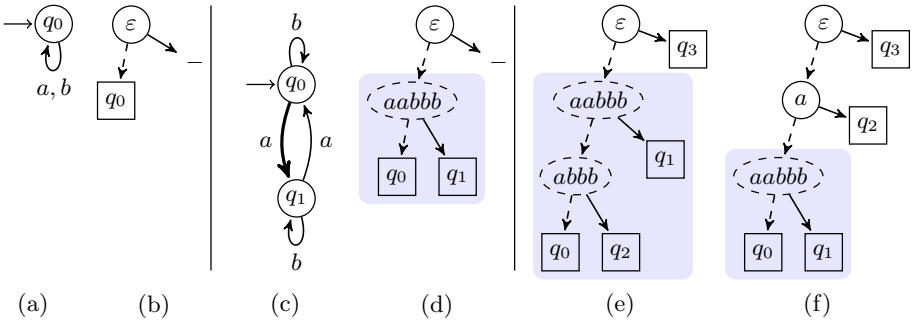

Fig. 4. Evolution of hypotheses and discrimination trees during a run of TTT: (a) initial hypothesis $\mathcal{H}_0$, (b) initial discrimination tree $\mathcal{T}_0$, (c) second hypothesis $\mathcal{H}_1$ (instable), (d) discrimination tree $\mathcal{T}_1$ for $\mathcal{H}_1$, (e) discrimination tree $\mathcal{T}_2$ for final hypothesis $\mathcal{H}_2$ showing blocks in blue, (f) discrimination tree $\mathcal{T}_2'$ after first discriminator finalization

- the *initial state* $q_0^{\mathcal{H}}$ is identified with the empty word $\varepsilon \in \mathcal{S}p$.
- transition targets are determined by means of *sifting*: given a state $q \in Q^{\mathcal{H}}$ identified by a prefix $u \in \mathcal{S}p$, its a-successor ($a \in \Sigma$) is determined by sifting ua into $\mathcal{T}$.
- a state $q \in Q^{\mathcal{H}}$ is in $F^{\mathcal{H}}$ if and only if its associated discrimination tree leaf is in the $\top$-subtree of the root of $\mathcal{T}$.

In the initial hypothesis $\mathcal{H}_0$ and discrimination tree $\mathcal{T}_0$, the setup is fairly simple: the initial state is the only state in the hypothesis, hence $\mathcal{S}p = \{\varepsilon\}$. As the corresponding DT leaf is in the $\bot$-subtree, q_0 is rejecting. Sifting $\varepsilon \cdot a$ and $\varepsilon \cdot b$ into $\mathcal{T}_0$ results in q_0. All transitions therefore form reflexive edges.

Hypothesis Refinement. Key to refining a hypothesis by means of a *counterexample*, i.e., a word $w \in \Sigma^+$ satisfying $\lambda^{\mathcal{H}}(w) \neq \lambda^{\mathcal{A}}(w)$, is Rivest&Schapire's observation [26,28] that w can be *decomposed* in the following way: there exist $u \in \Sigma^*$, $a \in \Sigma$, $v \in \Sigma^*$ such that $w = u \cdot a \cdot v$ and $\lambda^{\mathcal{A}}(\lfloor u \rfloor_{\mathcal{H}}\, a \cdot v) \neq \lambda^{\mathcal{A}}(\lfloor ua \rfloor_{\mathcal{H}} \cdot v)$.

Such a decomposition makes apparent that the words $\lfloor u \rfloor_{\mathcal{H}}\, a$ and $\lfloor ua \rfloor_{\mathcal{H}}$ lead to different states in $\mathcal{A}$ (as their output for v differs), but to the same state in $\mathcal{H}$. Therefore, the state $q_{old} = \mathcal{H}[ua]$ needs to be *split*. In the hypothesis, this is achieved by introducing a new state q_{new} with access sequence $\lfloor u \rfloor_{\mathcal{H}}\, a$ (note that this preserves prefix-closedness of $\mathcal{S}p$). In the discrimination tree, the leaf corresponding to q_{new} is split, introducing v as a *temporary* discriminator.

A possible counterexample for $\mathcal{H}_0$ could be $w = bbbaaabbb$, since $\lambda^{\mathcal{A}}(w) = \top \neq \lambda^{\mathcal{H}_0}(w)$. This counterexample contains a lot of redundant information: the b symbols exercise only self loops in $\mathcal{A}$, and thus do not contribute to the discovery of new states. Part – but not all – of the redundant information will be eliminated by the first counterexample analysis step, which yields the decomposition $\langle bbbb, a, aabbb \rangle$. Hence, a state with access sequence a is added to the next hypothesis $\mathcal{H}_1$ (Fig. 4c), and the corresponding discrimination tree $\mathcal{T}_1$ (Fig. 4d) contains a new inner node labeled with $aabbb$.

Hypothesis Stabilization. As a result of the previous step, it might happen that the constructed hypothesis contradicts information that is present in the discrimination tree. Consider, for example, the hypothesis $\mathcal{H}_1$ and its associated discrimination tree $\mathcal{T}_1$, shown in Figures 4c and d, respectively. State q_1 is identified by prefix $a \in \mathcal{S}p$. Since it is the $\top$-child of the inner node labeled with *aabbb*, we can deduce that $\lambda^{\mathcal{A}}(a \cdot aabbb) = \top$. However, the hypothesis $\mathcal{H}_1$ predicts output $\bot$.

An important observation is that the word *aaabbb* again forms a counterexample. This counterexample is treated in the same way as described in the previous step: by first decomposing it and then splitting the corresponding leaf in the discrimination tree, thus introducing a new state in the hypothesis. The resulting discrimination tree $\mathcal{T}_2$ can be seen in Figure 4e, and the corresponding hypothesis $\mathcal{H}_2$ is already the final one, i.e., the automaton shown in Figure 3a. We call a hypothesis like $\mathcal{H}_1$ *instable*, as it is refined without a call to an "external" equivalence oracle.

Discriminator Finalization. When comparing the inferred discrimination tree $\mathcal{T}_2$ (Fig. 4e) to the one shown in Figure 3b, one notices immediately that the discriminators occurring in $\mathcal{T}_2$ are much longer. This is due to the fact that the counterexample $w = bbbaaabbb$ contained a lot of redundant information, which is in part still present in the data structure. If these redundancies were not eliminated, subsequent refinements (if there were any) would frequently pose membership queries involving *aabbb* while sifting new transitions into the tree. To underline the dramatic impact this has, note that any word $aaab^i$, $i \in \mathbb{N}$, would have been a valid counterexample. Thus, the amount of redundancy that is present in these discriminators is generally unbounded.

TTT treats discriminators derived directly from counterexamples as *temporary* (represented by the dashed outlines of the inner nodes in Fig. 4). Furthermore, in Figures 4 d through f, parts of the discrimination tree are enclosed in rectangular regions. These correspond to maximal subtrees of the discrimination tree with temporary discriminators, and we refer to them as *blocks*. The TTT algorithm will *split* these blocks by subsequently replacing temporary discriminators at block roots with *final* ones. New final discriminators v' are obtained by prepending a symbol $a \in \Sigma$ to an existing final discriminator $v \in \mathcal{D}$, i.e., $v' = av$. This can be understood as adding a single node to the *discriminator trie* (cf. Fig. 3c). In Figure 4e, the effect of replacing the temporary discriminator *aabbb* in $\mathcal{T}_2$ with the final discriminator a is shown, resulting in the discrimination tree $\mathcal{T}_2'$. Note that the replacement discriminator does not need to partition the states in the same way as the temporary discriminators, but it needs to separate at least two states in the respective block. In particular, *aabbb* still occurs in $\mathcal{T}_2''$ (Fig. 4f), but *abbb* has vanished.

After replacing *aabbb* in $\mathcal{T}_2''$ with the final discriminator aa, the discrimination tree already shown in Figure 3b is obtained. As can be seen, it no longer contains any redundant information (i.e., b's) in any of the discriminators. Without discriminator finalization, this would have required the stripped-down, minimal counterexample *aaa* in the first place.

3.5 Complexity

We now briefly report the complexity of the TTT algorithm. In particular, we focus on three complexity measures:

- *Query complexity*, i.e., the number of overall membership queries posed by the algorithm.
- *Symbol complexity*, i.e., the total number of symbols contained in all these membership queries.
- *Space complexity*, i.e., the amount of space taken up by the internal data structures of the algorithm.

We neglect the time spent on internal computations of the algorithm (e.g., for organizing data structures). This is well-justified by existing reports on practical applications of automata learning, which usually mention the time required for either symbol executions [8], system resets [9], or the space taken up by the observation table data structure [4] as bottlenecks.

Query and Symbol Complexity. We limit ourselves to a brief sketch of query and symbol complexity. Basically, for both correctness and (query) complexity, the same arguments as for other active learning algorithms apply (cf., e.g., [20,28]). We assume that k is the size of the alphabet Σ, the target DFA $\mathcal{A}$ has n states, and the length of the longest counterexample returned by an equivalence query is m. TTT in the worst case requires $O(n)$ equivalence queries and $O(kn^2 + n \log m)$ membership queries, each of length $O(n+m)$. This pessimistic estimate is due to the fact that (degenerate) DTs can be of height $O(n)$ (cf. Fig. 3b). This worst-case query and symbol complexity coincides with Rivest and Schapire's algorithm [26], though we will see in Section 4 that in practice there is a huge gap between the two.

Space Complexity. Interesting from a theoretical perspective is the fact that the TTT algorithm exhibits *optimal* space complexity, not considering the (temporary) storage required for storing counterexamples. In general, the optimality becomes apparent when looking at Figure 3. All of the data structures are (based on) trees, which require an amount of space linear in the number of the leaves. Thus, the space required for the complete hypothesis (with all transitions, i.e., $\Theta(kn)$) dominates the overall space complexity.

Intuitively, it is obvious that every correct learning algorithm has to store the hypothesis (as it constitutes its output), and thus requires space in $\Omega(kn)$. Therefore, TTT has optimal space complexity. Furthermore, this space complexity is significantly below the space complexity of other algorithms, such as L^* [2], Rivest and Schapire's [26], or Kearns and Vazirani's [20]. These require space in $\Theta(kn \cdot (n+m))$, $\Theta(kn^2 + nm)$, or $\Theta(kn + nm)$, respectively.

Let us briefly remark that to formally prove space optimality, the above intuition is not sufficient. In particular, it does not take into account the reduction of the search space due to the restriction to *canonical* automata only. However, Domaratzki *et al.* [13] proved a lower bound of $f_k(n) \geq (k - o(1))n2^{n-1}n^{(k-1)n}$ on the number of distinct canonical DFA with n states over an input alphabet of size k. This implies that encoding a canonical DFA requires, on average,

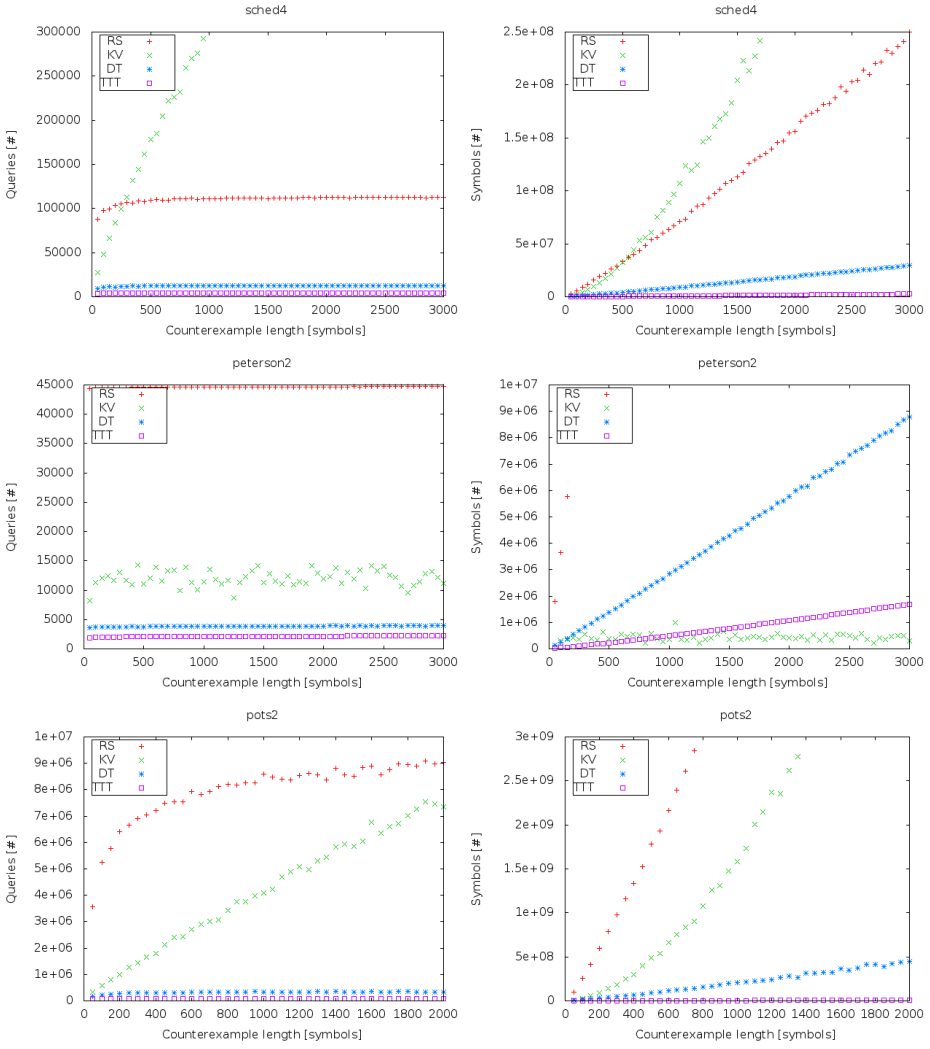

Fig. 5. Experimental results for the CWB examples `sched4`, `peterson2`, and `pots2` (top to bottom). Queries (left) and symbols (right) are shown as a function of the counterexample length.

$\Omega(\log f_k(n)) = \Omega(nk \log n)$ bits, which coincides with TTT's space complexity in the logarithmic cost model.

4 Experimental Evaluation

In this section, we report on an evaluation of our first implementation of TTT. We have implemented TTT in the LearnLib framework,[2] which is open source and can easily be extended. Furthermore, it comes with a number of standard algorithms, which facilitates performance comparison. The implementation of TTT, along with the examples used as experiments and the evaluation scripts, can be obtained from `https://github.com/LearnLib/learnlib-ttt`.

Target Systems. In order to assess the performance on systems with realistic structure, we learned models of systems distributed with the Concurrency Workbench:[3] Milner's scheduler for four processes (`sched4`, $n = 97, k = 12$), Peterson's mutual exclusion algorithm for two processes (`peterson2`, $n = 50$, $k = 18$), and a model of a telephony system with two clients (`pots2`, $n = 664$, $k = 32$).

Equivalence Queries. To reflect the setup shown in Figure 1, we randomly generated traces of fixed length on the target systems. If these traces were not supported by the hypothesis, we fed them as counterexamples to the learning algorithm. We let the length of these traces vary between 50 and 3000 (2000 for `pots2`), in increments of 50.

Metrics. We measured both the number of membership queries, and the number of symbols contained in all of these queries combined. We averaged over 10 runs to account for variations in the counterexample trace generation.

Comparison. We compared our implementation of TTT against algorithms shipped with LearnLib: Rivest and Schapire's algorithm [26] (RS), Kearns and Vazirani's algorithm [20] (KV), and the "discrimination tree" algorithm[4] [15] (DT), which can be described as TTT without finalizing discriminators (cf. Sec. 3.4). The former algorithm is based on an observation table, while the latter two are based on discrimination trees. All these algorithms have in common that they only add a single suffix from the counterexample to the data structure. In contrast, algorithms like L^* [2], Suffix1by1 [18], Maler and Pnueli's [22], or NL^* [6] add (nearly) *all* prefixes or suffixes of a counterexample to the observation table. We found that these algorithms were entirely infeasible (i.e., resulting in `OutOfMemoryError`s) for long counterexamples.

4.1 Results

The results of our evaluation on the three systems (top to bottom: `sched4`, `peterson2`, `pots2`) are displayed in Figure 5. Both the number of membership

[2] `http://www.learnlib.de/`

[3] `http://homepages.inf.ed.ac.uk/perdita/cwb/`

[4] This algorithm is also known as the *Observation Pack* algorithm.

queries (left column) and the total number of symbols (right column) are plotted as a function of the length of counterexample traces.

In terms of membership queries, TTT outperforms all other algorithms on all examples. When compared to the DT algorithm, the difference is comparatively small, with TTT requiring 25%–50% as many membership queries. However, this is still remarkable when considering that the main difference between TTT and DT is the *extra effort* for finalizing discriminators (cf. Sec. 3.4). We conclude from this that by discriminator finalization, we obtain "more general" discriminators, which lead to better-balanced trees than the very specific, long ones directly extracted from the counterexamples. When compared to the RS and KV algorithms, the difference in membership queries spans several orders of magnitude.

When looking at the number of symbols, TTT consistently beats DT. On the `sched4` and `peterson2` examples, we observe a reduction in the number of symbols by approximately one order of magnitude. On the `pots2` example, which is the largest of the systems we considered, there even is a 60× reduction!

The `peterson2` example poses a special case. Here, the number of membership queries apparently is nearly constant for all counterexample lengths, but the variation for the KV algorithm is considerable. In terms of symbols, the KV algorithm outperforms TTT when counterexamples consists of 800 symbols or more (for a length of 3000, TTT needs roughly 4.5× as many symbols as KV). Manual inspection of the model showed that it is structured in a DAG-like fashion, with only very few loops. Hence, counterexamples on this system contain relatively little redundant information. However, this was the only example we investigated[5] where KV performed that strongly in terms of symbols. Furthermore, especially the `pots2` example underlines that preferring the KV algorithm over TTT might be an extremely poor choice: for a counterexample length of 2000, KV on average requires 670× more symbol executions than TTT does. When considering Rivest&Schapire, this factor increases to up to 1100×.

5 Related Work

The MAT model for active automata learning was established by Angluin [2], along with the presentation of the famous observation table-based L^* algorithm. The technique gained major interest after it was discovered as a means of enabling model-based techniques in scenarios where no such models were available. Notable works in this direction include its application to model checking [25], and to model-driven test-case generation [14]. The practical applicability was further improved by adapting the L^* DFA learning algorithm to Mealy machines [14,27].

While much effort has been devoted to optimizations in practical scenarios (e.g., using various filters [23]), improvements at the "pure" algorithmic level are

[5] Other examples included randomly generated DFA, and instances of Figure 2a for up to 5 resources, which we do not report upon due to size constraints. All data necessary to run these experiments can be obtained via GitHub.

comparably rare. Maler and Pnueli [22] suggested adding all suffixes of the counterexample to the table. Rivest and Schapire [26] found that adding a single suffix was sufficient, and that this suffix could be determined using a binary search. It has been observed that this leads to non-canonical intermediate hypotheses [28]. Several heuristics have thus been proposed to maintain suffix-closedness of the discriminators, such as Shahbaz's algorithm and SUFFIX1BY1 [18].

Kearns and Vazirani [20] were the first to employ a *discrimination tree*. A general framework for active learning named *Observation Packs* was introduced by Balcazar *et al.* [3]. This framework provides a unifying view on the aforementioned algorithms. Its name has been adopted for an algorithm developed by Howar [15], which can be summarized as combining the discrimination tree data structure with Rivest and Schapire's counterexample analysis.

A fairly recent contribution in the classical scenario of black-box inference of regular languages is the NL* algorithm [6], inferring NFA instead of DFA. These NFA may be exponentially more succinct than the corresponding DFA, and can in such cases be learned with less membership queries. However, the number of required equivalence queries grows from linear to quadratic. Furthermore, it is unclear how (or even if) the NL* algorithm could be adapted to infer Mealy machines.

6 Conclusion

We have presented TTT, an active automata learning algorithm which stores the essential data in three tree-like data structures: a *spanning tree* defining unique access sequences, embedded into the hypothesis' transition graph, a *discrimination tree* for distinguishing states, and a *discriminator trie* for storing the suffix-closed set of discriminators. This leads to an extremely compact representation, as it strips all the information down to the *essentials* for learning. In fact, the combined space required for all data structures is asymptotically the same as the size of the hypothesis, $\Theta(kn)$. We demonstrated the effects of this *redundancy-free* data structure on a number of examples. TTT outperformed other learning algorithms when considering the total number of membership queries on all of the examples. When considering the number of symbols, Kearns&Vazirani's algorithm in fact outperformed TTT in one out of three systems (even though it required a much higher number of membership queries). However, the system in question was the smallest one, and on all other systems, Kearns&Vazirani's algorithm performed poorly. On average, TTT yields a one to two orders of magnitude reduction in terms of symbol executions, and a 50%–75% reduction in terms of membership queries when compared to the Observation Pack algorithm, which ranks second. We conclude that the "cleanup" of the internal data structures that TTT performs is well worth the extra effort.

6.1 Future Work

There are several lines of work we want to explore. The first one is to further investigate the impact of TTT in practical setups. A necessary step for this will